



HAL
open science

JFLA 2024 - 35es Journées Francophones des Langages Applicatifs

Delphine Demange, Adrien Guatto

► **To cite this version:**

Delphine Demange, Adrien Guatto. JFLA 2024 - 35es Journées Francophones des Langages Applicatifs. 35es Journées Francophones des Langages Applicatifs (JFLA 2024), pp.1-328, 2024. hal-04407194

HAL Id: hal-04407194

<https://inria.hal.science/hal-04407194v1>

Submitted on 20 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

35^{ÈMES} JOURNÉES FRANCOPHONES
DES
LANGAGES APPLICATIFS

ACTES DE LA CONFÉRENCE
JFLA 2024

DU 30 JANVIER AU 2 FÉVRIER 2024
SAINT-JACUT-DE-LA-MER – CÔTES-D'ARMOR

ÉDITÉ PAR

DELPHINE DEMANGE

UNIV RENNES, IRISA, CNRS, INRIA, RENNES

ADRIEN GUATTO

UNIVERSITÉ PARIS CITÉ, IRIF, PARIS

ÉVÉNEMENT ORGANISÉ AVEC LE SOUTIEN DE

Inria



DATE DE PUBLICATION : JANVIER 2024

PRÉAMBULE

Préface

Les JFLA réunissent chaque année, dans un cadre convivial, concepteurs, théoriciens, développeurs et utilisateurs des langages fonctionnels, des assistants de preuve et des outils de vérification de programmes. Le spectre des travaux présentés aux JFLA est très large : il touche les aspects les plus théoriques de la conception des langages applicatifs jusqu'à leurs applications industrielles. La 35^{ème} édition des JFLA se déroulera à l'Abbaye de Saint-Jacut, à Saint-Jacut-de-la-Mer, dans les Côtes-d'Armor.

Cette année, nous avons sélectionné 10 articles de recherche longs, 7 articles de recherche courts, et 8 démonstrations de prototypes. Cette sélection a été réalisée par les membres du comité de programme, que nous remercions chaleureusement, à partir de 34 articles initialement soumis. Le programme de cette édition couvre des sujets variés relevant tous de la programmation en général et de son versant fonctionnel en particulier. Les contributions sélectionnées marient allègrement sémantique mathématique et conception de langages, sûreté et performance du logiciel, types précis et théorèmes fondamentaux. Que vous aimiez vérifier formellement des structures de données sophistiquées ; analyser statiquement d'immenses programmes ; écrire du code concurrent retors en OCaml 5 ; ou bien d'autres choses encore, vous devriez trouver de quoi piquer votre curiosité parmi les articles qui emplissent ces pages.

Nous aurons aussi le grand plaisir d'écouter plusieurs exposés invités. Guillaume Baudart et Christine Tasson présenteront leurs contributions scientifiques récentes dans le domaine de la programmation réactive probabiliste, autour du langage ProbZelus. Marie Kerjean, Micaela Mayero et Pierre Rousselin nous feront part de leur retour d'expérience quant à l'utilisation de l'assistant de preuve Coq pour l'enseignement des mathématiques. Matthieu Sozeau nous offrira un tour d'horizon des différentes facettes du projet MetaCoq d'implémentation de Coq en Coq. Enfin, Jules Villard nous présentera les dernières avancées de la plateforme d'analyse statique Infer développée et utilisée dans un cadre industriel.

Ces actes regroupent les résumés des exposés invités, ainsi que les articles sélectionnés. Nous vous en souhaitons une bonne lecture !

Nous remercions, au premier chef, les auteurs de tous les articles soumis cette année aux JFLA. Le grand nombre d'articles soumis témoigne de la belle richesse et de la forte diversité des activités de recherche dans nos thématiques scientifiques. Nous tenons aussi à exprimer nos sincères remerciements aux rapporteurs et aux membres du comité de programme. Leur travail essentiel de relecture, de rapport, et de suivi des révisions des articles est remarquable. Nous remercions nos chers orateurs et oratrices invitées, qui nous feront le plaisir et l'honneur de partager leur expertise, leurs connaissances, et de nous communiquer leur enthousiasme !

Ces journées ne pourraient avoir lieu sans l'implication et le travail de plusieurs personnes aux fonctions essentielles de support à la recherche. Merci à Édith Blin (SCM, Inria Rennes) et Lydie Mabil (SAER, Inria Rennes) pour leur implication dans le travail de planification, d'organisation et de suivi de gestion de l'événement. Merci à Laurent Briens (SAF, Inria Rennes) pour son suivi financier exemplaire. Merci à Hélène Lowinger (IES, Inria), de nous avoir épaulés lors de la publication des actes sur le portail institutionnel en ligne, participant ainsi à la garantie d'un accès libre et pérenne aux actes de la conférence.

Nous remercions enfin les entreprises et organismes mécènes. Leur soutien financier permet de subventionner la venue de plusieurs participant·es, doctorant·es et étudiant·es. L'implication des mécènes souligne aussi la pertinence et l'intérêt des travaux et résultats de recherche scientifique exposés pendant ces journées.

Nous vous souhaitons, à toutes et tous, de passer d'excellentes JFLA, conviviales et stimulantes, riches de rencontres, de découvertes, et de discussions !

Delphine Demange et Adrien Guatto

Comité d'organisation

Delphine Demange	Univ Rennes, Inria, CNRS, IRISA, Rennes	Présidente
Adrien Guatto	Université Paris Cité, IRIF, Paris	Vice-président
Edith Blin	Inria, Rennes	Communication et événementiel
Lydie Mabil	Inria, Rennes	Assistante d'équipes de recherche

Comité de programme

Guillaume Baudart	Inria, ENS Paris, Paris
Sylvie Boldo	Inria, LMF, Orsay
Adrien Champion	OCamlPro, Paris
Arthur Charguéraud	Inria, Strasbourg
Raphaëlle Crubillé	CNRS, Université d'Aix-Marseille, Marseille
Frédéric Dabrowski	LIFO, Université d'Orléans, Orléans
Pierre-Évariste Dagand	CNRS, IRIF, Paris
Delphine Demange	Univ Rennes, Inria, CNRS, IRISA, Rennes
Jean-Christophe Filliâtre	CNRS, LMF, Orsay
Hugo Férée	Université Paris Cité, IRIF, Paris
Adrien Guatto	Université Paris Cité, IRIF, Paris
Chantal Keller	Université Paris-Orsay, LMF, Orsay
Dominique Larchey-Wendling	CNRS, Université de Lorraine, LORIA, Nancy
Assia Mahboubi	Inria, Nantes
Luc Maranget	Inria, Paris
Raphaël Monat	Inria, Lille
Damien Pous	CNRS, ENS Lyon, Lyon
Lionel Rieg	Verimag, Grenoble-INP – Ensimag, Grenoble
Jocelyn Sérot	Institut Pascal, Université Clermont-Auvergne, Clermont-Ferrand
Sophie Tourret	Inria, Nancy
Boris Yakobowski	AdaCore, Paris
Yannick Zakowski	Inria, Lyon

Autres rapporteurs

Guillaume Bertholon	Université de Strasbourg, Strasbourg
Evan Cavallo	Université de Göteborg, Göteborg
Jonathan Protzenko	Microsoft Research Redmond, Redmond

Sommaire

Préambule

Exposés invités

Programmation réactive probabiliste.....	10
<i>Guillaume Baudart et Christine Tasson</i>	
Utilisation de Coq pour l'enseignement en Licence 1 et en seconde	12
<i>Micaela Mayero et Pierre Rousselin</i>	
MetaCoq : de la métaprogrammation à l'extraction certifiée pour Coq	15
<i>Matthieu Sozeau</i>	
Infer: a compositional and extensible platform for static analysis	16
<i>Jules Villard</i>	

Structures de données

De l'avantage de nuancer les décisions binaires.....	20
<i>Claude Marché et Denis Cousineau</i>	
Comparing EventB, {log} and Why3 Models of Sparse Sets	40
<i>Maximiliano Cristia et Catherine Dubois</i>	
Rough Pearl: Manufacturing Cons-Cells	51
<i>Pierre-Évariste Dagand, Pierre Letouzey et Ellenor Fatemeh Taghayor</i>	

Sûreté du logiciel

Réutilisation de caches et d'invariants pour l'analyse statique.....	64
<i>Mamy Razafintsialonina, David Bühler, Antoine Miné, Valentin Perrelle et Julien Signoles</i>	
Alias : pointeurs espionnés en série	85
<i>Tristan Le Gall, Jan Rochel, Florian Faissole, Julien Signoles et Denis Cousineau</i>	
À la recherche de tous les vrais bugs.....	105
<i>Arthur Correnson</i>	
Correct tout seul, sûr à plusieurs	116
<i>Clément Allain et Gabriel Scherer</i>	

Compilation

Chameleon : un minimiseur de programmes pour et en OCaml.....	136
<i>Milla Valnet, Nathanaëlle Courant, Guillaume Bury, Pierre Chambart et Vincent Laviro</i>	
Correct, Fast LR(1) Unparsing	146
<i>François Pottier</i>	
Source-to-Source Optimizations using Separation Logic	166
<i>Guillaume Bertholon, Arthur Charguéraud et Thomas Koehler</i>	

Chamois : agile development of CompCert extensions for optimization and security	171
<i>David Monniaux et Sylvain Boulmé</i>	
L'interprète, le JIT et la licorne	177
<i>Frédéric Recoules et Sébastien Bardin</i>	
<hr/>	
Sémantique	
<hr/>	
Resource Categories from Differential Categories	190
<i>Lison Blondeau-Patissier</i>	
Skeletal Semantics for a Fragment of Python	205
<i>Martin Andrieux et Alan Schmitt</i>	
Liveness Properties in Geometric Logic for Domain-Theoretic Streams	219
<i>Colin Riba et Solal Stern</i>	
Executable semantics of Arm's Architecture Specification Language	240
<i>Hadrien Renaud</i>	
Towards a linear functional translation for borrowing	244
<i>Sidney Congard</i>	
<hr/>	
Langages de programmation	
<hr/>	
Type Inference of Polymorphic and Overloaded Functions	256
<i>Mickaël Laurent</i>	
Stimulus : un langage synchrone à contrainte	260
<i>Bertrand Jeannot, Etienne Closse, Fabien Gauche et Daniel Weil</i>	
Un prototype de système de types graduels ensemblistes pour Elixir	264
<i>Guillaume Duboc</i>	
Destination-passing style programming : a Haskell implementation	268
<i>Thomas Bagrel</i>	
Modular efficient deconstruction with typed pointer reversal	288
<i>Jean Caspar et Guillaume Munch-Maccagnoni</i>	
<hr/>	
Preuve formelle, ingénierie de la preuve	
<hr/>	
Towards the Fundamental Theorem of Calculus for the Lebesgue Integral in Coq	300
<i>Reynald Affeldt et Zachary Stone</i>	
Packaging Proofs with Why3find	305
<i>Loïc Correnson</i>	
A diagram editor to mechanise categorical proofs	318
<i>Ambroise Lafont</i>	
<hr/>	
Annexes	
<hr/>	

EXPOSÉS INVITÉS

Programmation réactive probabiliste

Guillaume Baudart¹ et Christine Tasson²

¹Université Paris Cité, INRIA, Paris, 75013, France

²ISAE-SUPAERO, Toulouse, France

Les langages de programmation synchrones ont été introduits pour la conception de systèmes embarqués. Aujourd’hui le langage industriel SCADE [Est19] est couramment utilisé pour la conception de systèmes embarqués comme les commandes de vol d’un avion ou le système de freinage d’un train. Lors de la conception de tels systèmes, il est primordial de prendre en compte l’incertitude due à l’environnement, aux capteurs, ou aux défauts matériels. Malheureusement, les langages synchrones existants n’offrent que peu de support pour modéliser des comportements probabilistes. Les langages de programmation probabilistes permettent de décrire des modèles probabilistes et proposent des méthodes automatiques pour inférer les paramètres du modèle à partir d’observations statistiques. ProbZelus est un langage qui combine programmation réactive synchrone et programmation probabiliste.

Dans la première partie de cet exposé nous présentons ProbZelus [BMA⁺20], une extension probabiliste du langage synchrone Zelus [BP13] (un langage académique proche de SCADE) qui permet de décrire des modèles probabilistes réactifs en interaction avec un environnement observable. Lors de l’exécution, un moteur d’inférence bayésien permet d’apprendre les distributions de paramètres du modèle à partir d’observations. Nous introduisons ProbZelus avec quelques exemples concrets avant de discuter la conception du langage. Nous détaillons ensuite les algorithmes d’inférence semi-symbolique au cœur de la machine d’exécution qui mêlent méthodes d’échantillonnage approximatives et calculs symboliques exacts.

Dans la seconde partie de cet exposé, nous présentons la sémantique formelle de ProbZelus. La sémantique des programmes synchrones peut être exprimée de différentes façons équivalentes. La sémantique co-itérative décrit des machines à états [CMPP23]. La sémantique relationnelle vérifie que les flots d’entrée et de sortie satisfont les contraintes du programme [BBD⁺17]. D’un autre côté, la sémantique probabiliste interprète les programmes à l’aide de distributions de probabilités [Koz81, Sta17]. Nous montrons comment combiner ces points de vue pour étendre les sémantiques synchrones au cadre probabiliste et ainsi raisonner sur l’équivalence de programmes. Nous pouvons alors démontrer la correction d’une passe de compilation nécessaire pour utiliser un algorithme d’inférence efficace pour les modèles mélangeant des paramètres d’espace (qui n’évoluent pas dans le temps) et d’états (qui varient).

Remerciements Ce travail a été financé par le projet émergence de la mairie de Paris, ReaLiSe. Merci à Louis Mandel et Thomas Ehrhard pour les nombreuses discussions fructueuses.

Références

- [BBD⁺17] Timothy BOURKE, Léo BRUN, Pierre-Évariste DAGAND, Xavier LEROY, Marc POUZET et Lionel RIEG : A formally verified compiler for lustre. *In PLDI*, 2017.
- [BMA⁺20] Guillaume BAUDART, Louis MANDEL, Eric ATKINSON, Benjamin SHERMAN, Marc POUZET et Michael CARBIN : Reactive probabilistic programming. *In PLDI*, 2020.
- [BP13] Timothy BOURKE et Marc POUZET : Zélus : a synchronous language with ODEs. *In HSCC*, 2013.
- [CMPP23] Jean-Louis COLAÇO, Michael MENDLER, Baptiste PAUGET et Marc POUZET : A constructive state-based semantics and interpreter for a synchronous data-flow language with state machines. *In EMSOFT*, 2023.
- [Est19] ESTEREL TECHNOLOGIES & ANSYS : SCADE Suite. <http://www.esterel-technologies.com/products/scade-suite/>, 2019.
- [Koz81] Dexter KOZEN : Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3) :328–350, 1981.
- [Sta17] Sam STATON : Commutative semantics for probabilistic programming. *In ESOP*, 2017.

Utilisation de Coq pour l'enseignement des mathématiques en L1 et en seconde

Marie Kerjean, Micaela Mayero, Pierre Rousselin

Université Sorbonne Paris Nord, LIPN, LIPN, LAGA

Avec Marie Kerjean et Micaela Mayero, nous avons créé en septembre 2021 un cours récurrent de 18h d'« Initiation aux preuves formelles » avec Coq au premier semestre de L1 double-licence mathématiques et informatique à l'université Sorbonne Paris Nord. Une expérience ponctuelle de 9h a également eu lieu en 2022 avec des élèves de seconde du lycée Joséphine Baker de Pierrefitte-sur-Seine.

Cet exposé est un retour sur ces expériences. Nous présenterons en détail certaines séances de travaux pratiques données en L1, la méthode employée pour l'écriture des sujets, l'évolution de ce module ainsi que les difficultés rencontrées par les enseignant·e-s et les étudiant·e-s. Enfin, nous évoquerons des pistes d'évolution pour l'enseignement des mathématiques avec Coq au niveau L1.

Des temps d'échange et d'expérimentation sont prévus. Les participants qui ne connaissent pas Coq pourront se servir des sujets de travaux pratiques comme d'un tutoriel; les experts de Coq se tourneront d'avantage vers des questions pédagogiques et techniques.

Notre toute première expérience est la création, en septembre 2021 du cours « Initiation aux preuves formelles » pour les étudiant·e-s de L1 double-licence mathématiques et informatique, dès le premier semestre. C'est un moment crucial où les étudiant·e-s doivent s'approprier la rigueur à la fois dans la rédaction de démonstrations en mathématiques et dans l'écriture de programmes en informatique. L'assistant de preuve Coq est choisi car nous avons les compétences localement. Ce module se déroule sur 6 séances de 3h de travaux pratiques et d'une évaluation finale d'1h30, toujours sur machine. La troisième édition de ce module vient de s'achever et nous reviendrons plus tard sur ses évolutions et ses limites actuelles.

L'expérience avec les élèves de seconde a eu lieu dans le cadre de « MathC2+ », stages de mathématiques se déroulant à l'université pendant les vacances scolaires à l'initiative de l'IREM¹ de l'université Sorbonne Paris Nord. Nous avons soumis une dizaine d'élèves volontaires de seconde à 3 séances de 3h de travaux pratiques, fin octobre 2022.

Ces expériences s'inscrivent dans un mouvement plus général de réflexions sur l'usage des assistants de preuve pour l'enseignement des mathématiques. L'article [KLRM⁺22] présente 5 expériences assez différentes d'enseignement en L1 avec des assistants de preuve. Du côté de Coq plus spécifiquement, le « défi LiberAbaci » [lib22] impulsé par l'INRIA « vise à améliorer l'accessibilité du système de preuve interactif Coq pour un public d'étudiant·e-s en mathématiques dans les premières années universitaires ».

1. Institut de recherche sur l'enseignement des mathématiques

Pour le cours « Initiation aux preuves formelles » en L1, nous avons choisi de ne donner que des séances de travaux pratiques, sans cours magistral, pour que les étudiant·e·s soient le plus possible en activité. Le « cours » est donc, à part quelques interruptions des enseignant·e·s, intégralement contenu dans les sujets de travaux pratiques. L'ouvrage [PAC⁺18] a eu une influence déterminante dans la conception de ce cours.

L'ambition première était d'aller jusqu'à des théorèmes de convergences simples sur les suites de nombres réels (dans les faits, peu d'étudiant·e·s sont allés jusque là). Par soucis de simplicité, nous avons fait les choix suivants² :

- travailler avec la bibliothèque standard ;
- travailler avec les tactiques « standard » de Coq ;
- ne travailler que l'écriture de preuves ;
- utiliser une seule logique : `Prop` mais pas `bool` ;
- n'introduire les tactiques automatiques qu'à la fin du module.

Le découpage des sujets de TP a évolué en 3 ans mais en gros suit le plan suivant :

1. Logique (intuitionniste) propositionnelle
2. Entiers naturels et récurrence
3. Calcul des prédicats
4. Nombres réels présentés sous forme axiomatique (voir [May01] et [Sém20])
5. Suites de nombres réels

L'activité « MathC2+ » avec les élèves de seconde est délicate à mettre en place car ces élèves ont peu de références mathématiques facilement formalisables. Nous nous concentrons avec eux sur la logique propositionnelle et proposons des sujets d'ouverture sur les fonctions affines ou le tiers exclu.

L'écriture des sujets demande du temps et de la rigueur. Il s'agit d'abord d'établir (puis de respecter) un *contrat* très fort entre enseignant·e·s et étudiant·e·s : il doit *toujours* être possible d'écrire une preuve avec les informations (tactiques, lemmes) données précédemment. Ensuite l'enseignant·e doit tout faire pour *maîtriser le flux d'informations*. Ceci impose, par exemple, d'introduire les tactiques et les notions nouvelles le plus progressivement possible ; de rédiger des exemples clairs, suivis d'exercices d'application directe, avant de demander, dans un second temps, plus d'initiatives aux étudiants.

Ces principes d'écriture entrent parfois en conflit avec les deux objectifs suivants (eux-mêmes parfois contradictoires). D'abord, c'est un cours de *mathématiques avec Coq* et non un cours de Coq. Dans ce contexte, tout ce qui ne relève pas de l'activité mathématique est une perte de temps³ : nous n'avons pas parlé de propositions inductives, ni de la syntaxe de la commande `Search`. Enfin, nous ne voulions pas non plus tout cacher derrière un écran de fumée. Il est tentant de créer un tout petit monde uniquement pour ce module, mais idéalement, il faudrait que les étudiants volontaires puissent ensuite, dans une certaine mesure, se frotter à Coq « dans la vraie vie ».

Ces expériences font apparaître des difficultés techniques, théoriques et pédagogiques. Coq se met parfois en travers du chemin. On voudrait un monde idéal où tout est prévisible et chaque tactique correspond *grosso modo* à une règle de déduction naturelle ou une étape (même minuscule) de raisonnement ou de réécriture. Mais les étudiant·e·s, c'est humain, raisonnent souvent par essai-erreur ; alors peut advenir de la magie imprévue. On aimerait également que Coq nous aide à respecter le contrat pédagogique. L'interface utilisateur et les bibliothèques utilisées peuvent aussi être questionnées. Un certain nombre de tentatives pédagogiques se sont aussi révélées infructueuses.

La conclusion ne nous appartient pas vraiment. Nous présenterons des travaux et retours d'étudiant·e·s. Nous donnerons nos impressions ainsi que celles d'une enseignante de mathématiques invitée. Il nous semble clair que l'utilisation d'un assistant de preuve favorise

2. Ces choix sont évidemment contestables et feront l'objet de discussions.

3. Bien sûr, le choix de ce qui est intéressant ou non mathématiquement est assez subjectif et dépend du public. Par exemple, est-il pertinent de présenter une définition des entiers naturels ? Est-il formateur pour un mathématicien de connaître les règles de déduction naturelle ?

l'activité des étudiants et agit comme un révélateur des difficultés mathématiques. C'est une expérience pédagogique exigeante (des deux côtés) mais passionnante. Nous espérons donner envie de tenter cette aventure.

Références

- [KLRM⁺22] M. KERJEAN, F. LE ROUX, P. MASSOT, M. MAYERO, Z. MESNIL, S. MODESTE, J. NARBOUX et P. ROUSSELIN : Utilisation des assistants de preuves pour l'enseignement en L1. *Gazette de la Société Mathématique de France*, octobre, 2022.
- [lib22] LiberAbaci : Enseigner les mathématiques à l'aide de Coq, 2022. <https://liberabaci.gitlabpages.inria.fr/>.
- [May01] Micaela MAYERO : *Formalisation et automatisation de preuves en analyses réelle et numérique*. Thèse de doctorat, Université Paris VI, décembre 2001.
- [PAC⁺18] Benjamin C. PIERCE, Arthur AZEVEDO DE AMORIM, Chris CASINGHINO, Marco GABOARDI, Michael GREENBERG, Cătălin HRÎTCU, Vilhelm SJÖBERG et Brent YORGEY : *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, mai 2018. Version 5.5. <http://www.cis.upenn.edu/~bcpierce/sf>.
- [Sém20] Vincent SÉMÉRIA : Nombres réels dans Coq. *Actes des 31es Journées Francophones des Langages Applicatifs (JFLA)*, pages 104–111, 2020.

MetaCoq : de la métaprogrammation à l'extraction certifiée pour Coq

Matthieu Sozeau

Inria Rennes – Bretagne Atlantique, LS2N

MetaCoq est une librairie qui a pour but d'étudier la théorie et l'implémentation de Coq lui-même, en Coq. MetaCoq comprend quatre composants principaux. Le premier, nommé Template-Coq, est une réification en Coq des termes et environnements manipulés par le noyau de Coq (en OCaml), qui permet d'interopérer directement avec le système en développant des métaprogrammes à l'aide du langage Gallina et d'une monade représentant l'interface du noyau de Coq. Le second, PCUIC, est une formalisation d'une variante du système de types du calcul de constructions inductives et de ses propriétés métathéoriques essentielles, qui permet de garantir la correction du système et d'étudier ses extensions possibles. Le troisième composant est une version certifiée d'un vérificateur de types et d'environnements pour ce calcul, garantissant la décidabilité de l'inférence de types et sa correction et complétude vis-à-vis de la spécification de PCUIC. Enfin, le quatrième composant est un développement certifié de la procédure d'effacement des preuves et des types de PCUIC vers un lambda-calcul pur, qui forme le coeur du mécanisme d'extraction de Coq, permettant en particulier d'extraire le vérificateur de types vers des langages exécutables de façon sûre. Au cours de cet exposé, je présenterai un tour d'horizon de ces quatre composants et leur utilisation pour la production de programmes certifiés en Coq avec des garanties de sûreté maximales.

Références

[Tea] The MetaCoq TEAM : The MetaCoq Project. <https://metacoq.github.io/>.

Infer: a compositional and extensible platform for static analysis

Jules Villard

Meta

Infer is an open-source static analysis platform that is used to prevent classes of bugs in code at Meta. Every month, Infer runs on thousands of code changes and detects thousands of bugs that are fixed by developers before they reach production. Over the years, Infer has evolved from a standalone analyser based on Separation Logic to a powerful platform for implementing cross-language interprocedural analyses. Static analysis writers only need to provide an intraprocedural analysis that computes the summary for a single procedure and Infer will transform it into a compositional interprocedural analysis that scales to millions of lines of real code.

In this talk, we'll go through some of the formal methods from program verification that are implemented inside Infer such as separation logic, abstract interpretation, and incorrectness logic, and how they make it a valuable tool for programmers.

References

[Tea] The Infer TEAM : The Infer Project. <https://fbinfer.com>.

STRUCTURES DE DONNÉES

De l'avantage de nuancer les décisions binaires*

Claude Marché¹ et Denis Cousineau²

¹Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, LMF, 91190, Gif-sur-Yvette, France

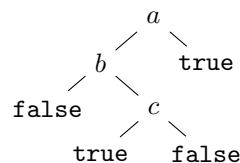
²Mitsubishi Electric R&D Centre Europe, Rennes, France

Nous présentons une extension des diagrammes de décision binaire classiques, consistant à rajouter aux feuilles possibles `true` et `false` des contraintes externes issues d'un domaine de contraintes paramétrable. Nousinstancions ce concept avec des contraintes linéaires sur des variables entières. Nous utilisons cette extension comme domaine abstrait pour inférer des invariants de boucle dans des programmes en WhyML, le langage de l'environnement de vérification Why3. L'application principalement visée est de vérifier des codes en langage Ladder, langage qui sert à programmer des contrôleurs logiques. De tels programmes utilisent typiquement un grand nombre de variables booléennes et simultanément quelques variables entières. Notre approche est évaluée par comparaison avec l'utilisation d'un interpréteur abstrait utilisé précédemment, et nous mettons en évidence les gains significatifs obtenus en terme de temps de calcul ainsi qu'en précision des invariants obtenus.

Mots-clés : Diagrammes de Décision Binaires, Interprétation Abstraite, Inférence d'invariants de boucle.

1 Introduction

Les *diagrammes de décision binaire*, que nous abrégeons en BDD selon la terminologie anglophone, forment une structure de données permettant de représenter efficacement des formules de la logique propositionnelle. Au tout premier abord, un BDD consiste à représenter une formule par un arbre binaire, chaque nœud étant l'objet d'une décision sur la valeur booléenne d'une variable. Ainsi une formule comme $a \vee (b \wedge \neg c)$ peut être vue comme l'arbre



où le sous-arbre à gauche d'une variable exprime le cas où la variable est interprétée à `false`, alors que le sous-arbre de droite concerne le cas elle est interprétée à `true`. Une manière équivalente de voir cet arbre, mais de façon textuelle, est l'expression

```
if a then true else (if b then (if c then false else true)) else false
```

*Ce travail est partiellement financé par un contrat bilatéral ProofInUse-MERCE entre l'équipe Inria Toccata et Mitsubishi Electric R&D Centre Europe, à Rennes.

formée de conditionnelles imbriquées. L'efficacité des BDD réside de manière cruciale au niveau de leur consommation en mémoire : en effet les arbres tels que ci-dessus sont en fait mémorisés comme des graphes acycliques où les sous-arbres identiques sont partagés en mémoire. Si l'on fixe *a priori* un ordre sur les variables, et si l'on fait l'effort de forcer un partage maximal, alors on obtient la propriété très forte que deux formules sont logiquement équivalentes si et seulement si elles sont stockées à la même case mémoire. Une telle propriété permet non seulement de décider très rapidement si une formule est insatisfaisable mais aussi permet de coder les programmes opérant sur des BDD de manière très efficace en temps de calcul, grâce aux techniques de mémoïsation classiques. Plus concrètement, dans une bibliothèque implémentant de tels BDD, comme par exemple la bibliothèque OCaml-BDD [10], l'efficacité en mémoire de la représentation et l'efficacité temporelle des opérations sur les BDD sont obtenues par des codes de nature impérative, à base de tables de hachage et de *hash-consing*. Malgré ce caractère non purement fonctionnel, une telle bibliothèque peut tout à fait être dotée d'une interface d'une nature fonctionnelle pure, cachant le caractère impératif derrière une barrière d'abstraction robuste [7].

Les BDD ont de nombreuses applications, comme la conception de circuits logiques, ou d'une manière générale la résolution de contraintes au sens large du terme. Dans cet article, notre intérêt pour les BDD est justifié par leur utilisation comme domaine abstrait pour l'inférence d'invariants de programme. Dans un contexte d'une application industrielle étudiée par Mitsubishi Electric R&D Centre Europe [5], nous avons cherché à rendre plus efficace la recherche d'invariants de boucle sur des programmes comportant un grand nombre de variables booléennes, ceci grâce aux BDD.

1.1 Contexte : inférence d'invariants par interprétation abstraite

À titre d'exemple, considérons le programme de la figure 1, un exemple jouet en langage C, spécifié par un contrat en ACSL [3]. On peut chercher à vérifier que ce code respecte son contrat en utilisant l'analyseur par interprétation abstraite fourni par Frama-C [2] :

```
> frama-c -eva -lib-entry -main toy toy.c
[...]
[eva:alarm] toy.c:4: Warning:
    function toy: postcondition got status unknown.
[...]
[eva:final-states] Values at end of function toy:
    x ∈ [2018..2059]
    b ∈ {0; 1}
```

On constate que l'état abstrait calculé en fin de fonction, qui sur-approxime l'ensemble des

```
int x,b;

/*@ requires 0 <= x <= 91;
   @ ensures x <= 2024 ;
   */
void toy (void) {
    b = 0;
    while (x <= 2017) {
        b = (x <= 1000);
        if (b) { x += 42; } else { x += 7; }
    }
}
```

Figure 1. Un exemple jouet en langage C, avec un contrat en ACSL

états accessibles, indique que x peut prendre n'importe quelle valeur entre 2018 et 2059, une information qui en particulier ne permet pas de conclure à la validité de la post-condition. Dans le détail, il faut comprendre que l'interpréteur abstrait a inféré un état abstrait point-fixe de la boucle, ici l'état

```
x ∈ [0..2059]
b ∈ {0; 1}
```

l'état final étant ensuite obtenu par intersection avec la négation de la condition de sortie de la boucle, soit $x > 2017$. La faiblesse ici réside donc initialement dans la trop grande sur-approximation obtenue sur l'état point-fixe de la boucle, qui peut être vu comme l'invariant de boucle $0 \leq x \leq 2059 \wedge 0 \leq b \leq 1$.

L'exemple ci-dessus illustre une situation très classique en interprétation abstraite : la précision obtenue dépend de l'expressivité des domaines abstraits utilisés. Dans le cas, présent il y a un besoin d'utiliser un domaine *relationnel*, qui ne se contente pas d'estimer les domaines de chaque variable indépendamment les unes des autres, mais exprime également des relations entre ces variables. Utiliser des domaines relationnels a cependant un coût algorithmique très important (aussi bien en terme d'utilisation mémoire que de temps de calcul) et ne peut être fait sans parcimonie. L'objectif de cet article est de proposer une forme de domaine abstrait relationnel qui sera bien adapté à notre application. À titre d'accroche pour ce qui va suivre, avec la méthode que nous proposons l'invariant de boucle obtenu sur l'exemple jouet est

```
if b then 42 ≤ x ≤ 1042 else 0 ≤ x ≤ 2024
```

ce qui, par intersection avec la négation de la condition de boucle, nous donne $2018 \leq x \leq 2024$, ce qui valide la post-condition.

1.2 Application visée : les programmes Ladder

Dans le cadre d'un projet en collaboration entre Inria et Mitsubishi Electric R&D Centre Europe, nous travaillons depuis quelques années sur la vérification de programmes Ladder. Il s'agit de codes servant à programmer des PLC (Programmable Logic Controllers), utilisés de manière très répandue dans les chaînes de montage automatisées. Nous avons publié des travaux montrant comment nous pouvons vérifier statiquement qu'un code Ladder satisfait une spécification de nature temporelle, exprimée par des diagrammes de transition [5]. Dans ces travaux, un code Ladder (et sa spécification temporelle) est traduit automatiquement en un programme en WhyML, le langage utilisé par l'environnement Why3 [11]. Si la preuve réussit, on a réussi à prouver formellement que le code Ladder respecte sa spécification. En cas d'échec, un outillage permet de remonter un contre-exemple produit par Why3 [4] vers un contre-exemple au niveau du Ladder.

Dans cette approche, le code WhyML généré se caractérise par, d'une part, un grand nombre de variables booléennes, et d'autre part, une structure de contrôle particulière : le code est formé d'une séquence de boucles successives, typiquement une dizaine à la suite. Pour que la preuve réussisse, chacune de ces boucles doit être munie d'un invariant de boucle suffisamment précis. Il n'est pas envisageable, pour l'applicabilité de la méthodologie, de supposer que l'utilisateur va donner les invariants lui-même. Ils doivent donc être générés automatiquement. C'est donc là que l'interprétation abstraite entre en jeu de manière cruciale. Pour atteindre nos objectifs, nous avons donc voulu ajouter à Why3 un moteur de génération d'invariants par interprétation abstraite. Heureusement un tel prototype existait un début de ce travail, réalisé par Lucas Baudin [1]. Nous avons donc repris le développement de ce prototype, que nous avons nommé *InferLoop*. Le gros point manquant au départ était justement le support des variables booléennes. Nous avons alors complété le code de *InferLoop* par un support des booléens dans les domaines abstraits. Ce support était analogue à ce que l'on ferait avec du code C, comme sur la figure 1, où les booléens sont vus

comme des entiers valant 0 ou 1. Ce prototype, après un peu de travail de mise au point, a permis de réaliser une première version d'un vérificateur de programmes Ladder [5]. À titre d'illustration, sur un code WhyML équivalent au code C de la figure 1, l'invariant généré est

$$(\neg b \wedge 0 \leq x \leq 91) \vee (b \wedge 42 \leq x \leq 1042) \vee (\neg b \wedge 1008 \leq x \leq 2024)$$

ce qui s'avère suffisant pour prouver la post-condition. On peut d'ailleurs remarquer la forme de l'invariant généré, une disjonction de conjonctions, qui révèle le fait que InferLoop utilise des domaines *disjonctifs* (cf Section 5).

Les résultats expérimentaux rapportés dans l'article Belo et al. [5] sont positifs dans le sens où un exemple de code Ladder non trivial réussit à être prouvé automatiquement. Le résultat est malgré tout mitigé par un constat d'inefficacité de la génération d'invariants, selon deux aspects. Un premier aspect est que InferLoop ne passe pas suffisamment à l'échelle pour traiter la dizaine de boucles en séquence qui devaient être supportées, ce qui a nécessité une adaptation en amont pour générer un code WhyML distinct pour chaque boucle, l'invariant de chaque boucle devant être réinjecté comme pré-condition de la boucle suivante. Un second aspect est que, malgré le découpage précédent, le temps de calcul des invariants était insatisfaisant : le temps passé à générer les invariants était de l'ordre de dix fois le temps passé à prouver le programme.

Pour résoudre ce problème de passage à l'échelle, nous avons cherché des solutions qui prendraient mieux en compte notre situation spécifique, à savoir des programmes faisant usage d'un grand nombre de variables booléennes. Les solutions potentielles que nous avons trouvées dans la littérature et que nous avons expérimenté (et discutées dans la section 5 ci-après), n'ont pas offert les résultats escomptés, nous nous sommes donc tournés vers une solution originale qui semblaient *a priori* mieux convenir aux besoins. Cette solution, que nous nommons *Diagrammes de décision binaires paramétriques*, consiste à généraliser les BDD classiques, en remplaçant les deux seules feuilles possibles `true` et `false` par des formules d'un domaine paramètre. Ce paramètre peut être instancié par exemple par un domaine dédié aux entiers.

1.3 Contributions et structure de cet article

La structure de cet article est la suivante : dans la section 2, nous rentrons dans le détail des opérations fournies par la bibliothèque OCaml-BDD, et comment nous nous en sommes servi pour un premier prototype d'interpréteur abstrait pour des programmes purement booléens. Nous présentons nos contributions à OCaml-BDD : des opérations supplémentaires nécessaires pour l'interprétation abstraite. Dans la section 3 nous présentons la structure de données de BDD paramétriques. Nous présentons d'abord son interface, et une partie de son implémentation, sous forme d'un foncteur OCaml `BDDparam`. Nous présentons alors l'instance de ce foncteur sur un domaine de contraintes linéaires sur les entiers. Dans la section 4 nous présentons des expérimentations conduites pour évaluer l'efficacité apportée, y compris les améliorations de rapidité et de précision sur les programmes WhyML traduits depuis Ladder. Enfin, dans la section 5, nous dressons quelques conclusions de notre travail, comparons celui-ci avec des travaux existants, et présentons des perspectives.

Le code complet de notre bibliothèque BDDinfer est disponible comme sous-répertoire du dépôt de Why3 [8].

2 Diagrammes de décision binaire

Dans un premier temps, nous nous sommes focalisé sur l'objectif de réaliser un prototype d'interpréteur abstrait générant des invariants de boucles pour des programmes avec uniquement des variables booléennes. L'idée est alors d'utiliser la bibliothèque OCaml-BDD pour implémenter un domaine abstrait de formules booléennes. Un exemple illustratif est donné

```

val random_bool () : bool
val a : ref bool
val b : ref bool
val c : ref bool

let f () =
  a := False; b := False; c := False;
  while not !c do
    if !b then c := True;
    if !a then b := True;
    let x = random_bool () in if x then a := True;
  done;
  assert { !a }

```

Figure 2. Un exemple de programme purement booléen avec une boucle, en WhyML. La fonction `random_bool` est déclarée sans corps, elle renvoie de façon non-déterministe un booléen.

sur la figure 2. Même très simple, un tel exemple est déjà représentatif des codes issus de programmes Ladder, où les changements d'état de devices logiques évoluent en plusieurs étapes d'itération. La fonction `random_bool` utilisé dans cet exemple est représentative des entrées lues depuis des capteurs, dont les valeurs sont non-déterministes.

2.1 Opérations nécessaires pour un interpréteur abstrait

Pour réussir à prouver l'assertion après la boucle dans l'exemple de la figure 2, il est nécessaire de découvrir un invariant suffisamment précis pour la boucle. Un tel invariant est naturellement une formule booléenne sur a , b et c , et l'on se propose de la générer sous la forme d'un BDD. Suivant les principes de base de l'interprétation abstraite, on va construire en chaque point du code un état abstrait (ici donc une formule, c'est-à-dire un BDD) qui sur-approxime les états accessibles à ce point. On va procéder par une exécution symbolique vers l'avant, en calculant l'état abstrait après chaque instruction à partir de l'état avant son exécution. Ainsi, sur l'exemple de la figure 2, au début du corps de `f` on part de l'état `true`, indiquant que toutes les valeurs de a , b et c sont possibles, puis on traverse les trois premières affectations et obtenons successivement les états $\neg a$, $\neg a \wedge \neg b$ et $\neg a \wedge \neg b \wedge \neg c$. Il faut noter donc que nous avons besoin d'opérations sur les BDD qui nous permettent de calculer le BDD représentant l'état du programme après une affectation. L'interprétation abstraite va ensuite rentrer dans le corps de la boucle, et interpréter la première conditionnelle. Pour interpréter une conditionnelle, il faut :

- intersecter l'état d'entrée avec la condition, interpréter la branche `then`
- intersecter l'état d'entrée avec la négation de la condition, interpréter la branche `else`
- calculer l'union des états obtenus, représentant donc l'ensemble des états accessibles après la conditionnelle

On voit clairement apparaître ici l'utilisation des opérations de conjonction, de négation et de disjonction, qui existent naturellement sur les BDD. L'interprétation de l'introduction d'une variable locale (`let..in..`) et d'un appel de fonction va demander des opérations moins naturellement présentes sur les BDD. L'ajout de la variable locale n'est pas difficile si on la voit comme la prise en compte d'une variable booléenne supplémentaire. L'interprétation d'un appel d'une fonction arbitraire est plus problématique. Il y a déjà au départ deux possibilités : exécuter le corps de la fonction, ou bien interpréter en une seule étape cet appel, en prenant en compte uniquement le contrat de la fonction. Notre interpréteur sait faire les deux, mais pour notre propos ici nous ne considérons que ce second cas. Ce second

cas couvre d'ailleurs le cas des affectations, les deux pouvant être vus comme des instances d'une opération générique que l'on appellera **Havoc**. Voici sa forme générale :

$$\text{Havoc}(x_1, \dots, x_k; C)$$

est une opération qui modifie en place les variables x_1, \dots, x_k de manière *a priori* non déterministe, mais en respectant la post-condition C . La condition C peut mentionner à la fois les valeurs des variables avant et après le **Havoc**. Les valeurs avant seront ici notées $\text{old}(x_i)$. Ainsi, une affectation $x \leftarrow e$ correspond à

$$\text{Havoc}(x; x = \text{old}(e))$$

où $\text{old}(e)$ désigne l'expression e où toutes les occurrences de x sont remplacées par $\text{old}(x)$. De même, un appel de fonction sans corps, de la forme

```
val f (...)
  requires P
  writes x1, ..., xk
  ensures Q
```

est vu comme

$$\text{assert } P; \text{Havoc}(x_1, \dots, x_k; Q)$$

Traiter un **Havoc** de façon générale dans le moteur d'interprétation est un peu plus délicat : il y a besoin de renommage, puis d'élimination des variables temporaires introduites. Nous détaillons cela ci-dessous. Enfin, pour traiter la boucle, il faut savoir calculer des *post-point-fixes*. En général, on s'appuie sur une opération spéciale du domaine : le *widening*. Avec les booléens, il se trouve que la disjonction (c.-à-d. l'union) est une opération de *widening* convenable car il ne peut y avoir de séquence indéfiniment croissante.

2.2 La bibliothèque OCaml-BDD et nos extensions

La figure 3 présente un extrait de l'interface de la bibliothèque OCaml-BDD, montrant les opérations essentielles à connaître : les constructeurs de BDD et la fonction de test de satisfaisabilité. Les variables sont codées par des entiers. En réalité, cette interface est la signature d'un foncteur qui doit être utilisé en lui donnant une valeur `max_var` qui est le numéro maximal de variable. Cette interface est purement fonctionnelle, mais son implémentation utilise du *hash-consing* pour garantir un partage maximal, ce qui permet à la fonction `is_sat` d'être implémentée par un simple test d'égalité à `zero`.

```
type variable = int
  (** A variable is an integer, ranging from 1 to [max_var] *)
type t (** The abstract type of BDDs *)

(** smart constructors *)
val zero : t (* i.e. false *)
val one : t (* i.e. true *)
val mk_var : variable -> t
val mk_not : t -> t
val mk_and : t -> t -> t
val mk_or : t -> t -> t

val is_sat : t -> bool
  (** Checks if a bdd is satisfiable *)
```

Figure 3. Interface de la bibliothèque OCaml-BDD (extrait).

Comme vu dans la section 2.1, les opérations de négation, conjonction et disjonction nous permettent de réaliser la gestion des conditionnelles et des boucles d'un programme à interpréter, mais il manque de quoi gérer l'opération `Havoc`. Nous réalisons le transfert d'un état abstrait S à travers une opération

$$\text{Havoc}(x_1, \dots, x_k; C)$$

en la décomposant en plusieurs étapes :

1. considérer des variables x'_1, \dots, x'_k n'apparaissant pas déjà dans S , visant à dénoter les nouvelles valeurs des variables x_i ;
2. calculer la conjonction S' de S et C' , où C' est obtenue en remplaçant dans C les occurrences de x_i par x'_i et les occurrences de `old(x_i)` par x_i ;
3. éliminer de S' toutes les anciennes occurrences de x_1, \dots, x_k , élimination vue comme une *élimination de quantificateur existentiel* ;
4. renommer les variables x'_1, \dots, x'_k en leur version sans prime.

Expliquons cette idée d'élimination existentielle sur un exemple : imaginons un état abstrait $S = a \wedge \neg b$ et l'opération d'échange

$$\text{Havoc}(a, b; a = \text{old}(b) \wedge b = \text{old}(a))$$

on introduit donc l'état (\leftrightarrow dénote l'équivalence)

$$S' = (a \wedge \neg b) \wedge (a' \leftrightarrow b \wedge b' \leftrightarrow a)$$

on suppose alors que la bibliothèque de BDD fournit une opération qui permet de construire le BDD pour la formule

$$\exists a, b. S'$$

Si l'implémentation est au point, on doit obtenir une formule équivalente à $\neg a' \wedge b'$ qui après renommage donnera l'état $\neg a \wedge b$. Ce mécanisme fonctionne donc si cette opération d'élimination existentielle est fournie, ce qui n'était pas le cas dans OCaml-BDD au départ. Il s'agit donc d'une contribution que nous avons faite à cette bibliothèque. La fonction fournie a le type `(variable -> bool) -> t -> t`, la fonction en argument indiquant si une variable doit être éliminée. La figure 4 présente le code du *smart constructor* en question, pour les lecteurs et lectrices qui seraient intéressées par la façon d'implémenter une telle fonction de manière efficace, en profitant du partage.

Comme vu ci-dessus, il nous faut une opération de renommage, qui n'est pas évidente sur les BDD, dont l'ordre des variables est imposé. Nous reviendrons sur ce sujet à la section suivante, pour le moment nous pouvons considérer qu'un renommage disons de x en une variable fraîche y peut-être effectué par une conjonction avec $x \leftrightarrow y$ puis l'élimination existentielle de x .

Tous les éléments sont maintenant disponibles pour écrire un interpréteur abstrait de base, selon les principes décrits à la section 2.1. À titre d'illustration, un tel interpréteur permet de générer, sur l'exemple de la figure 2, l'invariant de boucle :

```
if a then if b then true else if c then false else true
      else if b then if c then false else true else true
```

Le moins que l'on puisse dire est que ce n'est pas très lisible. Ainsi, un tout dernier élément que nous avons proposé comme contribution à OCaml-BDD est la reconstruction d'une formule à partir d'un BDD, reconstruction qui essaye de reconstituer des connecteurs booléens quand c'est possible. Avec cette amélioration cosmétique, l'invariant généré est

```
if a then (b ∨ ¬c) else (¬b ∧ ¬c)
```

On peut d'ailleurs calculer la conjonction avec la négation de la condition de boucle, pour obtenir l'état de sortie de boucle $a \wedge b \wedge c$ qui en particulier permet de garantir que l'assertion finale du code de l'exemple de la figure 2 est vraie.


```

let rec quantifier_elim cache op filter b =
  try H1.find cache b with Not_found ->
    let res = match b.node with
      | Zero | One -> b
      | Node(v,l,h) ->
        let low = quantifier_elim cache op filter l in
        let high = quantifier_elim cache op filter h in
        if filter v then op low high else mk v low high
    in H1.add cache b res; res

let mk_exist filter b =
  let cache = H1.create cache_default_size in
  quantifier_elim cache mk_or filter b

```

Figure 4. Le code du constructeur d'élimination existentielle. Un cache est créé à chaque appel, pour ne calculer qu'une seule fois sur les nœuds du DAG. La descente récursive, quand elle rencontre une variable à éliminer, effectue une disjonction des sous-arbres. Le constructeur interne `mk` permet d'effectuer un *hash-consing* et également simplifie quand les deux sous-arbres sont identiques.

3 Diagrammes de décision binaire paramétriques

Dans la section 2, nous avons montré comment la bibliothèque OCaml-BDD, augmentée de quelques ajouts, permet de réaliser un interpréteur abstrait précis et efficace sur des programmes ayant uniquement des variables booléennes. Nous nous intéressons maintenant au cas des programmes ayant des variables non booléennes, en particulier des variables entières. On peut imaginer de construire un domaine abstrait « produit cartésien », qui traiterait en parallèle les booléens et les entiers, mais alors on n'aurait aucune possibilité d'exprimer des relations entre eux, comme dans l'exemple de la figure 1. En fait, un tel domaine produit est facile à faire dans un premier temps, et sans surprise, sur l'exemple de la figure 1, l'invariant généré est $0 \leq x \leq 2058$, exactement celui trouvé par Frama-C.

Dans la mesure où les exemples d'application que l'on vise comportent malgré tout en grande majorité des variables booléennes, nous avons cherché à continuer à se baser sur les BDD. L'idée que nous proposons maintenant est de « nuancer » les arbres de décision binaires, en étendant les valeurs possibles aux feuilles de ces arbres : au lieu de se limiter aux feuilles `true` et `false`, nous proposons de mettre aux feuilles des BDD des contraintes issues d'un autre domaine, par exemple un domaine de contraintes linéaires sur les entiers.

Il s'avère impossible de continuer à utiliser la bibliothèque OCaml-BDD directement, puisque nous souhaitons modifier l'implémentation au cœur de la structure. Pour rester néanmoins le plus générique possible, nous avons développé une version fonctorisée de la bibliothèque, où l'argument du foncteur donne les éléments du domaine sous-jacent nécessaire. Nous avons appelé cela des *BDD paramétrés*. La bibliothèque `BDDparam` construite est décrite dans la section 3.1 qui suit, et est ensuite instanciée dans la section 3.2. Le code est disponible comme sous-répertoire du dépôt de Why3 [8].

3.1 Merci foncteur

La difficulté qui se présente pour réaliser notre foncteur est que nous voulons conserver la propriété essentielle des BDD, à savoir un partage maximal des sous-arbres. Il nous faut supposer que le domaine paramètre soit capable de proposer une fonction d'égalité et une fonction de hachage sur son langage de contraintes qui lui permette d'identifier efficacement les contraintes identiques. C'est une hypothèse que nous faisons. Mais une

difficulté supplémentaire va se rajouter, qui s'explique quand nous voulons instancier notre foncteur avec la bibliothèque `Apron` [14] : une contrainte linéaire sur des variables y est associée à un *environnement*, qui en interne associe des noms de variables à des numéros, correspondant à des indices dans les matrices qui servent à calculer efficacement sur les polyèdres. L'égalité de deux contraintes de ce langage n'a de sens que pour deux contraintes sur le même environnement.

Nous avons alors cherché à minimiser la taille de la signature du domaine paramètre, tout en permettant de supporter ces caractéristiques complexes. Le résultat est présenté sur la figure 5. Le type `p_index`, qui est explicitement un entier, est utilisé comme identifiant unique d'une contrainte, dans un contexte donné. Le type `p_context` de ces contextes est abstrait dans cette signature. Enfin, le type des contraintes est abstrait également, désigné sous le nom de `p_state`. Nous utilisons en effet un vocabulaire proche de l'application à l'interprétation abstraite, en particulier `meet` désigne la conjonction et `join` la disjonction. Nous déléguons la tâche de *hash-consing* les états à chaque instance : en effet, l'idée est que la table de hash-consing qu'il est nécessaire de maintenir soit une partie intégrante du

```

module type ParamDomain = sig

  type p_index = int
  type p_context
  type p_state

  val meet : p_context -> p_index -> p_index -> p_index option
  val join : p_context -> p_index -> p_index -> p_index option
  val widening : p_context -> p_index -> p_index -> p_index option
  (** when result is [None], it means [bottom] for [meet] and [top]
      for [join] and [widening] *)

  val exist_elim : p_context -> p_index -> p_context -> p_index option
  (** [exist_elim ctxi i ctx] fetches the formula [i] in context
      [ctxi], builds a corresponding formula in context [ctx] and
      returns an index for the result. When result is [None], it means
      [top]. The target context [ctx] is supposed to be a sub-context
      of [ctxi], so that this operation corresponds to existential
      elimination of variables of [ctxi] that are not in [ctx]. *)

  val entails : p_context -> p_index -> p_index -> bool
  (** returns [true] when the first formula entails the second *)

  val change : (p_state -> p_state) ->
    p_context -> p_index -> p_context -> p_index
  (** [change f ctx_src i ctx_dst] returns an index in context
      [ctx_dst] for [f s] where [s] is the state of index [i] in
      [ctx_src] *)

  val meet_with_constraint : (p_state -> p_state) ->
    p_context -> p_index option -> p_index option
  (** [meet_with_constraint f ctx i] returns an index in context
      [ctx] for [f s] where [s] is the state of index [i] in
      the same context. The case [None] here means [bottom] *)

end

```

Figure 5. La signature du module attendu comme argument du foncteur `BDDparam`.

contexte. Ainsi la fonction `meet` reçoit deux indices de contraintes dans le *même* contexte, et renvoie l'indice de la conjonction, ou bien `None` si la contrainte résultante est `false`. Des fonctions similaires permettent de calculer les `join` (disjonction) et `widening`, mais également l'élimination des quantificateurs existentiels, comme identifiée comme nécessaire dans la section précédente. Comme l'élimination des variables quantifiées existentiellement change les variables de la contrainte, le contexte du résultat est un autre contexte. La fonction `entails` permet décider si une contrainte est conséquence logique d'une autre. La fonction `change` est une fonction générique de mapping de formule à formule, avec contextes différents en entrée et sortie. Elle doit permettre (entre autres) un renommage de variables. Enfin, la fonction `meet_with_constraint` est similaire à `change` mais s'applique cette fois sur le même contexte en entrée et en sortie, et supporte le cas où les formules sont `false` en particulier en sortie.

```

1 type t = { tag: int; node : view }
2 and view = Zero | One | Leaf of int | Node of variable * bdd * bdd
3
4 let mk_param (i : int) : t =
5   try Hint.find param_table i with Not_found ->
6     let t = gentag () in
7     let n = { tag = t; node = Leaf i } in
8     Hint.add param_table i n; n
9
10 let mk v low high =
11   if low == high then low else hashcons_node v low high
12
13 let meet ctx b1 b2 =
14   let cache = H2.create cache_default_size in
15   let rec app ((u1,u2) as u12) = if u1 == u2 then u1 else
16     match u1.node, u2.node with
17     | Zero, _ | _, Zero -> zero
18     | One, _ -> u2 | _, One -> u1
19     | _ -> try H2.find cache u12 with Not_found ->
20       let res = match u1.node, u2.node with
21         | Node(v1,l1,h1), Node(v2,l2,h2) ->
22           if v1 == v2 then
23             mk v1 (app (l1, l2)) (app (h1, h2))
24           else if v1 < v2 then
25             mk v1 (app (l1, u2)) (app (h1, u2))
26           else (* v1 > v2 *)
27             mk v2 (app (u1, l2)) (app (u1, h2))
28         | Leaf i, Leaf j ->
29           begin match D.meet ctx i j with
30             | Some k -> mk_param k
31             | None -> zero
32           end
33         | Node(v1,l1,h1), Leaf _ ->
34           mk v1 (app (l1, u2)) (app (h1, u2))
35         | Leaf _, Node(v2,l2,h2) ->
36           mk v2 (app (u1, l2)) (app (u1, h2))
37         | (Zero|One), _ | _, (Zero|One) -> assert false
38       in
39     H2.add cache u12 res; res
40   in
41   app (b1, b2)

```

Figure 6. Extrait du code du foncteur `BDDparam`. L'opération `meet` utilise un cache sur des paires de BDD pour mémoriser les appels identiques. Aux lignes 28–32, on fait appel à l'opération `meet` du domaine paramètre `D` pour faire une conjonction de deux feuilles. Aux lignes 33–36 on repousse récursivement les conjonctions d'une feuille avec un nœud interne de BDD.

Notre foncteur prend un module de la signature précédente en argument. La signature de sortie de ce foncteur est très similaire à l'extrait de la figure 3, et en particulier elle reste purement fonctionnelle. Un ajout est que pour implémenter une version plus efficace du renommage de variable que la méthode proposée brièvement à la fin de la section 2.1, nous ajoutons explicitement une telle fonction de renommage.

Un extrait de l'implémentation de notre foncteur `BDDparam` est donné sur la figure 6, montrant le type de donné interne d'arbre, les constructeurs `mk_param` et `mk` pour respectivement les contraintes paramètres aux feuilles des BDD et leurs nœuds internes, et enfin la fonction

```

1 module ApronDom = struct
2
3   type p_index = int
4   type p_state = Polka.strict Polka.t Abstract1.t
5   type p_context = {
6     apron_env : Environment.t;
7     mutable counter : int;
8     state_to_int : int H.t;
9     int_to_state : p_state Hint.t;
10  }
11
12  let get ctx i =
13    try Hint.find ctx.int_to_state i with Not_found ->
14      Format.eprintf "get: %d not found in %s." i; assert false
15
16  let record ctx s =
17    try H.find ctx.state_to_int s with Not_found ->
18      let i = ctx.counter in
19        ctx.counter <- ctx.counter + 1;
20        Hint.add ctx.int_to_state i s;
21        H.add ctx.state_to_int s i; i
22
23  let meet ctx i j =
24    let si = get ctx i in let sj = get ctx j in
25      let s = Abstract1.meet man si sj in
26      if Abstract1.is_bottom man s then None else Some (record ctx s)
27
28  let exist_elim ctxi i ctx =
29    let si = get ctxi i in
30    let s = Abstract1.change_environment man si ctx.apron_env false in
31    if Abstract1.is_top man s then None else Some (record ctx s)
32
33  let entails ctx i j =
34    let si = get ctx i in let sj = get ctx j in Abstract1.is_leq man si sj
35
36  let change f ctxi i ctx =
37    let si = get ctxi i in let s = f si in record ctx s
38
39  let meet_with_constraint f ctxi i =
40    let s = match i with
41      | Some i -> let si = get ctxi i in f si
42      | None -> f (Abstract1.top man ctxi.apron_env)
43    in if Abstract1.is_bottom man s then None else Some (record ctxi s)
44
45  end

```

Figure 7. L'instance de la signature de la figure 5 avec le support des contraintes linéaires fournies par la bibliothèque `Apron`. Le contexte contient un environnement `Apron`, et deux tables de hachage réciproques l'une de l'autre. La fonction `get` permet de récupérer une formule à partir de son indice, et la fonction `record` enregistre une formule dans ce contexte, en lui attribuant un nouvel indice si elle n'y est pas déjà.

`meet`, qui fait appel à l'opération `meet` du domaine paramètre au niveau des feuilles. Le module `Hint` fournit des tables de hachage indexées par des entiers.

3.2 Une instance : un domaine abstrait mixte entiers-booléens

Notre instance avec des contraintes linéaires sur les entiers utilise la bibliothèque `Apron` [14]. Le module à passer en argument du foncteur est donné sur la figure 7. Le module `H` fournit des tables de hachage indexées par des formules `Apron`, pour lesquelles des fonctions de hachage et de comparaison sont disponibles.

4 Évaluations expérimentales

Nos évaluations se basent sur une comparaison avec le prototype `InferLoop` précédent de `Why3`. Comme annoncé en accroche d'introduction, `BDDinfer` fait aussi bien que `InferLoop` sur le code de la figure 1, en générant un invariant permettant de prouver la post-condition. Nous montrons ci-dessus un premier exemple qui montre que `BDDinfer` peut gagner significativement en précision. Puis, dans un second temps, nous comparons avec `InferLoop` en terme de temps de calcul, cette fois sur du code `WhyML` généré automatiquement à partir de `Ladder`.

4.1 Un exemple de gain de précision

Le code de la figure 8 est un exemple jouet, construit par inspiration et réduction d'exemples réels de modélisation de codes `Ladder`. Avec `InferLoop`, l'invariant généré est

$$\neg s_1 \vee (0 \leq c \leq 42 \wedge s_1 \wedge \neg serr)$$

L'invariant est insuffisamment précis pour que l'assertion soit prouvable. Avec `BDDinfer`, l'invariant généré est

`if serr then ($\neg s_1 \wedge \neg s_0 \wedge 43 \leq c$) else (if s_1 then $\neg s_0 \wedge 0 \leq c \leq 42$ else $s_0 \wedge c = 0$)`

on voit que l'invariant en forme de conditionnelle capture précisément les différents phases d'itération. L'assertion est prouvable. Noter que l'on peut produire une variation de cet

```

let f ()
  requires { s0 = True}
  requires { s1 = serr = False}
  requires { c = 0 }
  =
  while (not !serr) do
    if !s1 then c := !c +1;
    let tmp = random_bool () in
    if !s0 && tmp then
      (s0 := False; s1 := True; c := 0)
    else
      if (!s1 && (!c > 42)) then
        (s1 := False; serr := True)
  done;
  assert { !serr ^ !c ≥ 43}

```

Figure 8. Un exemple illustrant l'importance de la précision des invariants.

exemple en insérant un `break` après `serr := True` (et du coup mettre la condition de boucle à `True`). L'invariant généré est alors

$$(\text{if } s_1 \text{ then } \neg s_0 \wedge 0 \leq c \leq 42 \text{ else } s_0 \wedge c = 0) \wedge \neg serr$$

L'assertion reste prouvable, et dans les faits on peut même se passer complètement de la variable `serr`.

4.2 Évaluation sur du code Ladder

Belo et al. [5] présentent une façon de vérifier qu'un programme Ladder satisfait une spécification donnée sous la forme d'un diagramme de temps. Un tel diagramme de temps représente un scénario temporel de changement de valeur des variables d'entrées du programme, assorti de la spécification des valeurs des variables de sortie associées. Cette vérification consiste en la traduction automatique d'un code Ladder et de sa spécification temporelle en un programme WhyML. La faisabilité de l'approche est démontrée en se basant sur le cas d'usage d'un programme de contrôle d'un chariot (figure 9). Son code Ladder est donné figure 10 est sa traduction automatique est détaillée dans la figure 11. Nous renvoyons à Belo et al. [5] pour les explications concernant la sémantique du langage Ladder, les détails d'implémentation en WhyML des instructions `RST`, `SET`, `PLS`, ainsi que celles dédiées à la gestion des minuteurs.

Le comportement attendu du chariot, et donc de son code Ladder, est spécifié par un diagramme de temps, donné dans la figure 12. Un tel diagramme de temps spécifie des changements séquentiels des valeurs d'entrée du programme (et spécifie les valeurs de sortie attendues). Nous appelons ces changements de valeurs d'entrée des *événements*. Entre deux événements, les valeurs de sortie sont spécifiées stables. Nous appelons *états* les ensembles d'exécutions du programme entre deux événements consécutifs. Le nombre d'exécutions du programme durant un état est arbitrairement grand.

Pour modéliser l'exécution du programme Ladder en regard du diagramme de temps spécifié, Belo et al. proposent l'encodage suivant. Si E_i est un événement du diagramme de temps, et S_i est son état associé. Soit X_i la variable d'entrée dont la valeur change à l'événement E_i (sur l'exemple du chariot et son diagramme de la figure 12, X_1 est X_0). Soit G_i la conjonction d'assertions concernant les valeurs d'entrée du programme pour l'événement E_i et l'état S_i (sur l'exemple du chariot, G_1 est $X_0 \wedge \neg X_1 \wedge \neg X_2 \wedge X_3 \wedge X_4$). On

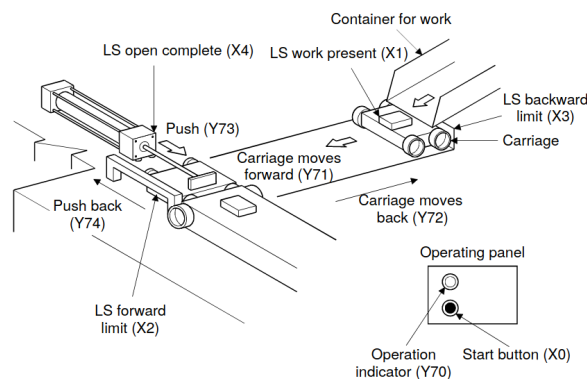


Figure 9. Carriage line control : le chariot transporte un objet le long de sa ligne, puis revient à son point initial, après qu'un bras mécanique ait enlevé l'objet du chariot.

note I_i la conjonction d'assertions concernant les valeurs de sorties pour l'évènement E_i et l'état S_i (sur l'exemple du chariot, I_1 est la formule $Y70 \wedge \neg Y71 \wedge \neg Y72 \wedge \neg Y73 \wedge \neg Y74$).

La modélisation en WhyML d'un évènement E_i et de l'état associé S_i se fait sous la forme d'une boucle de style *do-while*. Les boucles *do-while* ne sont pas présentes dans la syntaxe de WhyML, nous utilisons donc deux fois le corps du programme Ladder traduit en WhyML, une première fois pour modéliser l'exécution (unique) correspondant à E_i , et une deuxième fois dans le corps d'une boucle *while* attenante qui simule les exécutions (dont le nombre est arbitraire, possiblement zéro) de l'état associé S_i . La condition de garde de la boucle correspond à l'hypothèse G_i sur les valeurs des variables d'entrée pour E_i et S_i . La spécification concernant les valeurs des variables de sortie est donnée sous la forme

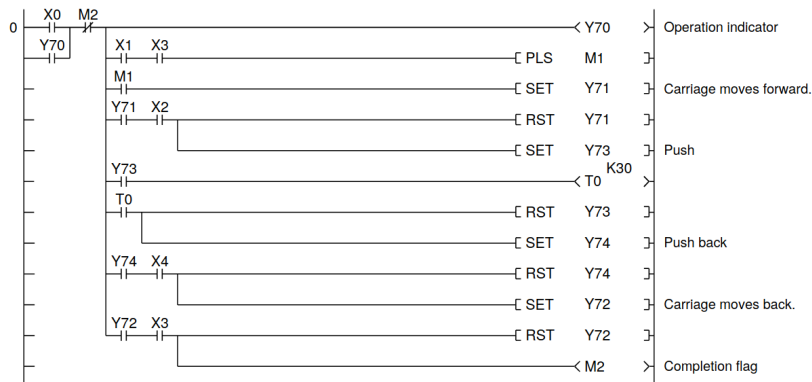
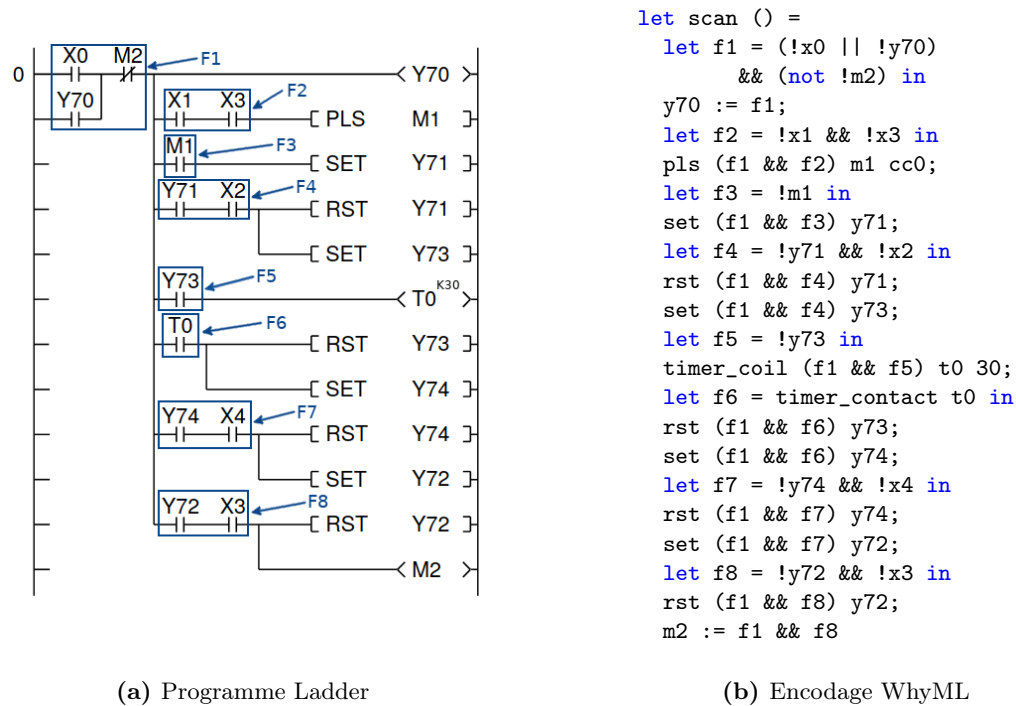


Figure 10. Carriage line control : programme Ladder



(a) Programme Ladder

(b) Encodage WhyML

Figure 11. Encodage d'une exécution du programme Ladder

de l'invariant I_i à prouver : l'initialisation de l'invariant correspond à la spécification de l'évènement tandis que sa préservation correspond à celle de l'état. Enfin, nous modélisons l'aspect non-déterministe du changement d'état par une affectation de la variable d'état X_{i+1} dont le changement déclenche l'évènement suivant E_{i+1} , à l'aide de la fonction `random_bool`.

Nous résumons cette formalisation par le pseudo-code

```
(* evenement E_i *)
scan();
X_{i+1} := random_bool();

(* etat S_i *)
while G_i do
  invariant { I_i }
  scan();
  X_{i+1} := random_bool();
done
```

dans lequel `scan()` représente l'exécution du code WhyML donnée en figure 11b.

Notre objectif initial était de directement encoder la succession de tous les évènements et états dans une même fonction WhyML afin de lancer la vérification grâce au calcul de plus faible pré-condition de Why3 et l'appel à des prouveurs SMT. Malheureusement les performances de l'interpréteur abstrait `InferLoop` de Why3 ne le permettaient pas, son temps de calcul semblant exponentiel en le nombre de boucles successives traitées (voir ci-dessous). Nous avons alors dû implémenter un mécanisme permettant de traiter chaque paire d'évènement et état dans une fonction séparée, en ré-injectant l'invariant généré pour l'état S_i comme pré-condition pour la formalisation de l'évènement E_{i+1} et de l'état S_{i+1} . Cela nous a permis de contourner ce problème de performances, en atteignant la vérification complète du cas d'usage présenté en un temps acceptable de l'ordre de 25 secondes (dont plus de 17 pour la génération d'invariants).

Nous avons donc comparé les performances du nouvel interpréteur abstrait `BDDinfer` avec celles de l'ancien `InferLoop`. Dans ce but, nous chronométrons la génération d'invariants sur différents fichiers WhyML. Les résultats sont présentés sur la figure 13. L'utilisation de `BDD` paramétrés dans le nouveau moteur d'interprétation abstraite `BDDinfer` de Why3 permet d'obtenir de bien meilleures performances. Cette amélioration des performances nous permet ainsi de ne plus avoir à créer des fonctions WhyML distinctes pour les différents évènements et états, et surtout de ne plus compliquer l'outillage nécessaire pour propager les invariants

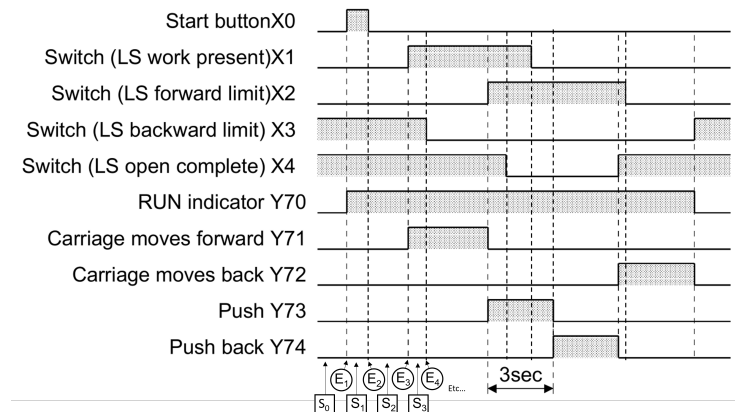


Figure 12. Diagramme de temps

év./état	1	2	3	4	5	6	7	8	9	10
InferLoop	1.02	0.95	2.05	0.97	5.45	1.49	1.21	1.44	1.29	0.86
BDDinfer	0.12	0.10	0.12	0.10	0.33	0.10	0.09	0.13	0.12	0.08
év./états	1	1→2	1→3	1→4	1→5	1→6	1→7	1→8	1→9	1→10
InferLoop	1.02	33.5	292	1089	-	-	-	-	-	-
BDDinfer	0.12	0.19	0.30	0.42	0.53	0.91	1.00	1.10	1.22	1.35
# vars	47	95	143	191	239	287	335	379	427	475

Figure 13. Comparaison des performances de l’ancien générateur d’invariant InferLoop de Why3 et du nouveau générateur BDDinfer basé sur les BDD paramétrés. La première partie de la table compare les performances, mesurées en secondes de temps CPU, pour une unique paire événement/état. Pour chacun de ces tests, qui ne comportent qu’une seule boucle, le temps de calcul est de manière systématique de l’ordre de 10 à 20 fois plus rapide avec BDDinfer. La seconde partie compare les performances pour des séries d’événement/état du point de départ jusqu’à un état E_i donné pour i de plus en plus grand. Chaque test contient donc une boucle de plus que le précédent. Avec InferLoop, l’explosion combinatoire est spectaculaire, il est même impossible d’atteindre le quatrième événement dans un temps raisonnable. Alors qu’avec BDDinfer, le temps de calcul augmente plus ou moins linéairement avec le nombre de boucles, et reste très raisonnable, moins de deux secondes pour la série complète. La toute dernière ligne donne le nombre de variables booléennes utilisées dans les BDD sous-jacents. On peut constater que la bibliothèque BDDparam tient bien le choc avec un nombre de variables allant jusqu’à environ 500.

générés pour un état E_i vers l’évènement suivant. Au final, nous sommes capables de traiter le cas d’usage complet en moins de 2 secondes, contre 25 précédemment.

Cette amélioration des performances nous permet également de simplifier notre implémentation de la génération de scénarios d’erreur quand un diagramme de temps n’est pas vérifié. En effet, quand l’encodage WhyML d’une paire d’état et d’évènement n’était pas prouvé par Why3, nous reconstruisions alors un scénario d’erreur Ladder composé de deux types d’éléments distincts. En premier lieu, nous collectons les domaines générés par l’interpréteur abstrait qui sont retournés comme valeurs possibles des variables internes du programme lors de l’exécution des états. En un second temps, on collecte des valeurs concrètes pour l’exécution des évènements, provenant du contre-exemple fourni par le prouveur SMT, lors de la tentative de preuve de la concaténation des modélisations WhyML des états et évènements successifs depuis le premier jusqu’à celui incriminé (en ré-injectant à nouveau les invariants générés pour tous les états concernés). Nous pouvons maintenant générer de tels scénarios d’erreur directement sans avoir à générer les invariants des états un à un, pour les ré-injecter *a posteriori*.

De plus la précision accrue du nouvel interpréteur abstrait BDDinfer a permis de construire de meilleurs scénarios d’erreur en cas de diagramme de temps non vérifié. En effet, pour modéliser la spécification concernant les minuteurs, nous introduisons une variable fraîche de comptage des exécutions pour chaque minuteur spécifié, et tentons d’obtenir l’identification de cette variable fraîche avec celle correspondant au minuteur grâce aux domaines relationnels de l’interpréteur abstrait (voir [5] pour plus de détail). Malheureusement le domaine généré n’était pas toujours suffisamment précis et il arrivait qu’il ne soit pas plus précis que $[0, +\infty[$ (typiquement lorsque l’on modifiait le programme pour que le nombre d’exécutions entre les évènements 5 et 8 ne corresponde plus au minuteur spécifié dans le diagramme de temps). Le nouvel interpréteur abstrait paraît bien plus précis et nous n’avons pas pu le mettre en

défaut sur les exemples que nous avons.

En conclusion, les améliorations apportées par ce nouvel interpréteur abstrait utilisant les BDD paramétrés nous ont permis de simplifier grandement notre façon de modéliser un programme Ladder et son diagramme de temps associé. L'amélioration des performances permet également de nous rassurer sur le passage à l'échelle de notre démarche de vérification à des programmes Ladder et diagrammes de temps plus conséquents. Enfin, l'amélioration de la précision des invariants générés et des domaines associés, a permis d'améliorer nettement les scénarios d'erreur quand une violation du diagramme de temps est détectée. De plus, cette amélioration de précision permet également d'entrevoir la possibilité d'appliquer cette méthodologie de vérification à des formes de spécifications temporelles plus complexes : passer d'un scénario représenté par un diagramme de temps linéaire, à un ensemble de scénarios décrits par une certaine forme d'automate.

5 Conclusions

Nous avons présenté un interpréteur abstrait basé sur une version originale de BDD, paramétrés par un domaine. Cette version se présente sous forme d'un foncteur OCaml. Nous avons montré comment instancier ce foncteur sur un domaine de contraintes linéaires sur les entiers. Nous avons évalué notre interpréteur sur des exemples, issus d'un contexte d'utilisation industrielle pour l'analyse statique de programmes Ladder pour les PLC. Ces exemples se caractérisent par un grand nombre de variables booléennes, mais aussi des variables entières. Nos évaluations montrent que nos BDD paramétrés sont particulièrement bien adaptés à ce type de programmes, aussi bien en terme de précision des invariants générés que de temps de calcul.

Une question que nous n'avons pas abordée est celle de la correction de l'approche : sommes-nous sûr que les invariants générés sont bien des invariants ? Nous n'avons pas de garantie forte, puisque nous dépendons du code OCaml de la bibliothèque `BDDparam` et de celui de l'interpréteur abstrait `BDDinfer` lui-même. Néanmoins notre implémentation fournit un garde-fou très sûr : nous demandons à `Why3` de reprouver que chaque invariant généré est initialement vrai et préservé par une itération. De cette façon nous avons une très forte garantie de correction de ces invariants. Cette méthode a d'ailleurs pu quelques fois détecter un bug dans notre implémentation !

5.1 Travaux connexes

Nous n'avons pas été facilement en mesure de comparer notre approche avec des approches existantes en interprétation abstraite. D'une part au niveau des outils eux-mêmes avec lesquels expérimenter, nous n'avons pas eu un grand succès, en particulier parce que nous recherchions des bibliothèques en OCaml. Nous avons tenté d'utiliser la bibliothèque `MOPSA` [15], mais l'API fournie est de trop haut niveau, elle ne permet pas de récupérer des invariants qui seraient générés au niveau de chaque boucle. Nous avons tenté d'utiliser la bibliothèque `BDDApron` [13] dont le nom est prometteur, mais nous avons dû renoncer par manque de compréhension de son interface et de son API, et qui plus est cette bibliothèque ne semble pas maintenue, et n'a pas de paquet `OPAM` compatible avec les versions d'OCaml supérieures à 4.08. `BDDApron` a été utilisée par Schrammel et Jeannet [17] pour analyser statiquement des codes `Lustre`, qui sont d'une certaine façon similaire à des programmes Ladder, par l'aspect data-flow synchrone. Cette similarité renforce l'idée que des domaines à base de BDD sont bien adaptés dans un tel contexte.

Dans la communauté de l'interprétation abstraite en général, l'utilisation de BDD ne semble pas du tout être populaire. On peut arguer que de toute façon les programmes où les variables booléennes sont prépondérantes ne sont pas une cible habituelle. Néanmoins, notre approche nous semble aller plus loin que la simple idée d'interpréter les opérations booléennes : la structure de BDD encode très naturellement des disjonctions de contraintes.

Pour gérer de telles disjonctions, on trouve des approches comme la *powerset completion* [9], la *state partitioning* ou *trace partitioning* [16], les analyses *path-sensitive* [12]. On trouve dans la littérature une notion de *decision tree abstract domains* [6] dont les présentations sont beaucoup plus abstraites que la nôtre, et dont il ne semble pas exister d'implémentation disponible. Il reste que les idées proposées et le vocabulaire utilisé par Chen et Cousot [6] sont clairement similaires aux nôtres, et nous avons, sans le savoir au départ, redécouvert cette idée d'utiliser des BDD. Par contre, dans l'article en question on ne voit pas apparaître le besoin pratique d'une opération d'élimination des quantifications existentielles, qui est pour nous un ajout indispensable. Urban et Miné [18] utilisent aussi une structure similaire appelé *Decision Tree Abstract Domain*, ceci pour un but très spécifique de preuve de terminaison. On y retrouve l'idée que les arbres de décision binaires sont une structure intéressante comme alternative au partitioning.

De notre point de vue, il nous semble que les BDD ont une plus grande popularité dans les communautés SMT, model-checking et résolution de contraintes, que dans l'interprétation abstraite. Au vu de nos expérimentations, cela semblerait plutôt un oubli dommageable.

5.2 Perspectives

Notre prototype BDDinfer est actuellement très spécialisé sur du code ne comportant que des variables entières et des variables booléennes. Dès que l'on lui propose un code qui comporte d'autres types de données, aucun invariant n'est inféré. À court terme nous souhaitons étendre le support en rajoutant un domaine générique de fonctions non-interprétés (classe UF dans le jargon des solveurs SMT) qui permettra de traiter beaucoup plus de programmes. Cela sera l'occasion d'instancier notre foncteur de BDD paramétrés sur un autre domaine que des contraintes linéaires.

Comme évoqué en fin de la section 4, les améliorations d'efficacité et de précision nous permet maintenant la possibilité d'appliquer cette méthodologie de vérification à des formes de spécifications temporelles plus complexes.

Une question ouverte pour nous est d'identifier l'intérêt que peut avoir notre méthode très centrée sur les booléens dans la communauté de l'interprétation abstraite en général. Comme il ressort ci-dessus, nous ne savons pas bien comparer notre approche avec les techniques existantes de gestion des disjonctions. N'étant pas des spécialistes de l'interprétation abstraite, nous serions heureux que la publication de cet article puisse nous permettre d'obtenir un retour d'experts de cette communauté.

Remerciements

Nous remercions Cláudio Belo Lourenço, qui a mis à jour le prototype InferLoop en lui ajoutant un support des variables booléennes, et en le rendant plus robuste; Yacine El Haddad, qui a mis en place le prototype initial de l'interpréteur abstrait BDDinfer; et Jean-Christophe Filliâtre, dont la bibliothèque OCaml-BDD a permis le succès de notre approche, et qui a accepté d'y intégrer nos patches proposant des extensions. Nous remercions également Florian Faissolle, Inoue Hiroaki et David Mentré qui ont participé à définir notre méthode pour la vérification de programmes Ladder en regard d'un diagramme de temps.

Références

- [1] Lucas Baudin. Deductive verification with the help of abstract interpretation. Technical report, Univ Paris-Sud, November 2017. URL : <https://hal.inria.fr/hal-01634318>.
- [2] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles,

- and Nicky Williams. The dogged pursuit of bug-free C programs : The Frama-C software analysis platform. *Communications of the ACM*, 64(8) :56–68, 2021. doi:10.1145/3470569.
- [3] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language, version 1.16*, 2020. URL : <https://frama-c.com/html/acsl.html>.
- [4] Benedikt Becker, Cláudio Belo Lourenço, and Claude Marché. Explaining counterexamples with giant-step assertion checking. In José Creissac Campos and Andrei Paskevich, editors, *6th Workshop on Formal Integrated Development Environments (F-IDE 2021)*, Electronic Proceedings in Theoretical Computer Science, May 2021. URL : <https://hal.inria.fr/hal-03217393>, doi:10.4204/EPTCS.338.10.
- [5] Cláudio Belo Lourenço, Denis Cousineau, Florian Faissolle, Claude Marché, David Mentré, and Hiroaki Inoue. Automated formal analysis of temporal properties of Ladder programs. *International Journal on Software Tools for Technology Transfer*, 24(6) :977–997, 2022. URL : <https://hal.inria.fr/hal-03737869>, doi:10.1007/s10009-022-00680-0.
- [6] Junjie Chen and Patrick Cousot. A binary decision tree abstract domain functor. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis*, pages 36–53, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.
- [7] Sylvain Conchon and Jean-Christophe Filliâtre. Semi-persistent data structures. In *17th European Symposium on Programming (ESOP'08)*, Budapest, Hungary, April 2008. URL : <https://inria.hal.science/hal-04045849>, doi:10.1007/978-3-540-78739-6_25.
- [8] Yacine El Haddad and Claude Marché. Source code for bddinfer library. Git repository, May 2023. <https://gitlab.inria.fr/why3/why3/-/tree/master/src/bddinfer>.
- [9] Gilberto Filé and Francesco Ranzato. The powerset operator on abstract interpretations. *Theoretical Computer Science*, 222(1) :77–111, 1999. doi:10.1016/S0304-3975(98)00007-3.
- [10] Jean-Christophe Filliâtre. An OCaml library for binary decision diagrams, version 0.4. Git repository, 2018. <https://github.com/backtracking/ocaml-bdd>.
- [11] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013. URL : <http://hal.inria.fr/hal-00789533>.
- [12] Julien Henry, David Monniaux, and Matthieu Moy. PAGAI : A path sensitive static analyser. In *3d Workshop on Tools for Automatic Program Analysis (TAPAS'2012)*, volume 289 of *Electronic Notes in Theoretical Computer Science*, pages 15–25, 2012. doi:10.1016/j.entcs.2012.11.003.
- [13] Bertrand Jeannet. La bibliothèque BDDAPRON de domaines abstraits logico-numériques. Web site <https://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/bddapron/>.
- [14] Bertrand Jeannet and Antoine Miné. Apron : A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, pages 661–667. Springer, 2009.
- [15] Matthieu Journault, Antoine Miné, Raphaël Monat, and Abdelraouf Ouadjaout. Démonstration de la plateforme Mopsa d’analyse statique de programmes par interprétation abstraite. In *JFLA 2021 - 32^{èmes} Journées Francophones des Langages Applicatifs*, 2021. URL : <https://hal.science/hal-03190426>.
- [16] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans. Program. Lang. Syst.*, 29(5), 2007. doi:10.1145/1275497.1275501.

- [17] Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In Eran Yahav, editor, *Static Analysis Symposium*, volume 6887 of *Lecture Notes in Computer Science*, pages 233–248. Springer, 2011. doi:10.1007/978-3-642-23702-7_19.
- [18] Caterina Urban and Antoine Miné. A decision tree abstract domain for proving conditional termination. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis Symposium*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014. doi:10.1007/978-3-319-10936-7_19.

Comparing EventB, $\{log\}$ and Why3 Models of Sparse Sets

Maximiliano Cristiá¹ and Catherine Dubois²

¹Universidad Nacional de Rosario and CIFASIS, Rosario, Argentina

²ENSIIE, Inria, Université Paris-Saclay, LMF, France

Many representations for sets are available in programming languages libraries. This paper focuses on sparse sets and two of their operations used in some constraint solvers for representing integer variable domains, which are finite sets of values, as an alternative to range sequences. We formalize this data structure and two of its operations and prove their correctness, in three deductive formal verification tools, EventB, $\{log\}$ and Why3. Furthermore, we draw some comparisons regarding specifications and proofs.

1 Introduction

Sets are widely used in programs. They are sometimes first-class objects of programming languages, e.g. SETL [23] or $\{log\}$ [13], but more frequently they are data structures provided in libraries. Many different representations are available, depending on the targeted set operations. In this paper, we deal with sparse sets, introduced by Briggs and Torczon [6], used in different contexts and freely available for different programming languages (Rust, C++ and many others). We focus on their use in constraint solvers as an alternative to range sequences or bit vectors for implementing domains of integer variables [19] which are nothing else than mathematical finite sets of integers. With such an implementation, searching and removing an element are constant-time operations. Furthermore sparse sets are cheap to trail and restore, which is a key point when backtracking for finding solutions.

Confidence in constraint solvers using sparse sets can be improved if the algorithms implementing the main operations are formally verified, as it has been done by Ledein and Dubois [20] for the traditional implementation of domains as range sequences. Hence, the main contribution of this paper is a verified implementation of integer variable domains as sparse sets and the main operations used in constraint solvers (i.e. remove and bind) in EventB, $\{log\}$ and WhyML and their associated verification tools. We prove that the implemented operations preserve the invariant properties and we also express and prove properties that can be seen as formal foundations of trailing and restoring. As far as we know, this is the first formally verified implementation of some operations on sparse sets. All specifications and proofs can be found here: <https://gitlab.com/cdubois/SparseSets>.

We have chosen to use EventB and $\{log\}$ as representatives of set-based formalisms, both providing native high level operators on sets, relations and functions. However, in this family they are quite different: the former has an imperative flavor and offers refinement as a development method where models are specified gradually; the latter is based on the constraint logic programming (CLP) paradigm. Why3 is representative of deductive program verification tools manipulating contracts.

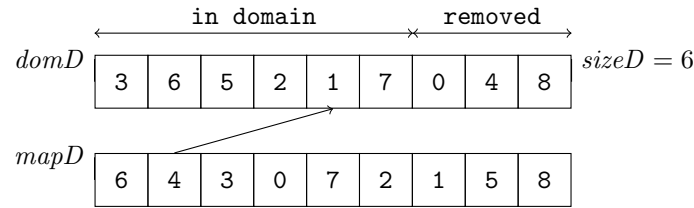


Figure 1. Example of a sparse set when $N=9$ and the current domain is $\{1, 2, 3, 5, 6, 7\}$

A second contribution of this paper is a comparison of these three formalizations with respect to aspects such as expressiveness, specification analysis and automated proof.

2 Sparse sets

We deal here with sets as subsets of natural numbers up to $N - 1$, where N is any non-zero natural number. A sparse set S is represented by two arrays of length N called $mapD$ and $domD$, and a natural number $sizeD$. The array $mapD$ maps any value $v \in [0, N - 1]$ to its index ind_v in $domD$, the value indexed by ind_v in $domD$ is v . The main idea that brings efficiency when removing an element or testing membership is to split $domD$ into two sub-arrays, $domD[0, sizeD - 1]$ and $domD[sizeD, N - 1]$, containing resp. the elements of S and the elements of $[0, N - 1]$ not in S . Then, if S is empty (resp. the full set), $sizeD$ is equal to 0 (resp. N). Fig. 1, inspired from a figure in [19], illustrates this representation.

Checking if an element i belongs to the sparse set S simply consists in the evaluation of the expression $mapD[i] < sizeD$. Removing an element from the set consists in moving this element to $domD[sizeD, N - 1]$ (with 2 swaps in $mapD$ and $domD$ and decreasing $sizeD$). Binding S to a single element of the set S follows the same idea: moving this element at the first place in $domD$ and assigning the value 1 to $sizeD$.

In our formalization, we only deal with two operations consisting in removing an element in a sparse set and binding a sparse set to a singleton set since these two operations are fundamental when solving constraints. Removing is necessary when domains are pruned. Binding a sparse set to a singleton set is done when the solver assigns variables. The solver never inserts values in a domain. Solvers may also need to walk through all the elements of a variable domain, exploring $domD[0..sizeD - 1]$. This is outside the scope of this work but it presents no particular difficulty.

Many constraint solvers use a data structure called *trail* to store undo information (such as domains) when backtracking on possible solutions. When sparse sets are used, only $sizeD$ needs to be kept in the trail. Domains can, then, be restored in constant time by setting the $sizeD$ variable back to its previous value [19].

Quoting Le Clément de Saint-Marcq et al. [19], there are three key predicates that should be invariants of any sparse set implementation:

- $D = \{domD[i] \mid 0 \leq i \leq sizeD\}$ (P₁)
- $mapD[v] = i \Leftrightarrow domD[i] = v$, for all i and v (P₂)
- $domD[sizeD .. N - 1]$ remains unchanged. (P₃)

These properties have been proved to hold in the three formalizations analyzed in this paper.

3 EventB formal development

In this section we succinctly introduce the EventB formal specification language and in more details the EventB models for sparse sets.

```

MACHINE Domain
VARIABLES D
INVARIANTS inv1:  $D \subseteq 0..N - 1$ 
Initialisation begin act1:  $D := 0..N - 1$  end
Event remove  $\hat{=}$ 
  any v where grd1:  $v \in D$ 
  then act1:  $D := D \setminus \{v\}$  end
Event bind  $\hat{=}$ 
  any v where grd1:  $v \in D$ 
  then act1:  $D := \{v\}$  end

```

Figure 2. EventB abstract specification, the Domain machine

3.1 EventB and Rodin

EventB [4] is a deductive formal method based on set theory and first order logic allowing users to design correct-by-construction systems. It relies on a state-based modeling language in which a model, called a machine, is made of a state characterized by variables and a collection of events describing state changes. The state consists of variables constrained by invariants. Proof obligations are generated to verify the preservation of invariants by events. A machine may use a context which introduces abstract sets, constants, axioms or theorems. A formal design in EventB starts with an abstract machine which is usually refined several times. Proof obligations are generated to verify the correctness of a refinement step.

An event may have parameters. When its guards are satisfied, its actions, if any, are executed, updating state variables. Actions may be -multiple- deterministic assignments, $x, y := e, f$, or -multiple- nondeterministic ones, $x, y :| BAP(x, x', y, y')$ where BAP is called a Before-After Predicate relating current (x, y) and next (x', y') values of state variables x and y . In the latter case, x and y are assigned arbitrary values satisfying the BAP predicate. When using such a non-deterministic form of assignment, a feasibility proof obligation (FIS) is generated in order to check that there exist values for x' and y' such that $BAP(x, x', y, y')$ holds when the invariants and guards hold. Furthermore when this kind of action is used and refined, the corresponding action in the refinement updating x and y is required to assign them values which satisfy the BAP predicate. A dedicated proof obligation called simulation (SIM) is automatically generated

In the following, we use Rodin, an Eclipse based IDE for EventB project management, model edition, refinement and proof, automatic proof obligations generation, model animation and code generation. Rodin supports automatic and interactive provers [16]. In this work we used the standard provers (AtelierB provers) and also the SMT solvers VeriT [3], CVC3 [1] and CVC4 [2]. More details about EventB and Rodin can be found in [4] and [5].

3.2 EventB formalization

The formalization is made of six components, i.e. two contexts, a machine and three refinements. Context Ctx introduces the bound N as a non-zero natural number and context $Ctx1$ extends the latter with helper theorems. The high level machine gives the abstract specification. This model contains a state composed of a finite set D , constrained to be a subset of the (integer) range $0..N - 1$, and two events, to remove an element from D or set D as a singleton set (see Fig. 2).

The first refinement (see Fig. 3) introduces the representation of the domain as a sparse set, i.e. two arrays $mapD$ and $domD$ modeled as total functions ($inv1$ and $inv2$) and also the variable $sizeD$ which is a natural number in the range $0..N$ ($inv3$). Invariants $inv4$ and $inv5$ constrain $mapD$ and $domD$ to be inverse functions of each other (property P_2 of Sect. 2). The gluing invariant $inv6$ relates the states between the concrete and former abstract machines¹. So the set $domD[0..sizeD - 1]$ containing the elements of the subarray from 0 to

¹In a refinement relationship, the machine which is refined is called the *abstract* machine whereas the refinement is called the *concrete* machine.


```

MACHINE SparseSets_ref1
REFINES Domain
SEES Ctx1
VARIABLES domD mapD sizeD
INVARIANTS
  inv1:  $domD \in 0..N-1 \rightarrow 0..N-1$     inv2:  $mapD \in 0..N-1 \rightarrow 0..N-1$ 
  inv3:  $sizeD \in 0..N$     inv4:  $domD ; mapD = id_{0..N-1}$ 
  inv5:  $mapD ; domD = id_{0..N-1}$     inv6:  $domD[0..sizeD-1] = D$ 
  inv7:  $\langle \text{theorem} \rangle \forall x, v \cdot x \in 0..N-1 \wedge v \in 0..N-1 \Rightarrow (mapD(v) = x \Leftrightarrow domD(x) = v)$ 
  inv8:  $\langle \text{theorem} \rangle domD \in 0..N-1 \mapsto 0..N-1$ 
Initialisation begin act1:  $mapD, domD := id_{0..N-1}, id_{0..N-1}$  act2:  $sizeD := N$  end
Event remove  $\hat{=}$  refines remove
  any v where  $v \in 0..N-1 \wedge 0 < sizeD \wedge mapD(v) < sizeD$ 
  then
     $mapD, domD, sizeD :| (domD' \in 0..N-1 \rightarrow 0..N-1 \wedge mapD' \in 0..N-1 \rightarrow 0..N-1$ 
     $\wedge domD' ; mapD' = id_{0..N-1} \wedge mapD' ; domD' = id_{0..N-1}$ 
     $\wedge domD'[0..sizeD'-1] = domD[0..sizeD-1] \setminus \{v\} \wedge \mathbf{sizeD}' < \mathbf{sizeD}$ 
     $\wedge (\mathbf{sizeD}..N-1) \triangleleft \mathbf{domD}' = (\mathbf{sizeD}..N-1) \triangleleft \mathbf{DomD}$ 
  end

```

Figure 3. EventB first refinement (excerpt)

```

Event remove  $\hat{=}$  refines remove
  any v where  $v \in 0..N-1 \wedge 0 < sizeD \wedge mapD(v) < sizeD$ 
  then act1:  $domD := domD \triangleleft \{mapD(v) \mapsto domD(sizeD-1), sizeD-1 \mapsto v\}$ 
    act2:  $mapD := mapD \triangleleft \{v \mapsto sizeD-1, domD(sizeD-1) \mapsto mapD(v)\}$ 
    act3:  $sizeD := sizeD-1$ 
  end

```

Figure 4. EventB Event remove in the second refinement

$sizeD-1$ is exactly the set D . This is exactly property P_1 of Sect. 2.

Theorems *inv7* and *inv8* are introduced to ease interactive proofs, they are proved as consequences of the previous formulas (*inv1* to *inv6*). *inv7* follows directly from a theorem of *Ctx1* whose statement is *inv7* where *domD* and *mapD* are universally quantified. Theorem *inv8* states that *domD* is an injective function.

Variables *mapD* and *domD* are both initialized to the identity function on $0..N-1$ (denoted $id_{0..N-1}$) and *sizeD* to N . Events of the initial machine are refined by non deterministic events. Thus *remove* assigns the three state variables with any values that satisfy invariants and also such that *sizeD* strictly decreases and removed elements in *domD* are kept at the same place (properties in bold font). The \triangleleft operator computes the domain restriction of a function or relation. Event *bind*, omitted in Fig. 3 for lack of space, follows the same pattern. The only reason to have introduced this intermediate model *SparseSets_ref1* is to express the properties written in bold font, one of them being the property P_3 of Sect. 2. In fact, because they relate two states, they cannot be expressed as invariants.

The second refinement has the same state than the previous one (see Fig. 4). Its events implement the operations and are a straightforward translation of the algorithms in [19].

To discharge the FIS proof obligations of *SparseSets_ref1*, we can use the values of *domD*, *mapD* and *sizeD* specified in *SparseSets_ref2* as witnesses. The SIM proof obligations of *SparseSets_ref2* require to prove that the latter values again satisfy the BAP predicate used in *SparseSets_ref1*. In order not to do these -interactive- proofs twice, we generalize them and prove them as theorems of the context. In this way, to provide a proof of the FIS and

SIM proof obligations, we only have to instantiate these theorems.

4 $\{log\}$ formal development

In this section we briefly present the $\{log\}$ tool and how we used it to encode sparse sets.

4.1 $\{log\}$

$\{log\}$ is at the same time a CLP language and satisfiability solver where sets and binary relations are first-class citizens [21, 17, 8]. The tool implements several decision procedures for expressive fragments of set theory and set relation algebra including cardinality constraints [15], restricted universal quantifiers [14], set-builder notation [10] and integer intervals [12]. Case studies developed with $\{log\}$ can be consulted in [9, 11, 7].

$\{log\}$ code enjoys the *formula-program duality* meaning that $\{log\}$ code can behave as both a formula and a program. When seen as a formula, it can be used as a specification on which verification conditions can be (sometimes automatically) proved. When seen as a program, it can be executed. Thus $\{log\}$ code is sometimes called *forgram*—a portmanteau word resulting from combining *formula* with *program*.

In the following formalization, we use the (still under development) state machine specification language (SMSL) defined on top of $\{log\}$. SMSL provides declarations very close to those of EventB to declare state variables (`variables`), operations (`operation`) and invariants (`invariant`). The latter is used to automatically generate verification conditions (VC) (proof obligations) on state machines. Users can use $\{log\}$ itself to automatically prove or disprove these VCs [22]. Unlike EventB, SMSL does not support the notion of refinement.

4.2 $\{log\}$ formalization

The $\{log\}$ formalization presented in this paper uses a combination of CLP and set-based, state-based specifications. While CLP is at the core of $\{log\}$, set-based, state-based specifications can be easily written by means of SMSL. Fig. 5 and 6 list representative parts of the $\{log\}$ forgram in which we use the same identifiers as for the EventB models as much as possible, within the syntactic constraints of $\{log\}$.

The $\{log\}$ forgram is mainly a state machine described with SMSL modifying 4 state variables (`DomD`, `MapD`, `SizeD`, `D`) by 2 operations, `remove` and `bind` (see Fig. 5). The two arrays are modelled by total functions and their *typing* constraints become invariant properties as in EventB (split here in small predicates to increase the chances of automated proofs). Property *P2* of Sect. 2 is also an invariant of this state machine (`inv4` and `inv5`, the latter, omitted in the figure, is the symmetric of the former). Parameter `I` is used to compute the identity relation on the integer interval $[0, N - 1]$ as shown in axiom `axm2`, which in turn is used in invariant `inv4`. $\{log\}$ inherits many of Prolog’s features. In particular, integer expressions are evaluated by means of the `is` predicate. Along the same lines, all set operators are implemented in $\{log\}$ as constraints. For example, `id(A,R)` is true when `R` is the identity relation on set `A`. The term `int(O,M)` corresponds to the integer interval $[0, M]$. Assertion `inv7` is introduced to help the solver², it can be deduced from previous invariants (as in Fig. 3). Therefore, we introduce it as a simple predicate but then we declare a theorem (`inv7_th`) whose conclusion is `inv7` but in a negated form because $\{log\}$ is a satisfiability solver. Later, $\{log\}$ will include `inv7_th` as a proof obligation and will attempt to discharge it. In `inv7`, the `foreach` constraint implements the notion of *restricted universal quantifier* (RUQ). That is, for some $\{log\}$ formula ϕ and set `A`, `foreach(X in A, $\phi(X)$)` corresponds to $\forall X.(X \in A \Rightarrow \phi(X))$ where `A` can be a set or a binary relation. In the latter case, the quantified expression can be an ordered pair, as is the case of `inv7` and `inv6` (in Fig. 6). $\{log\}$ also offers the `exists` constraint implementing the notion of

²Without it, some proofs are not automatic.

restricted existential quantifier (REQ) used in `inv6` to state a double set inclusion. The important point about REQ and RUQ is not only their expressiveness but the fact that there is a decision procedure involving them [14].

The `remove` operation is encoded as a `{log}` predicate. State variables are included as explicit arguments since in `{log}` there is no global state. Next-state variables are denoted by adding an underscore character to the base name (`SizeD_`). *Set unification* is used to implement function application. For instance, `DomD = {[S,Y2], [Y1,Y5] / DomD1}` is equivalent to: $\exists y_2, y_5, DomD_1. (DomD = \{(SizeD - 1, y_2), (y_1, y_5)\} \cup DomD_1)$, where $y_1 = MapD(v)$ (due to the previous set unification). Non-membership constraints following the equality constraint prevent `{log}` from generating repeated solutions. Hence, when `remove` is called with a set term in its fourth argument, this term is unified with `{[S,Y2], [Y1,Y5] / DomD1}`. If the unification succeeds, then the images of `S` and `Y1` are available.

```

parameters([N,I]).
variables([D,DomD,MapD,SizeD]).

axiom(axm1).      axm1(N) :- 1 =< N.
axiom(axm2).      axm2(N,I) :- M is N - 1 & id(int(0,M),I).

invariant(inv11). inv11(DomD) :- pfun(DomD).
invariant(inv12). inv12(N,DomD) :- N1 is N - 1 & dom(DomD,int(0,N1)).
invariant(inv13). inv13(N,DomD) :- N1 is N - 1 & ran(DomD,R) & subset(R,int(0,N1)).
invariant(inv4).  inv4(N,I,DomD,MapD) :- axm2(N,I) & comppf(DomD,MapD,I).

inv7(MapD,DomD) :- foreach([ [V,Y1] in MapD, [X,Y2] in DomD ],
  (Y1 = X implies Y2 = V) & (Y2 = V implies Y1 = X) ).
theorem(inv7_th).
inv7_th(N,MapD,DomD) :-
  neg(inv4(N,I,DomD,MapD) & inv5(N,I,DomD,MapD) implies inv7(MapD,DomD)).

operation(remove).
remove(N,SizeD,MapD,DomD,V,SizeD_,MapD_,DomD_) :-
  M is N - 1 & V in int(0,M) & 0 < SizeD & S is SizeD - 1 &
  MapD = {[V,Y1],[Y2,Y4] / MapD1} & disj({[V,Y1],[Y2,Y4]},MapD1) & Y1 < SizeD &
  DomD = {[S,Y2],[Y1,Y5] / DomD1} & disj({[S,Y2],[Y1,Y5]},DomD1) &
  DomD_ = {[S,V],[Y1,Y2] / DomD1} & MapD_ = {[V,S],[Y2,Y1] / MapD1} & SizeD_ = S.

```

Figure 5. Some representative axioms, invariants and operations of the `{log}` forgram

The state machine is complemented with some user-defined proof obligations (see Fig. 6) which are introduced as theorems to ensure that the `{log}` forgram verifies properties P_1 (`inv6` in the forgram) and P_3 . Precisely theorem `remove_pi_inv6` states that if `inv6` holds and `remove` and its abstract version³ (not shown in the paper) are executed, then `inv6` holds in the next state. Likewise, theorem `remove_b2` ensures that if `remove` is executed and the functional image⁴ of the interval `int(SizeD,N-1)` through `DomD_` is FI, then it must coincide with the functional image of the same interval but through `DomD`.

The VCs generated by `{log}` include satisfiability of the conjunction of all axioms, satisfiability of each operation and invariance lemmas for each and every operation and invariant. For invariance lemmas, `{log}` includes a minimum set of hypotheses in order to have to solve a simpler goal, reducing the possibilities of a complexity explosion. Hypotheses can be manually added and the proof run again. This process can be iterated until all proofs are done—or the complexity explosion cannot be avoided. The command `findh` helps the user to find missing hypotheses. `{log}` discharges all the VCs for the complete forgram.

³`remove` and its abstract version can be distinguished by their arities.

⁴`fimg` is a user-defined `{log}` predicate computing the relational image through a function.

```

inv6(D,DomD,SizeD) :-
  S is SizeD - 1 & foreach([X,Y] in DomD, X in int(0,S) implies Y in D) &
  foreach(X in D, exists([A,B] in DomD, A in int(0,S) & B = X)).

theorem(remove_pi_inv6).
remove_pi_invr6(N,SizeD,MapD,DomD,V,SizeD_,MapD_,DomD_) :- inv7(MapD,DomD) &
  neg(inv6(D,DomD,SizeD) & remove(V,D,D_) &
    remove(N,SizeD,MapD,DomD,V,SizeD_,MapD_,DomD_) implies inv6(D_,DomD_,SizeD_)).

theorem(remove_b2).
remove_b2(N,SizeD,MapD,DomD,V,SizeD_,MapD_,DomD_) :-
  neg(N1 is N - 1 & remove(N,SizeD,MapD,DomD,V,SizeD_,MapD_,DomD_) &
    fimg(int(SizeD,N1),DomD_,FI) implies fimg(int(SizeD,N1),DomD,FI)).

```

Figure 6. User-defined proof obligations

5 Why3 formal development

In this section we briefly introduce the Why3 platform and describe with some details our specification of sparse sets.

5.1 WhyML and Why3

Why3 [18] is a platform for deductive program verification providing a language for specification and programming, called WhyML. It relies on external automated and interactive theorem provers, to discharge VCs. Here we used the SMT provers CVC4 and Z3. Proof tactics are also provided, making Why3 a proof environment close to the one of Rodin for interactive proofs. Why3 supports modular verification.

WhyML allows the user to write functional or imperative programs featuring polymorphism, algebraic data types, pattern-matching, exceptions, references, arrays, etc. These programs can be annotated by contracts and assertions and thus verified. User-defined types with invariants can be introduced, the invariants are verified at the function call boundaries. Furthermore to prevent logical inconsistencies, Why3 generates a verification condition to show the existence of at least one value satisfying the invariant. To help the verification, a witness is explicitly given by the user (by clause in Fig. 7). The `old` operator can be used inside post-conditions to refer to the value of a term at the call program point. Programs may also contain ghost variables and ghost code to facilitate specification and verification. From verified WhyML programs, correct-by-construction OCaml programs (and recently C programs) can be automatically extracted.

5.2 Why3 formalization

We first define a record type, `sparse`, whose mutable fields are a record of type `sp_data` containing the computational elements of a sparse set representation and a ghost finite set of integer numbers which is the abstract model of the data structure. The type invariant of `sparse` relates the abstract model with the concrete representation as in Property P_1 of Sect. 2. It is used to enforce consistency between them. Invariants enforcing consistency between the two arrays `mapD` and `domD` and the bound `sizeD` are attached to the `sp_data` type: length of the arrays is `n`, contents are belonging to the integer range $0..n - 1$ and the two arrays are inverse of each other (Property P_2), `sized` is in $0..n$. These type definitions and related predicates are shown in Fig. 7.

Our formalization contains three functions, `swap_sp_data`, `remove_sparse` (see Fig. 8) and `bind_sparse` (omitted here), which update their arguments. They are the straightforward translation of the algorithms in [19], except for the supplementary ghost code (last

```

predicate ran (a: array int) (n: int) =
  0 <= n && a.length = n && forall i. 0<=i<n -> 0<=a[i]< n

type sp_data = {n: int; mutable domD, mapD : array int; mutable sizeD: int; }
invariant {ran domD n && ran mapD n && 0 <= sizeD <= n &&
  forall v,i. (0<=i<n && 0<=v<n) -> (domD[i]=v <-> mapD[v]=i)} by ...

type sparse = {mutable data: sp_data; mutable ghost setD: fset int;}
invariant {subset setD (interval 0 data.n) &&
  forall x: int.(exists i:int. 0 <= i < data.sizeD && x = data.domD[i]) <-> mem x setD} by ...

```

Figure 7. WhyML types for sparse sets

```

predicate same_end (a : array int) (b : array int) (s : int) (n : int) =
  forall i. s <= i < n -> a[i] = b[i]

let swap_sp_data (a : sp_data) (i : int) (j : int)
requires {0<=i<a.n && 0<=j<a.n}
ensures {exchange (old a.domD) a.domD i j}
ensures {exchange (old a.mapD) a.mapD a.domD[i] a.domD[j]} =
  swap a.domD i j; a.mapD[a.domD[i]] <- i; a.mapD[a.domD[j]] <- j;

let remove_sparse (v : int) (a : sparse)
requires {0<=v<a.data.n && a.data.mapD[v] < a.data.sizeD && a.data.sizeD > 0}
ensures {old a.data.sizeD > a.data.sizeD}
ensures {same_end a.data.domD (old a.data.domD) (old a.data.sizeD) a.data.n} =
  swap_sp_data a.data a.data.mapD[v] (a.data.sizeD - 1);
  a.data.sizeD <- a.data.sizeD - 1;
  a.setD <- remove v a.setD

```

Figure 8. WhyML functions for sparse sets

statement in `remove_sparse`) which updates the abstract model contained in `a.setD`. Function `swap_sparse_data` is a helper function. The contract of `swap_sparse_data` makes explicit the modifications of both arrays `a.mapD` and `a.domD`, using the `exchange` predicate defined in the library. VCs for this function concern the conformance of the code to the two post-conditions (trivial as it is ensured by `swap`) and also the preservation of the invariant attached to the `sparse_data` type—i.e. mainly that `a.mapD` and `a.domD` after swapping elements remain inverse from each other. Both `remove_sparse` and `bind_sparse` act not only on the two arrays and the bound but also on the ghost part, i.e. the corresponding mathematical set `a.setD`. Thus VCs here not only concern the structural invariants related to `mapD`, `domD` and `sizeD` but also the ones deriving from the use of the `sparse` type, proving the link between the abstract logical view (using finite sets) and the computational one implemented through arrays. The property P_3 is expressed here as a post-condition.

All proofs are discovered by the automatic provers except for some proof obligations related to the `remove` function. Nevertheless these interactive proofs remain simple thanks to some Why3 tactics that inject some hints to help external provers to finish the proofs.

6 Comparison and discussion

Clearly, all three formalisms and tools are expressive enough for the problem at hand. They all allow axioms, invariants and operations to be expressed. The EventB specification is probably the most readable. Properties P_1 and P_2 of Sect. 2 emphasised in [19] are expressed

TOOL	VC	AUTO	MANUAL
Rodin	46	34	12
$\{log\}$	38 (6)	38	0
Why3	53	51	2

Table 1. Summary of the verification efforts

as invariants in the three formalisms. Property P_3 about the removed part of the domain, which must relate two states, is expressed as a post-condition of the operations.

Writing P_3 in EventB proved to be complex. In fact, it was necessary to add a somewhat artificial level of refinement for Rodin to be able to generate the desired VCs link. This property can be more easily defined in $\{log\}$ and Why3.

In general, all three tools automatically generate similar VCs. However, in Why3 and EventB, abstract and concrete models can be naturally linked through refinement or ghost code and the tools automatically generate the corresponding VCs. $\{log\}$ still needs work to express how two models are linked in terms of abstraction/refinement relations. All VCs in EventB and Why3 are automatically generated, which is not the case in $\{log\}$, making the $\{log\}$ version of our verification effort less trustworthy than Why3 and Rodin.

Table 1 summarizes the results of the three verification efforts (for the two operations). The first column gives the number of VCs —numbers in brackets correspond to manually written VCs. The second (resp. third) column contains the number of automatically (resp. interactively) proved VCs.

In EventB, 46 proof obligations were generated (about half of them from the first refinement) of which 34 were automatically proved by the (AtelierB) standard provers and VeriT. For the 12 that were proved interactively, VeriT was very helpful when additional, back-up hypotheses were added. Only two proofs required real human intervention. Using the process described in Sect. 4, $\{log\}$ unloads all 38 VCs in about 7 minutes. Likely the existence of dedicated set-theoretic decision procedures proved crucial, since $\{log\}$ is the only tool that automatically discharges all VCs after a simple hypothesis discovery procedure.

Why3 makes it possible to apply transformations (e.g. split conjunctions) to a proof goal before calling an automatic prover on it. Some of these transformations are very simple, e.g. split conjunctions, and can then be applied systematically and automatically. Most of the VCs generated in our formalization have been proven automatically thanks to the split transformation. Only two of them, both dealing with type invariants, required human interaction to insert some more complex transformations, e.g., a case analysis on indices in `mapD` (case `(i=a_data.mapD[v])`). In the end, 53 VCs were proved—47 by CVC4 and 6 by Z3—in 9 seconds.

7 Conclusion

We formally verified the implementation of two operations on sparse sets using three formal languages and associated tools, focusing on the operations and correctness properties required by a constraint solver when domains of integer variables are implemented with sparse sets. In particular we compared the different statements of the required properties —namely P_1 , P_2 and P_3 given in Sect. 2— and their proofs.

As future work, the formal developments can be completed with other operations. A performance evaluation of the extracted code could then be performed. A second line of work is to implement and verify, in Why3 or EventB, a labelling procedure that assigns values to variables, such as those used in constraint solvers, it would be necessary to backtrack on the values of some domains and thus make use of the theorems proved in this paper. Since labelling is native in $\{log\}$ when CLP(FD) [24] is enabled, assignment of values to variables is trivial although less trustworthy than a formally verified algorithm.

References

- [1] cvc3. <https://cs.nyu.edu/acsys/cvc3/>.
- [2] cvc4. <https://cvc4.github.io/>.
- [3] verit. <https://verit.loria.fr/>.
- [4] J. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.
- [5] J. Abrial, M. J. Butler, S. arxlerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, 2010.
- [6] P. Briggs and L. Torczon. An efficient representation for sparse sets. *LOPLAS*, 2(1-4):59–69, 1993.
- [7] M. Cristiá, G. De Luca, and C. Luna. An automatically verified prototype of the android permissions system. *Journal of Automated Reasoning*, 67(2):17, May 2023.
- [8] M. Cristiá and G. Rossi. Solving quantifier-free first-order constraints over finite sets and binary relations. *J. Autom. Reason.*, 64(2):295–330, 2020.
- [9] M. Cristiá and G. Rossi. Automated proof of Bell-LaPadula security properties. *J. Autom. Reason.*, 65(4):463–478, 2021.
- [10] M. Cristiá and G. Rossi. Automated reasoning with restricted intensional sets. *J. Autom. Reason.*, 65(6):809–890, 2021.
- [11] M. Cristiá and G. Rossi. An automatically verified prototype of the Tokeneer ID station specification. *J. Autom. Reason.*, 65(8):1125–1151, 2021.
- [12] M. Cristiá and G. Rossi. A decision procedure for a theory of finite sets with finite integer intervals. *CoRR*, abs/2105.03005, 2021.
- [13] M. Cristiá and G. Rossi. $\{log\}$: set formulas as programs. *Rend. Ist. Mat. Univ. Trieste*, 53:24, 2021. Id/No 23.
- [14] M. Cristiá and G. Rossi. A set-theoretic decision procedure for quantifier-free, decidable languages extended with restricted quantifiers. *CoRR*, abs/2208.03518, 2022. Under consideration in *Journal of Automated Reasoning*.
- [15] M. Cristiá and G. Rossi. Integrating cardinality constraints into constraint logic programming with sets. *Theory Pract. Log. Program.*, 23(2):468–502, 2023.
- [16] D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. Integrating SMT solvers in rodin. *Sci. Comput. Program.*, 94:130–143, 2014.
- [17] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM Trans. Program. Lang. Syst.*, 22(5):861–931, 2000.
- [18] J. Filiâtre and A. Paskevich. Why3 - where programs meet provers. In M. Felleisen and P. Gardner, editors, *22nd European Symposium on Programming, ESOP 2013, Held as Part of ETAPS 2013, Rome, Italy, Proceedings*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.
- [19] V. Le Clément de Saint-Marcq, P. Schaus, C. Solnon, and C. Lecoutre. Sparse-Sets for Domain Implementation. In *CP workshop on Techniques foR Implementing Constraint programming Systems (TRICS)*, pages 1–10, 2013.

- [20] A. Ledein and C. Dubois. Facile en coq : vérification formelle des listes d'intervalles. In *31ème Journées Francophones des Langages Applicatifs*, 2019.
- [21] G. Rossi. `{log}`. <http://www.clpset.unipr.it/setlog.Home.html>, 2008. Last access 2023.
- [22] G. Rossi and M. Cristiá. `{log}` user's manual. Technical report, Dipartimento di Matematica, Università di Parma, 2020. <https://www.clpset.unipr.it/SETLOG/setlog-man.pdf>.
- [23] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets - An Introduction to SETL*. Texts and Monographs in Computer Science. Springer, 1986.
- [24] M. Triska. The finite domain constraint solver of SWI-Prolog. In *FLOPS*, volume 7294 of *LNCS*, pages 307–316, 2012.

Rough Pearl: Manufacturing Cons-Cells

Pierre-Évariste Dagand, Pierre Letouzey, and Ellenor Fatemeh Taghayor

Université Paris Cité, CNRS, IRIF

What if we could compose some data-types by first picking a recursive structure and then grafting data at selected locations, according to a well-defined blueprint? What if, moreover, this underlying structure had a straightforward semantics in terms of the number of elements it can support? This is the promise of *numerical representations*, which made it to the Functional Programming Pantheon through the seminal work of Okasaki and were recently celebrated at Collège de France by Leroy.

This work offers to develop a uniform presentation of these objects in dependent type theory. We abide by two overarching principles. First, we strive to provide a generic and uniform representation of this family of data-types. Second, we strive to exactly capture the data-logic of sizes and cardinality at play. It is in the act of balancing the two opposite forces of generality and fine-grained invariants that we hope to find a Pearl.

Year 3024. An extra-terrestrial vessel has been recovered from the confines of the Sagittarius Arm of the Milky Way. The vehicle is identified as “USS LVMH”, according to scriptures on the hull. Documents found on board suggest that this vessel was carrying away a wealthy clientele of happy-fews fleeing a dying planet called Earth. An in-depth study of the ship shows that this lost civilization had elaborate religious rituals (dubbed “Agile ceremony”, lead by “Scrum masters”) with a strong animist influence (a deity named “Coq” played a central role in their theology). They also mastered advanced programming concepts. Investigations reveal that the ship tragically veered off course due to a floating point error in a machine-checked procedure, exploiting a loss of subject reduction in their proof checker. An act of sabotage is not excluded at this point.

Like any sufficiently advanced civilization, their programming language is indistinguishable from a dependently-typed functional programming, as we use them here on Rama. This report aims at documenting some programming idioms found in their code base. We ask our readers to keep an open-mind, as notation and usage may seem alien. While the following results are part of the Ramaian programming folklore, we believe that the Earthlings cast them in a novel, interesting way.

This work is, in particular, concerned with numerical representations [Okasaki, 1999, Chapter 9]. A typical example of a numerical representation are the Peano natural numbers

```
type Nat ≜
  | • : Nat
  | (n : Nat) 1 : Nat
```

that correspond to unary numbers, that is numbers in base 1^k coefficiented by the digit **1**. We remark that Earthlings wrote these numbers with the most significant bit first, *i.e.* we

have

$$\begin{aligned} \text{Nat} \Rightarrow \mathbb{N} & \quad (\bullet \quad 1 \quad 1 \quad 1) \\ & = 0 + 1 \times 1^2 + 1 \times 1^1 + 1 \times 1^0 \end{aligned}$$

However, let us not be fooled by the postfix notation: as for traditional lists, this definition gives constant-time access to the least significant bit (the rightmost digit), as expected.

From the structure of unary numbers, we derive a *data*-structure: lists, which is obtained by ornamenting [Mcbride, 2010, Dagand, 2017] each digit 1 at position k with 1^k pieces of data

$$\begin{aligned} \text{type List } (A : \star) & \triangleq \\ & | \bullet : \text{List } A \\ & | (xs : \text{List } A) \triangleright (a : A) : \text{List } A \end{aligned}$$

In particular, lists inherit the left-leaning orientation of unary numbers. We recover the traditional “cons” operator by flipping both arguments. The symbol “ \star ” was pronounced “Type” by some Earthlings while it was pronounced “Set” by some others. Either way, it represents the (properly stratified) collection of all types.

The relationship between lists and numbers is materialized through the obvious projection that forgets the extra data and only keeps the recursive structure

$$\begin{aligned} \text{length } (xs : \text{List } A) & : \text{Nat} \\ \text{length } \bullet & \triangleq \bullet \\ \text{length } (xs \triangleright a) & \triangleq (\text{length } xs) \ 1 \end{aligned}$$

A key insight of numerical representations is that a significant part of the implementation and complexity analysis of data-structure can be systematically lifted up from the underlying numerical representation. For instance, since unary numbers give constant time access to the least significant digit, we obtain constant-time “cons” (by the very definition of `List`). We can also check that we can implement a constant-time predecessor over unary numbers

$$\begin{aligned} \text{pred } (n : \text{Nat}) & : \text{Nat} \\ \text{pred } \bullet & \triangleq \bullet \\ \text{pred } (n \ 1) & \triangleq n \end{aligned}$$

and blindly follow this guide to obtain the `tail` of a list, through copy-and-paste (most likely!) or more refined schemes [Dagand and McBride, 2014, Williams and Rémy, 2018]:

$$\begin{aligned} \text{tail } (xs : \text{List } A) & : \text{List } A \\ \text{tail } \bullet & \triangleq \bullet \\ \text{tail } (xs \triangleright _) & \triangleq xs \end{aligned}$$

Similarly, the recursive definition of addition of unary number

$$\begin{aligned} (m : \text{Nat}) + (n : \text{Nat}) & : \text{Nat} \\ \bullet + n & \triangleq n \\ (m \ 1) + n & \triangleq m + (n \ 1) \end{aligned}$$

is in fact specifying the size of the data-structure obtained by the concatenation of two lists. The tail-recursive implementation of addition thus yields the tail-recursive `rev_append` function

$$\begin{aligned} \text{rev_append } (xs \ ys : \text{List } A) & : \text{List } A \\ \text{rev_append } \bullet \ ys & \triangleq ys \\ \text{rev_append } (xs \triangleright a) \ ys & \triangleq \text{rev_append } xs \ (ys \triangleright a) \end{aligned}$$

Note that, wearing our computer scientist hat, we chose a tail-recursive, accumulator-based implementation of Peano addition. Had we been wearing our mathematician hat, we would probably have defined a stack-happy variant of addition as

$$\begin{aligned} (m : \text{Nat}) + (n : \text{Nat}) &: \text{Nat} \\ \bullet + n &\triangleq n \\ (m \ 1) + n &\triangleq (m + n) \ 1 \end{aligned}$$

which would directly yield a stack-happy, in-order append function over lists.

This approach allows decoupling the treatment of the data (here, singletons of A -elements) from the treatment of the structure (here, unary numbers). The latter is entirely handled at the numerical level while, on this example, the former is trivial.

Indeed, unary numbers enjoy a certain conceptual simplicity, offering reasoning principles that are familiar to all high-school students. However, this representation remains highly inefficient: they take up linear space rather than logarithmic space, as offered by traditional Arabic or binary numerals. It is somewhat straightforward to define a binary number representation:

```
type Bin ≙
  |      • : Bin
  | (bs : Bin) 0 : Bin
  | (bs : Bin) 1 : Bin
```

Once again, binary numbers are *written* with the most significant digit first

$$\begin{aligned} \text{Bin} \Rightarrow \mathbb{N} \quad & (\bullet \quad 1 \quad 0 \quad 1 \quad 1) \\ & = 0 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \end{aligned}$$

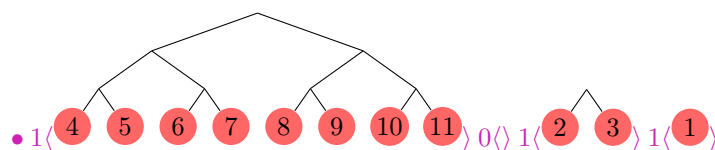
but we do get constant-time access to the least significant digit (rightmost constructor, following postfix notation).

One should however point out that this definition is not canonical: the number 0 can be coded as \bullet but also $\bullet \ 0$ and in fact an infinite number of ways as $\bullet \ 0 \dots 0$. Historical implementation of binary numbers strive to achieve canonicity [Agda standard library, Coq standard library], we shall see in a later section how this is handled in the context of numerical representations.

Random access-lists [Okasaki, 1999] provide logarithmic lookup and functional update operations, in addition to the usual interface of lists. They can be obtained by ornamenting each digit 1 at position k with 1×2^k pieces of data. For random-access lists, a suitable container for storing powers-of-2 elements would, for example, be leaf binary tree (complete binary tree where data is stored in the leaves). Similarly, each digit 0 at position k is decorated with $0 \times 2^k = 0$ pieces of data. If we are willing to trust the programmer in respecting the size invariants, we could define the following data-structure:

```
type RALOI(A : ★) ≙
  |      • : RALOI A
  |      (bs : RALOI A) 0⟨ ⟩ : RALOI A
  | (bs : RALOI A) 1⟨(t : LeafBinaryTree A)⟩ : RALOI A
```

An 11-elements inhabitant of this type would be:



The promise of numerical representations is that defining the successor function for binary numbers

$$\begin{aligned} \text{incr } (bs : \text{Bin}) &: \text{Bin} \\ \text{incr } \bullet &\triangleq \bullet 1 \\ \text{incr } (bs 0) &\triangleq bs 1 \\ \text{incr } (bs 1) &\triangleq (\text{incr } bs) 0 \end{aligned}$$

translates into a blueprint for **cons**-ing an element to a random-access list

$$\begin{aligned} \text{cons } (a : A)(bs : \text{RALOI } A) &: \text{RALOI } A \\ \text{cons } a \bullet &\triangleq \bullet 1 \langle a \rangle \\ \text{cons } a (bs 0 \langle \rangle) &\triangleq bs 1 \langle a \rangle \\ \text{cons } a_1 (bs 1 \langle a_2 \rangle) &\triangleq (\text{cons } (\text{link } a_1 a_2) bs) 0 \langle \rangle \end{aligned}$$

The only act of creativity left consists in appealing to **link** to join two leaf binary trees of 2^k so as to build a concatenated 2^{k+1} data container. This is an act of data management, all the structural work being handled by the definition of **incr**.

As witnessed by our discussion, the data container is subject to strict cardinality constraints, which are uniquely determined by the meaning of the numerical representation. For instance, we can belabor the “semantics” of the k -th digit of a binary number as follows

$$\begin{aligned} \text{Bin} \Rightarrow \mathbb{N}_k \bullet &= 0 \\ \text{Bin} \Rightarrow \mathbb{N}_k (bs 0) &= \text{Bin} \Rightarrow \mathbb{N}_{k+1} bs + 0 \times 2^k \\ \text{Bin} \Rightarrow \mathbb{N}_k (bs 1) &= \text{Bin} \Rightarrow \mathbb{N}_{k+1} bs + 1 \times 2^k \end{aligned}$$

Read as a specification, this tells us that data-structures built from binary numbers must not decorate the \bullet constructor while ensuring that each non-zero digit at position k is decorated with 2^k pieces of data. Note that the above definition of random-access list rely solely on the programmer’s discipline to ensure this invariant.

The distrusting nature of Earthling one-percenters led them to find ways to enforce these invariants through the type system. Using nested types [Bird and Meertens, 1998], Hinze [2001] has shown how well-sized data containers could be encoded in languages of the ML family. Nested types rely on non-uniform parameters to carry their mission

$$\begin{array}{l} \text{type } \text{RALOIN } (A : \star) \triangleq \\ \quad | \quad \bullet : \text{RALOIN } A \\ \quad | \quad (bs : \text{RALOIN } (A \times A)) 0 \langle \rangle : \text{RALOIN } A \\ \quad | \quad (bs : \text{RALOIN } (A \times A)) 1 \langle (a : A) \rangle : \text{RALOIN } A \end{array}$$

For instance, a random-access list with 11 elements would be encoded as thus

$$\bullet 1 \langle \langle \langle (4, 5), (6, 7) \rangle, (8, 9), (10, 11) \rangle \rangle 0 \langle \rangle 1 \langle (2, 3) \rangle 1 \langle 1 \rangle$$

This approach fell into disuse as it forcefully smashes intricate data containers (leaf binary trees in this case) down to a hodgepodge of binary products, making proper algorithmic work on the underlying data container meaningless. It also suffers from poor support in a dependently-typed setting, despite heroic efforts [Montin et al., 2022] made in that direction.

Our explorations suggest that an idiomatic, type-theoretic treatment of correct-by-construction numerical representations may be within our grasps. The presentation given in this paper is inspired by various experiments in the Coq and Agda proof assistants, at various levels of maturity and completeness. We therefore do not have any pretense concerning any machine-checked result.

Our objective is to hint at a compositional approach to numerical data-structure design. It draws inspiration from the work of Hinze [2001], which used nested types to encode a variety of numerical representations. The present paper can be understood as an actualization of this research program in a dependently-typed setting. We believe that one can combine an off-the-shelf numerical representation, a suitable data-container and, with minimal and localized efforts, obtain a data-structure and its key operations. At any rate, our experiments so far have been encouraging.

1 A Menagerie of Numerical Representations

In the following, we shall cover a carefully chosen selection of numerical representations. Each illustrates a salient property of numerical representation, which we intend to capture in our generic representation given in Section 2.

1.1 Well-typed or well-formed binary random-access lists

Rather than resort to nested types, we can exploit inductive families, presented either as an inductive predicate $\text{valid-RAL}\mathbb{1} : (k : \mathbb{N})(bs : \text{RAL}\mathbb{1} A) \rightarrow \star$, namely

$$\text{valid-RAL}\mathbb{1}_k \bullet \quad \frac{\text{valid-RAL}\mathbb{1}_{k+1} bs}{\text{valid-RAL}\mathbb{1}_k (bs \ 0\langle \rangle)} \quad \frac{\text{valid-RAL}\mathbb{1}_{k+1} bs \quad \text{size } t = 2^k}{\text{valid-RAL}\mathbb{1}_k (bs \ 1\langle t \rangle)}$$

$$\boxed{\text{valid-RAL}\mathbb{1} \triangleq \text{valid-RAL}\mathbb{1}_0}$$

or as an inductive family $\text{dRAL}\mathbb{1} : (k : \mathbb{N})(A : \star) \rightarrow \star$, namely

$$\text{type } \text{dRAL}\mathbb{1}_{(k:\mathbb{N})}(A : \star) \triangleq \begin{array}{l} | \\ | \\ | \end{array} \begin{array}{l} \bullet : \text{dRAL}\mathbb{1}_k A \\ (bs : \text{dRAL}\mathbb{1}_{k+1} A) \ 0\langle \rangle : \text{dRAL}\mathbb{1}_k A \\ (bs : \text{dRAL}\mathbb{1}_{k+1} A) \ 1\langle t : \text{LeafBinaryTree}_{2^k} A \rangle : \text{dRAL}\mathbb{1}_k A \end{array}$$

$$\boxed{\text{dRAL}\mathbb{1} \triangleq \text{dRAL}\mathbb{1}_0}$$

Conceptually, these data-*logics* can be understood as the fusion of $\text{RAL}\mathbb{1} \Rightarrow \mathbb{N}$ onto the underlying numerical data-structure, in a way reminiscent from algebraic ornaments [Dagand, 2017]. Note that these indexed types are *not* defining size-indexed structures (unlike, say, the perennial vector, *i.e.* length-indexed lists). Indices here are used to count the number of digits starting from the least significant one.

Take-away: we ought to be able to capture size invariants, either through intrinsic encoding (inductive family) or extrinsic encoding (inductive predicates).

1.2 Zeroless binary random-access lists

As hinted at earlier, lacking canonicity makes for cumbersome definitions (*e.g.*, when defining equality). Interestingly, this problem shows up during algorithmic work as well, for different reasons. Sometimes, canonicity is frowned upon so as to avoid having increment and decrement enter a “resonant” mode, with carries rippling left and right. This is the motivation behind data-structures such as 2-3 trees [Hinze, 2018]. Other times, a strict,

zero-less representation is adopted so as to improve the density of the data representation. For instance, 1-2 binary numbers [Hinze, 2001] is defined as

```

type Bin12 ≐
|           • : Bin12
| (bs : Bin12) 1 : Bin12
| (bs : Bin12) 2 : Bin12
    
```

which, in turn, leads to 1-2 binomial random-access list

```

type RAL12(A : ★) ≐
|           • : RAL12 A
|           (bs : RAL12 A) 1⟨(t : LeafBinaryTree A)⟩ : RAL12 A
| (bs : RAL12 A) 2⟨(t0 : LeafBinaryTree A)⟩⟨(t1 : LeafBinaryTree A)⟩ : RAL12 A
    
```

where every constructor (aside from \bullet) play an informational role, each of them storing some A . This is unlike the $0\langle \rangle$ constructor of the `RAL01` type, which played no other role than encoding the size of the underlying structure (a role which is nonetheless crucial from an operational standpoint).

Take-away: we ought to be able to support arbitrary sets of digits, especially ones without a zero.

1.3 Skew binary numbers/binomial heaps

An alternative approach to dispose of informationally-trivial 0s consists in adopting a “run-length encoded” presentation. Rather than having a constructor for the digit 0, one can decorate each non-zero digit with the number of *preceding* zeros. Alternatively, one could adopt a “sparse” representation, maintaining a set of pair of indices and coefficients for non-zero digits. From a type-theoretic standpoint, a sparse representation is trickier to deal with, for example to implement equality (*e.g.*, two representations are equal up to the order of the set of indices, opening the Doors of Setoid Hell¹).

Myers’ [Myers, 1983] skew binomial heap provides an historical example of run-length encoded representation, using a skew binary $(2^{k+1} - 1)$ base:

```

type SBin ≐
|           • : SBin
| (bs : SBin) 1(c:N) : SBin
| (bs : SBin) 2(c:N) : SBin
    
```

Here, the digits $1.^c.$ and $2.^c.$ carry an offset c . For instance, the digit $1.^c.$ at position k accounts for $1 \times (2^{(c+k)+1} - 1)$ elements, *i.e.* offsetting the base interpretation by c .

Note that, once again, this representation is not canonical. For example, $\bullet 2.^1. 1.^0.$ and $\bullet 2.^0. 2.^0.$ both denote the natural number 15. However, there exists a strict subset of skew binary numbers that is closed under increment, decrement and any other numerical operation one needs to implement a binomial heap. Moreover and amazingly, increment and decrement happen in *constant-time* on these canonical skew binary numbers: it is enough to look at the first 2 least significant digits to propagate the carry.

The normal form can be expeditiously characterized as “maybe start with a $2.^c.$ followed by any, possibly empty, number of $1.^i.$ ”. Piponi and Yorgey [2015] indicates how one could

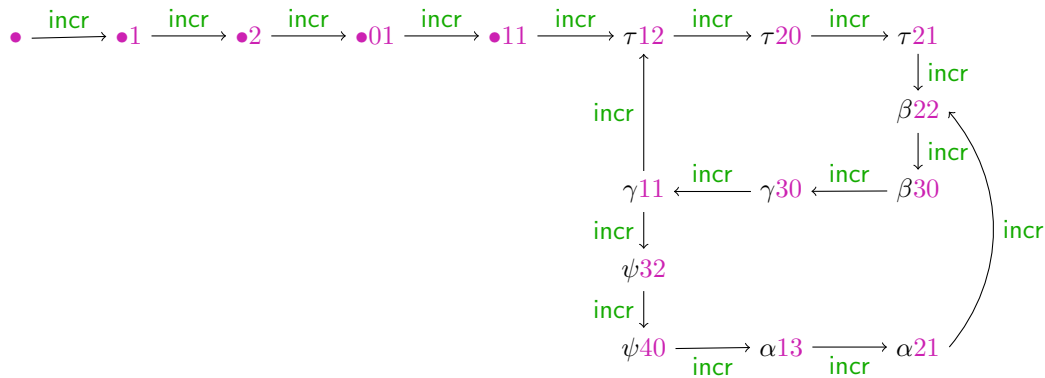
¹As soon as there are multiple *equivalent* representations of a number, we are condemned to show that all our operations over these representations produce equivalent results. In practice, this adds a significant burden to an implementation effort.

systematically turn such a regular expression into an indexing discipline for `SBin` and, by extension, its corresponding numerical representation. It is also straightforward to work it out by hand in this case.

Take-away: we ought to be able to enforce canonicity of the numerical representation, either through intrinsic encoding (inductive family) or extrinsic encoding (inductive predicates). Failing that, some operations could not be implemented with the proper complexity.

1.4 Magic skew binary, or “These Numbers All Go to Eleven”

Our investigation lead us to stumble upon a mesmerizing numbering system, magic skew binary [Elmasry et al., 2012]. Based on the skewed binary base $(2^{k+1} - 1)$, magic skew binary use digits between 0 and 4. As for Myers’ skew binary, only a strict subset of these numbers can be reached through the successor function. This subset has been characterized by Elmasry et al. [2012] as the union of 16 abstract forms: the successor function `incr` takes magic numbers from one form to another according to the following transitions:



The 5 symbolic states α , β , γ , ψ and τ can themselves be described as regular expressions (to be read from right-to-left, in keeping with our notations)

$$\begin{aligned} \tau &= 2^* | 12^* \\ \alpha &= \gamma 12^* \\ \beta &= 2^* | \tau 12^* | \psi 32^* \\ \gamma &= 1 | \tau 2 | \beta 3 | \psi 4 \\ \psi &= \gamma 0 | \alpha 1 \end{aligned}$$

For instance, the canonical representation of the number 10 amounts to

$$\text{Magic} \Rightarrow \mathbb{N} \left(\begin{array}{cccc} \bullet & 1 & 0 & 3 \\ = & 0 & + 1 \times (2^{2+1} - 1) & + 0 \times (2^{1+1} - 1) & + 3 \times (2^{0+1} - 1) \end{array} \right)$$

This numerical representation offers a proving ground to the various representational choices presented earlier. For instance, the 16 abstract forms could certainly (though painfully!) be turned into an inductive family of canonical-by-construction magical binary numbers. However, we are still left with implementing the successor function `incr`. In Elmasry et al. [2012], this is achieved through a two-step process “(1) increment and denormalize, then (2) fix and renormalize”. We have formalized and proved the correctness of this process over an extrinsic presentation in Coq. A canonical-by-construction, intrinsic representation calls for a radically different approach. Too radical to be found before the JFLA deadline.

Take-away: in some circumstances, we ought to have access to both canonical and non-canonical representations, operations over the former being implemented in terms of the latter.

2 Unified Numerical Representation

We can summarize our findings by offering a single, unified numerical representation. In our journey, we have identified 4 ingredients:

```

base : ℕ → ℕ
digits : ★
T : ★ → ★
size-T : T A → ℕ

```

We must first choose a numerical **base**. We have seen unary, 1^k , binary, 2^k , as well as skew binary $2^{k+1} - 1$. We must then choose a set of **digits**, *i.e.* the coefficients by which the base is multiplied. We have seen the singleton $\{1\}$, the sets $\{0, 1\}$, $\{1, 2\}$ or even $\{0, 1, 2, 3, 4\}$. We must choose a data container **T**, for which we shall ensure – through programmer’s discipline – or enforce – through proofs or type-checking – that the **size-T** of data container follow the specification set by **base**. We have witnessed the use of leaf binary trees and hinted at the use of binomial heaps.

Provided these ingredients, we can define a generic numerical representation:

```

type Num (A : ★) ≜
|
| (bs : Num A) [(d : digits)] [(c:ℕ)] : Num A

```

If we further enforce that offsets **c** are equal to 0 everywhere, we obtain a dense representation. Otherwise, this representation offers a run-length encoding by default. Similarly, **size-T** invariants and, further, canonical forms have to be enforced through further indexing or predicates on top of this base definition. We conjecture that there exists a sufficiently big crank that, once turned, yields the desired data-structures.

Note that operations (such as increment/insertion, decrement/deletion, addition/concatenation, *etc.*) remain to be engineered on a case-by-case basis. The overall methodology consists in first implementing the numerical operation (*e.g.*, increment), which is solely concerned with the numerical data-structure, from which one builds up the corresponding operation over the numerical data-type (*e.g.*, insertion).

In this context, unary numbers/lists could be (over-)engineered as an instance of

```

List-base (k : ℕ) : ℕ
List-base k ≜ 1k
type List-digits ≜
| 1 : List-digits
List-T (A : ★) : ★
List-T A ≜ A

```

Similarly, skew binary numbers/skew binomial heaps correspond to an instance of

```

SkewBin-base (k : ℕ) : ℕ
SkewBin-base k ≜ 2k+1 - 1
type SkewBin-digits ≜
| 1 : SkewBin-digits
| 2 : SkewBin-digits
SkewBin-T (A : ★) : ★
SkewBin-T A ≜ LeafBinaryTree A

```


3 Conclusion

The link between numerical representations and data-structures has been extensively explored by some of the giants of computer science [Knuth, 1998, Vuillemin, 1978]. Ever since Okasaki’s seminal work [Okasaki, 1999], it has been an integral part of the functional programming folklore. Recently, Leroy gave a crisp exposition (in French) as part of his 2023 lecture series at College de France². Following Hinze [2001], he showed how structural invariants and, in fact, the data containers could be built using nested types.

The present work was born out of the first author’s inability to simply capture these invariants in type theory. Other type-theorists, such as Ko and Gibbons [2017] and Swierstra [2020], tackled specific instances of the pattern but a general treatment remained elusive. We believe that this research program can succeed if (1) we strictly decouple the various levels of indexing involved, (2) we define the semantics of the underlying numerical systems through a carefully crafted recursive definition. We hope to be able to provide further evidences at the meeting.

Future work. This document builds up on various piecemeal experiments in Agda and Coq: a first order of business will consist in aggregating these results in a single, coherent library in both languages, exploiting their respective strengths and weaknesses (*e.g.*, dependent types *vs.* inductive predicates). In the process, we will have to clarify whether these definitions are obtained by internally *instantiating* generic Agda or Coq definitions, or through external *meta-programming*. Early experiments suggest that, barring some reasonable amount of duplication, a purely internalized presentation could be within reach.

A key motivation for the study of numerical representation lies in the promise of a compositional treatment of algorithmic complexity of data-structure operations, which would be decomposed as the cost of operations on the underlying data-container combined with the cost of arithmetic over the numerical representation. It would be interesting to fulfill this claim through machine-checked complexity proofs [Guéneau et al., 2018].

Our presentation has threaded very lightly around the topic of data containers. Being subject to strict size constraints, it is extremely tempting to use inductive families to enforce these invariants. However, there is a significant impedance mismatch to be solved at this interface. On the one hand, a numerical representation will want to impose precise cardinality constraint. On the other hand, typical data containers are indexed by more structural information (such as the height of a tree), which is only sometimes and indirectly related to the size (*e.g.*, log of the size for a leaf binary tree). Vectors and Braun trees [Okasaki, 1997] (which were suggested to us by Jean-Christophe Filliâtre) are among the few data-structures to be indexed by their actual size (in unary for vectors, in binary for Braun trees).

Conversely, numerical representations could be indexed by their size, as this is a standard case of reornamentation [Dagand, 2017]. However, it is not clear how exacting it would be to program with these definitions. Ultimately, we would certainly want our numerical representations to also satisfy the requirement for being data containers.

Finally, a grand challenge, which reaches beyond the grasps of our functional programming toolbox, would be to provide techniques and tools to identify the canonical forms of number systems. Judging by the work of Knuth [1998], Myers [1983] and Elmasry et al. [2012], these canonical forms are traditionally found by manually iterating the successor function starting from 0 and proving, after the fact, that these abstract forms are stable. Magic binary skew numbers offer a humbling testimony to the intellectual power of some of our colleagues.

Acknowledgments. Our interest in these questions was rekindled thanks to inspiring discussions with Catherine Dubois, Amélie Ledein, and Mathieu Montin. We are grateful to them for their time and enthusiasm.

²<https://www.college-de-france.fr/fr/agenda/cours/structures-de-donnees-persistantes/systemes-de-numeration-et-types-non-reguliers>

References

- Agda standard library. Binary numbers. <https://github.com/agda/agda-stdlib/blob/master/src/Data/Nat/Binary/Base.agda#L29-L32>.
- R. S. Bird and L. G. L. T. Meertens. Nested datatypes. In J. Jeuring, editor, *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, volume 1422 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 1998. doi: 10.1007/BFb0054285. URL <https://doi.org/10.1007/BFb0054285>.
- Coq standard library. Binary numbers. <https://github.com/coq/coq/blob/master/theories/Numbers/BinNums.v#L21-L24>.
- P. Dagand. The essence of ornaments. *J. Funct. Program.*, 27:e9, 2017. doi: 10.1017/S0956796816000356. URL <https://doi.org/10.1017/S0956796816000356>.
- P. Dagand and C. McBride. Transporting functions across ornaments. *J. Funct. Program.*, 24(2-3):316–383, 2014. doi: 10.1017/S0956796814000069. URL <https://doi.org/10.1017/S0956796814000069>.
- A. Elmasry, C. Jensen, and J. Katajainen. Two skew-binary numeral systems and one application. *Theory Comput. Syst.*, 50(1):185–211, 2012. doi: 10.1007/s00224-011-9357-0. URL <https://doi.org/10.1007/s00224-011-9357-0>.
- A. Guéneau, A. Charguéraud, and F. Pottier. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In A. Ahmed, editor, *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, 2018. doi: 10.1007/978-3-319-89884-1_19. URL https://doi.org/10.1007/978-3-319-89884-1_19.
- R. Hinze. Manufacturing datatypes. *J. Funct. Program.*, 11(5):493–524, 2001. doi: 10.1017/S095679680100404X. URL <https://doi.org/10.1017/S095679680100404X>.
- R. Hinze. On constructing 2-3 trees. *J. Funct. Program.*, 28:e19, 2018. doi: 10.1017/S0956796818000187. URL <https://doi.org/10.1017/S0956796818000187>.
- D. E. Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition*. Addison-Wesley, 1998. ISBN 0201896842. URL <https://www.worldcat.org/oclc/312898417>.
- H. Ko and J. Gibbons. Programming with ornaments. *J. Funct. Program.*, 27:e2, 2017. doi: 10.1017/S0956796816000307. URL <https://doi.org/10.1017/S0956796816000307>.
- C. McBride. Ornamental algebras, algebraic ornaments. Manuscript available online, 2010. URL <http://personal.cis.strath.ac.uk/~conor/pub/OAAO/Ornament.pdf>.
- M. Montin, A. Ledein, and C. Dubois. Libndt: Towards a formal library on spreadable properties over linked nested datatypes. In J. Gibbons and M. S. New, editors, *Proceedings Ninth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2022, Munich, Germany, 2nd April 2022*, volume 360 of *EPTCS*, pages 27–44, 2022. doi: 10.4204/EPTCS.360.2. URL <https://doi.org/10.4204/EPTCS.360.2>.
- E. W. Myers. An applicative random-access stack. *Inf. Process. Lett.*, 17(5):241–248, 1983. doi: 10.1016/0020-0190(83)90106-0. URL [https://doi.org/10.1016/0020-0190\(83\)90106-0](https://doi.org/10.1016/0020-0190(83)90106-0).

- C. Okasaki. Three algorithms on braun trees. *J. Funct. Program.*, 7(6):661–666, 1997. doi: 10.1017/s0956796897002876. URL <https://doi.org/10.1017/s0956796897002876>.
- C. Okasaki. *Purely functional data structures*. Cambridge University Press, 1999. ISBN 978-0-521-66350-2.
- D. Piponi and B. A. Yorgey. Polynomial functors constrained by regular expressions. In R. Hinze and J. Voigtländer, editors, *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, volume 9129 of *Lecture Notes in Computer Science*, pages 113–136. Springer, 2015. doi: 10.1007/978-3-319-19797-5_6. URL https://doi.org/10.1007/978-3-319-19797-5_6.
- W. Swierstra. Heterogeneous binary random-access lists. *J. Funct. Program.*, 30:e10, 2020. doi: 10.1017/S0956796820000064. URL <https://doi.org/10.1017/S0956796820000064>.
- J. Vuillemin. A data structure for manipulating priority queues. *Commun. ACM*, 21(4):309–315, 1978. doi: 10.1145/359460.359478. URL <https://doi.org/10.1145/359460.359478>.
- T. Williams and D. Rémy. A principled approach to ornamentation in ML. *Proc. ACM Program. Lang.*, 2(POPL):21:1–21:30, 2018. doi: 10.1145/3158109. URL <https://doi.org/10.1145/3158109>.

SÛRETÉ DU LOGICIEL

Réutilisations de caches et d’invariants pour l’analyse statique incrémentale

Mamy Razafintsonina^{1,2}, David Bühler¹, Antoine Miné²,
Valentin Perrelle¹ et Julien Signoles¹

¹Université Paris-Saclay, CEA, List, Palaiseau, France

²Sorbonne Université, CNRS, LIP6, Paris, France

L’analyse statique de programmes permet aujourd’hui d’analyser des programmes de grande taille, avec une très bonne précision, tout en étant raisonnablement rapide. Néanmoins, les temps d’analyse continuent de se compter en minutes, voire dizaines de minutes, ce qui rend compliqué leur intégration dans les processus de développement : les modifications d’un programme y sont très fréquentes et requièrent donc d’obtenir rapidement les résultats de l’analyseur. Néanmoins, ces modifications sont souvent mineures, de l’ordre de quelques lignes de code tout au plus. L’analyse statique incrémentale exploite cette caractéristique pour permettre à un analyseur statique de se contenter d’actualiser les résultats d’une analyse antérieure plutôt que de tout recalculer, ce qui permet des gains de temps significatifs. Cet article présente deux nouvelles approches pour l’analyse statique incrémentale, l’une réutilisant des caches de fonction et l’autre des invariants de boucle. Nous les avons implémentées dans *Eva*, l’analyseur de valeurs par interprétation abstraite de *Frama-C* en utilisant une nouvelle fonctionnalité de cette plateforme permettant de comparer deux programmes. Nos travaux ont été évalués sur un ensemble de commits de programmes réels.

1 Introduction

Les logiciels modernes sont caractérisés par leur complexité, leur taille, mais aussi par leur évolution constante, que ce soit pour ajouter de nouvelles fonctionnalités, corriger des erreurs ou répondre à de nouvelles exigences. Cette évolutivité constante rend l’analyse statique de programme à la fois cruciale et difficile. Cruciale, car les bugs dans les logiciels peuvent avoir des conséquences désastreuses, particulièrement sur les systèmes critiques : il est donc souvent nécessaire de lancer des analyses sur ceux-ci, afin d’assurer la qualité du logiciel et de détecter d’éventuels problèmes le plus tôt possible. Mais cela est aussi difficile, car ces analyses prennent beaucoup de temps en raison de la taille du code, alors même que les changements apportés à chaque modification sont souvent petits et très localisés. L’analyse statique incrémentale est une solution prometteuse pour pallier ces problèmes. En effet, elle tire parti de la similarité entre différentes versions d’un même programme, afin de permettre une analyse plus rapide en réutilisant les résultats des analyses précédentes. Dans l’état de l’art [MD15, MOJ18], l’analyse complète d’un programme est souvent privilégiée pour garantir une approche sûre, sans omission d’alarmes. Toutefois, en pratique, l’analyse incrémentale, bien que non complète, peut apporter plus de rapidité, en assouplissant possiblement la sûreté [CDD⁺15, MGR13]. Le dilemme réside donc dans la détermination précise des portions du programme à ré-analyser et celles dont les résultats peuvent être réutilisés

des analyses précédentes. En effet, un changement syntaxique minime peut engendrer des effets sémantiques importants sur des parties syntaxiquement non modifiées du programme, rendant la ré-analyse potentiellement plus coûteuse que la modification elle-même.

Dans cet article, nous proposons une méthode d'analyse incrémentale pour l'interprétation abstraite [CC77, Cou21], fondée sur une correspondance de programmes et des heuristiques pour identifier de manière sûre des résultats d'analyse de fonctions et de boucles qui peuvent être réutilisés pour accélérer l'analyse du nouveau programme. L'approche choisie garantit la sûreté, tout en préservant une grande précision. Elle est générique et peut être implémentée sur n'importe quel interpréteur abstrait. Nous avons choisi ici le greffon *Eva* [BBY17] de la plateforme *Frama-C* [BBB⁺21] dédiée à l'analyse de code C. Cette implémentation a été évaluée sur un ensemble de commits issus du développement de deux programmes réels, *PolarSSL* et *Monocypher*. Cette évaluation montre une réduction significative du temps d'analyse contre une légère hausse de la consommation mémoire, tout en limitant la perte de précision par rapport à une analyse traditionnelle. En offrant une analyse incrémentale plus rapide, tout en conservant la sûreté et une grande précision, nous espérons favoriser l'intégration des analyseurs statiques dans les processus de développement [JSMHB13, CB16], notamment ceux d'intégration continue. En résumé, nos contributions sont les suivantes :

- une technique de correspondance de programmes pour identifier les fonctions non modifiées ;
- une approche basée sur une heuristique pour la correspondance entre les conditions de boucle de deux fonctions modifiées ;
- une approche sûre pour réutiliser le cache de l'analyse d'un programme précédent pour accélérer l'analyse d'un programme modifié ou non.

Travaux connexes Les techniques de mise en cache et de réutilisation des résultats de calculs, en particulier celles fondées sur la mémoïsation des fonctions pures [ALL96, FT90], ont été largement étudiées et appliquées, y compris dans l'analyse de programmes, remontant au moins au développement d'analyses flot de données pour supporter la compilation continue et réactive [Ryd83]. Des travaux plus récents ont contribué à l'incrémentalisation de plusieurs classes d'analyse de programmes, notamment les analyses flot de données [AB14, DAL⁺17], et les analyses fondées sur des extensions de *Datalog* [SEV16]. Ces approches, bien qu'efficaces, sont généralement limitées à des classes d'analyses spécifiques, imposant des restrictions sur les domaines abstraits en excluant certains domaines comme ceux de hauteur infinie.

Les analyses modulaires [CC02], fondées sur les résumés [VJL07, MGR13, O'H18, CDOY11, VdPSVEDR20, CDD⁺15] s'étendent naturellement à l'analyse incrémentale. En particulier, [VdPSVEDR20] propose un cadre modulaire d'analyse flot de donnée fondée sur un algorithme de liste de travail. Lors d'une analyse incrémentale, les composants affectés par les modifications sont initialement ajoutés dans la liste, et ceux indirectement affectés le sont au fur et à mesure. *Infer* [CDD⁺15] et *Coverity* [MGR13] supportent l'analyse incrémentale et fonctionnent sur des programmes de grande taille. Cependant, leurs approches n'offrent pas de garantie de sûreté. En particulier, l'analyse incrémentale d'*Infer* retourne seulement les nouvelles alarmes des fonctions analysées. Un problème partagé des résumés est la perte de précision induite par le formalisme utilisé. Une analyse modulaire sûre implique un résumé capable de représenter tous les contextes d'appels possibles, ce qui induit un compromis entre précision et coût de l'analyse. Notre approche est inspirée des résumés et garantit la sûreté, tout en préservant une grande précision de l'analyse.

Les travaux dans [SEV20, EST⁺22] sont plus récents et proposent des approches pour améliorer la précision. L'analyse incrémentale est implémentée sur un solveur générique de point-fixe [SV21]. Ils montrent que les structures de données utilisées par le solveur peuvent être exploitées pour limiter les parties à ré-analyser après une modification. [SCS21] propose une nouvelle sémantique opérationnelle qui réifie l'interprétation abstraite sur un graphe acyclique orienté, maintient les dépendances sur le graphe et implémente l'incrémentalité sur le graphe en invalidant les nœuds affectés par les modifications. Cette approche nécessite des constructions dédiées pour les boucles, les appels de fonctions dynamiques ou les récursions.

Cependant, ces travaux n'ont pas été évalués sur des programmes réels et les cas d'étude sont limités à des programmes simples. [NEH19] propose une approche réutilisant les points-fixes précédemment calculés, afin d'accélérer l'analyse des programmes dynamiques comme JavaScript. Leur approche se fonde sur une méthode de comparaison de code source pour faire la correspondance entre deux programmes. La sûreté est garantie par une analyse complète du programme. Cette technique est proche de la nôtre. Cependant, étant donné que la précision de l'analyse dépend directement de la méthode de comparaison et correspondance, nous nous concentrons seulement sur la réutilisation des points-fixes de boucle des fonctions modifiées. Mopsa [MOJ18, MOM20] et Astrée [MD15] utilisent des caches pour les fonctions et les itérations de point-fixe pour accélérer l'analyse interprocédurale et multi-processus. Cependant, ces caches ne sont pas conçus pour l'analyse incrémentale et ne sont pas réutilisés pour les versions suivantes du programme.

D'autres travaux se concentrent plutôt sur la comparaison de programmes. Ainsi, les travaux dans [FMB⁺14, DP16] proposent des algorithmes efficaces pour transformer un AST en un autre, avec des séquences de modifications appelées `edit scripts`. Ces approches permettent de trouver une séquence minimale de modifications pour transformer un AST en un autre, mais ne garantissent pas l'équivalence sémantique entre deux programmes, qui est nécessaire pour identifier les résultats réutilisables de manière sûre. L'application de ces approches à l'analyse incrémentale nécessite une étude plus approfondie. Cela pourrait être une piste intéressante pour améliorer notre approche.

Les travaux dans [DM19] se concentrent sur la comparaison de programmes au niveau de la sémantique, en utilisant une nouvelle sémantique dénotationnelle pour vérifier si deux programmes se comportent de la même manière lorsqu'ils sont exécutés avec les mêmes entrées. Cette approche est intéressante, notamment pour inférer que deux fonctions sont équivalentes. L'application de cette approche à l'analyse incrémentale n'est pas étudiée.

Plan de l'article L'article est organisé comme suit. La section 2 fournit un contexte théorique introductif sur l'interprétation abstraite. La section 3 présente la technique sur quelques exemples motivateurs. La section 4 présente les outils existants dans Frama-C sur lesquels notre approche repose. La section 5 présente notre approche pour l'analyse incrémentale. La section 6 discute de l'implémentation et de l'évaluation de notre approche. Finalement, la section 7 conclut l'article et présente quelques perspectives de recherche.

2 Préliminaires

L'interprétation abstraite [CC77, Cou21] est une théorie générale permettant de définir des domaines abstraits pour sur-approximer de manière sûre les comportements des programmes, tout en fournissant des garanties formelles sur les propriétés analysées.

Analyse d'un programme Soit P un programme, \mathcal{L} l'ensemble de tous les emplacements mémoires possibles dans P et \mathbb{V} l'ensemble de toutes les valeurs qu'on peut stocker dans un emplacement mémoire. Une mémoire $M : \mathcal{L} \rightarrow \mathbb{V}$ est une fonction qui associe à chaque emplacement mémoire $l \in \mathcal{L}$ une valeur $v \in \mathbb{V}$. Soit $\mathcal{D} \stackrel{def}{=} \mathcal{P}(M)$ le domaine concret représentant l'ensemble de tous les états mémoires possibles du programme P . Un domaine abstrait \mathcal{D}^\sharp est un ensemble qui sur-approxime l'ensemble des états concrets dans $\mathcal{P}(M)$. Formellement, \mathcal{D}^\sharp est un treillis muni d'une relation d'ordre \sqsubseteq , d'une opération d'union (resp. d'intersection) abstraite \sqcup^\sharp (resp. \sqcap^\sharp) et d'un plus petit (resp. plus grand) élément \perp (resp. \top). Nous introduisons $\gamma : \mathcal{D}^\sharp \rightarrow \mathcal{P}(M)$ la fonction de concrétisation qui associe à chaque état abstrait dans \mathcal{D}^\sharp un état concret dans $\mathcal{P}(M)$. Cette fonction permet de convertir un état abstrait vers un état concret.

Pour toute opération concrète $F : \mathcal{P}(M) \rightarrow \mathcal{P}(M)$ et sa version abstraite $F^\sharp : \mathcal{D}^\sharp \rightarrow \mathcal{D}^\sharp$, la propriété de sûreté $\forall X^\sharp \in \mathcal{D}^\sharp, \gamma(F^\sharp(X^\sharp)) \supseteq F(\gamma(X^\sharp))$ garantit que l'opération abstraite F^\sharp est une sur-approximation de l'opération concrète F .

Analyse d'une boucle L'analyse d'une boucle calcule un point-fixe en itérant sur son corps et en accumulant les états accessibles à chaque itération, jusqu'à l'obtention d'un état stable, appelé invariant. Soit $(\mathcal{D}^\sharp, \sqsubseteq)$, $X^\sharp \in \mathcal{D}^\sharp$ et un opérateur abstrait $F^\sharp : \mathcal{D}^\sharp \rightarrow \mathcal{D}^\sharp$, le test de stabilité est vérifiée par $F^\sharp(X^\sharp) \sqsubseteq X^\sharp$. Dans le concret, on a un (plus petit) point-fixe $X = F(X)$, qui correspond au meilleur invariant inductif. Dans l'abstrait, on a un post-point-fixe $F^\sharp(X^\sharp) \sqsubseteq X^\sharp$, qui n'est pas forcément un point-fixe. De plus, l'existence d'un point-fixe n'est pas toujours garantie, mais celle du post-point-fixe l'est : il suffit de choisir \top . Le cadre de l'interprétation abstraite garantit l'existence du post-point-fixe abstrait, qui est une abstraction sûre du point-fixe concret. Le nombre d'itérations pour l'atteindre peut néanmoins être important. Pour accélérer la convergence au prix d'une précision moindre, une analyse par interprétation abstraite utilise un opérateur spécial, appelé élargissement et noté ∇ . Plus précisément, $\nabla : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \rightarrow \mathcal{D}^\sharp$ est un opérateur binaire d'un domaine abstrait $(\mathcal{D}^\sharp, \sqsubseteq)$ s'il calcule une borne supérieure pour toute paire $X^\sharp, Y^\sharp \in \mathcal{D}^\sharp$, c'est-à-dire si $X^\sharp \sqsubseteq X^\sharp \nabla Y^\sharp$ et $Y^\sharp \sqsubseteq Y^\sharp \nabla X^\sharp$, et si, pour toute séquence $Y_i^\sharp \in \mathcal{D}^\sharp$, la séquence X_i^\sharp définie par $X_0^\sharp = Y_0^\sharp$ et $X_{n+1}^\sharp = X_n^\sharp \nabla Y_{n+1}^\sharp$ converge en temps fini. Cet opérateur est intégré dans l'algorithme de calcul du point-fixe. Considérons une boucle dont l'état abstrait initial est X_0^\sharp . À chaque itération n , le nouvel état abstrait X_{n+1}^\sharp est calculé en appliquant l'opérateur abstrait F^\sharp sur l'état courant X_n^\sharp , c'est-à-dire $X_{n+1}^\sharp = F^\sharp(X_n^\sharp)$. Plutôt que d'utiliser directement X_{n+1}^\sharp , on utilise l'opérateur d'élargissement ∇ pour calculer $X_{n+1}^\sharp = X_n^\sharp \nabla F^\sharp(X_n^\sharp)$. En pratique, ∇ est généralement appliqué après un certain nombre d'itérations sans élargissement pour rester précis, avant d'accélérer la convergence.

3 Cas pratiques et illustrations des concepts

Dans cette section, nous présentons des exemples de programmes qui illustrent l'intérêt de l'analyse incrémentale. Nous présentons deux approches différentes, mais complémentaires, pour accélérer l'analyse incrémentale. Considérons le programme suivant :

```

1 // Compute the area of a square (Changed)
2 int calculateArea(int side) {
3 - return side + side; // Incorrect formula
4 + return side * side; // Correct formula
5 }
6 // Compute the perimeter of a square (Unchanged)
7 int calculatePerimeter(int side) {
8 return 4 * side;
9 }
10 // Main function (Unchanged)
11 void main() {
12 int side = 5;
13 // side = [5; 5]
14 int area = calculateArea(side);
15 // side = [5; 5], area = [10; 10]
16 int perimeter = calculatePerimeter(side);
17 // side = [5; 5], area = [10; 10], perimeter = [20; 20]
18 printf("Area: %d, Perimeter: %d\n", area, perimeter);
19 }

```

Ici, la fonction `calculateArea` calcule l'aire d'un carré. La formule initiale à la ligne 3 est incorrecte. Une version ultérieure du code la corrige par la ligne 4. Les deux autres fonctions, `calculatePerimeter` et `main`, demeurent inchangées. En effectuant une analyse d'intervalle sur le programme incorrect, nous obtenons les résultats d'analyse indiqués en commentaires dans la fonction `main`. Ils associent, à la fin de chaque instruction de cette fonction, un intervalle (ici, singleton) à chaque variable du programme. Par exemple, à la ligne 15, l'intervalle pour la variable `area` après le calcul incorrect est le singleton `[10; 10]`.

Dans le cas d'une analyse statique non incrémentale, toute modification dans une fonction comme `calculateArea` nécessiterait une nouvelle analyse complète de la fonction `main` et de toutes les fonctions qu'elle appelle, y compris `calculatePerimeter`. Ce processus peut être très coûteux en ressource, surtout si les fonctions en question sont complexes. Cependant ici, la modification est strictement localisée à la fonction `calculateArea`. Dès lors, il est possible de réutiliser les résultats d'analyse précédents pour la fonction `calculatePerimeter`. Pour être plus précis, lors de la première analyse, nous sauvegardons la paire (I^\sharp, O^\sharp) de toutes les fonctions analysées. Ces états abstraits I^\sharp et O^\sharp représentent l'ensemble des états mémoires accessibles en entrée et en sortie de la fonction. Une abstraction, comme les intervalles par exemple, ne représente que la borne inférieure et la borne supérieure pour chaque variable. Nous verrons dans la section 4 quelles sont les limitations de cette approche. Pour la fonction `calculatePerimeter`, nous sauvegardons la paire d'entrée/sortie $([5; 5], [20; 20])$. Lors d'analyses ultérieures, en présence d'un appel à `calculatePerimeter`, deux conditions doivent être vérifiées : d'une part, la fonction elle-même ne doit pas avoir subi de modification syntaxique ni appeler de fonction en ayant subi une et, d'autre part, l'intervalle I_{new}^\sharp associé au paramètre d'entrée `side` doit être identique à celui I^\sharp de l'analyse précédente (par exemple, $I_{new}^\sharp = I^\sharp = [5; 5]$). Si ces deux conditions sont réunies, nous pouvons directement réutiliser l'intervalle O^\sharp préalablement sauvegardé pour la valeur de sortie, éliminant ainsi le besoin de ré-analyser le corps de la fonction. En revanche, si l'une de ces conditions n'est pas satisfaite, une analyse complète du corps de la fonction est effectuée comme d'habitude. Nous verrons plus en détail dans la section 4 les propriétés nécessaires pour l'efficacité et la correction de cette approche. Considérons à présent un autre programme :

```

1 // Main function (Changed)
2 int main() {
3     int factorial = 1;
4     int i = 1;
5     int n = 5;
6     while (i <= n) {
7 -     factorial = factorial * i++;
8 +     factorial *= i;
9 +     i++;
10    }
11 }
```

Dans ce code, la fonction `main` calcule la factorielle de $n = 5$. La modification introduite à l'intérieur du corps de la boucle est une simple réécriture stylistique, qui n'a aucun impact sur la sémantique du programme. Cependant, une analyse statique non incrémentale nécessiterait une nouvelle analyse complète de la fonction `main`, car nous ne pouvons pas inférer syntaxiquement que ces programmes sont équivalents. Ce processus peut être très coûteux en ressource, surtout si le corps de la boucle est complexe ou fait appel à des fonctions elles-mêmes compliquées. Par ailleurs, la technique proposée à l'exemple précédent ne permet pas d'accélérer l'analyse puisque, dans ce cas, le code de la fonction analysée a été directement modifié. Le tableau suivant montre le résultat d'une analyse sur la fonction factorielle originale de notre exemple : l'analyse converge en trois itérations, au prix d'une perte de précision sur le résultat de la fonction pour lequel l'analyse conclut seulement qu'il est strictement positif. Une analyse de la version modifiée du programme, sans analyse incrémentale, aboutit exactement aux mêmes résultats avec le même nombre d'itérations.

itérations	i	fact	n
1	[1, 2]	[1, 1]	[5, 5]
2	[1, +∞[[1, +∞[[5, 5]
3	[1, +∞[[1, +∞[[5, 5]

Pour accélérer l'analyse de la boucle modifiée, on peut néanmoins réutiliser l'invariant calculé lors de l'analyse précédente. Il est à noter que le calcul du point-fixe de la boucle

commence généralement par un état initial \perp . Cependant, nous pouvons également initialiser l'analyse à partir d'un autre état : tant que le test de stabilité $F^\sharp(X^\sharp) \sqsubseteq X^\sharp$ permettant de savoir si le point-fixe est atteint ou non, est réalisé correctement, nous obtiendrons un invariant correct à la fin, quelle que soit la suite des itérés parcourus. Ainsi, nous pouvons initialiser le calcul de point-fixe du programme modifié à partir de l'invariant ($i = [1; +\infty[; fact = [1; +\infty[; n = [5; 5]$) calculé lors de l'analyse précédente. Au mieux, si le nouvel invariant de boucle du programme modifié n'a pas changé, une itération avec le nouveau corps de la boucle à partir de l'ancien invariant suffit à montrer qu'il est toujours un invariant. Au pire, ce n'est pas un invariant, et on continue alors l'itération jusqu'à obtenir une nouvelle valeur stable. Nous obtenons alors les résultats suivants :

itérations	i	fact	n
1	$[1; 2] \sqcup^\sharp [1; +\infty[$	$[1; 2] \sqcup^\sharp [1; +\infty[$	$[5; 5] \sqcup^\sharp [5; 5]$
2	$[1; +\infty[$	$[1; +\infty[$	$[5; 5]$

Ils montrent une convergence en deux itérations, au lieu de trois pour l'analyse non incrémentale, pour un résultat identique. Le test de stabilité est vérifié par $F^\sharp(X_1^\sharp) \sqsubseteq X_1^\sharp$ avec $X_1^\sharp = \{i = [1; 2] \sqcup^\sharp [1; +\infty[; fact = [1; 2] \sqcup^\sharp [1; +\infty[; n = [5; 5] \sqcup^\sharp [5; 5]\}$ et $F^\sharp(X_1^\sharp) = \{i = [1; +\infty[; fact = [1; +\infty[; n = [5; 5]\}$. Même si une telle réduction du nombre d'itérations nécessaires au calcul du point-fixe peut sembler minime sur cet exemple illustratif. Nous verrons que cette approche permet de réduire considérablement le temps nécessaire à une analyse complète sur des exemples plus significatifs, mais peut entraîner une perte de précision. En particulier quand on utilise des stratégies d'itérations plus fines (élargissement amélioré) qui nécessitent un nombre plus grand d'itérations pour maintenir la précision. Dans ce cas, le gain en nombre d'itérations peut être bien plus significatif.

4 Éléments de Frama-C utiles à l'analyse incrémentale

Frama-C est une plateforme open-source, composée d'un ensemble d'analyseurs dédiés à l'analyse de programme C. Elle permet de vérifier des propriétés de sûreté et de sécurité sur des programmes C à l'aide de méthodes formelles. Parmi ses analyseurs, on trouve le greffon *Eva*, fondé sur l'interprétation abstraite. Il s'agit d'une analyse de valeurs permettant notamment d'inférer des invariants portant sur les variables (et, plus généralement, les zones mémoires) du programme. *Eva* permet également à l'utilisateur de sélectionner, configurer et combiner différents domaines abstraits pour adapter l'analyse à des besoins spécifiques. Cette section présente les outils existants dans Frama-C et *Eva* qui ont été sujets à des extensions pour permettre l'analyse incrémentale présentée dans la section 5.

4.1 Memexec : Cache de Résultats d'Analyse de Fonctions

Eva analyse un programme de manière monolithique, en partant du point d'entrée du programme jusqu'aux points de sortie. L'analyse des fonctions est entièrement dépendante des contextes d'appel : à chaque fois qu'une fonction est appelée dans le programme, *Eva* prend en compte l'état du programme au point d'appel pour analyser la fonction. En conséquence, chaque fonction est analysée une fois par pile d'appel. Bien que cette approche dépendante du contexte garantisse une grande précision, elle peut s'avérer très coûteuse en termes de ressources. Pour pallier ce problème, *Eva* utilise un mécanisme appelé *Memexec* [Yak15]. Ce mécanisme agit comme un cache pour sauvegarder les résultats d'analyse d'une fonction. Lorsqu'une même fonction est appelée deux fois dans un contexte identique, *Memexec* permet à l'analyseur de réutiliser les résultats d'analyse précédemment calculés plutôt que d'analyser à nouveau le corps de la fonction. Ce processus de mise en cache et de réutilisation accélère considérablement l'analyse du programme, tout en conservant un degré élevé de précision.

Pour mieux comprendre ce mécanisme, nous allons modéliser son fonctionnement. Soit P un programme, $M : \mathcal{L} \rightarrow \mathbb{V}$ une mémoire, où \mathcal{L} est l'ensemble des emplacements mémoires utilisés par P , \mathcal{D}^\sharp un domaine abstrait et $F : \mathcal{P}(M) \rightarrow \mathcal{P}(M)$ une fonction concrète de P , et

$F^\sharp : \mathcal{D}^\sharp \rightarrow \mathcal{D}^\sharp$ une fonction abstraite qui sur-approxime F . Considérons également deux états abstraits $I^\sharp \in \mathcal{D}^\sharp$ et $O^\sharp \in \mathcal{D}^\sharp$ représentant respectivement l'entrée et la sortie de l'analyse de F , ce qui peut être formalisé par $F^\sharp(I^\sharp) = O^\sharp$. Dans ce cadre, I^\sharp est une sur-approximation de l'ensemble des états concrets possibles avant l'exécution de F , tandis que O^\sharp est une sur-approximation de l'ensemble des états concrets possibles après l'exécution de F .

Hypothèse de base En première approximation, l'analyse d'une fonction dépend uniquement de son état abstrait d'entrée I^\sharp pour produire un état abstrait de sortie O^\sharp .

Analyse de F L'analyse de F est enrichie pour qu'elle calcule également l'ensemble des emplacements mémoires lus $R \subseteq \mathcal{L}$ et écrits $W \subseteq \mathcal{L}$ pendant l'analyse du corps de F pour produire son état abstrait de sortie. Le résultat de l'analyse de F avec I^\sharp comme argument est noté $F_{rw}^\sharp(I^\sharp) = (O^\sharp, R, W)$. Les valeurs R et W sont les emplacements mémoires respectivement lus et écrits calculés lors de l'analyse de F . La projection d'un état abstrait S^\sharp sur les emplacements mémoires L est notée $S^\sharp \downarrow L$, conservant l'information calculée pour L et oubliant les informations sur les autres en y associant la valeur \top . À la fin de l'analyse de F , le quadruplet $(R, W, I^\sharp \downarrow R, O^\sharp \downarrow W)$ est sauvegardé dans un cache appelé Memexec, et est appelé un résumé de F .

Lors d'une analyse ultérieure de F avec un nouvel état abstrait d'entrée I_{new}^\sharp , si cet état abstrait est égal à l'état abstrait d'entrée du cache pour les emplacements mémoires lus R , c'est-à-dire si $I_{new}^\sharp \downarrow R = I^\sharp \downarrow R$, alors on peut réutiliser le résultat d'analyse précédent mis en cache, sans ré-analyser le corps de F . Sinon, on analyse F avec I_{new}^\sharp et on met à jour le cache de Memexec. Lorsqu'on utilise le cache, le résultat est obtenu en combinant l'état abstrait de sortie du cache et le nouvel état d'entrée, projetés respectivement sur les emplacements mémoires écrits et ceux non modifiés, c'est-à-dire, en notant \overline{W} le complémentaire de W :

$$O^\sharp \downarrow W \sqcap^\sharp I_{new}^\sharp \downarrow \overline{W} \quad (1)$$

Exemple Considérons le programme suivant :

```

1 void f(int *p) { *p = 1; }
2 void main() {
3   int a = 0, b = 0;
4   f(&a);
5   f(&a);
6   f(&b);
7 }
```

Le résumé inféré après analyse du premier appel de la fonction f à la ligne 4 est le suivant :

$$(R = \{p\}, W = \{a\}, I^\sharp \downarrow R = \{p \rightarrow \{\&a\}, a \rightarrow \top, b \rightarrow \top\}, \\ O^\sharp \downarrow W = \{p \rightarrow \top, a \rightarrow [1; 1], b \rightarrow \top\})$$

On suppose que ce résumé est sauvegardé dans le cache de Memexec. Lors de l'analyse du second appel de la fonction f à la ligne 5, l'état abstrait d'entrée est $I_{new}^\sharp = \{p \rightarrow \{\&a\}, a \rightarrow [0; 0], b \rightarrow [0; 0]\}$ et la projection de I_{new}^\sharp sur les emplacements mémoires lus R est $I_{new}^\sharp \downarrow R = \{p \rightarrow \{\&a\}, a \rightarrow \top, b \rightarrow \top\}$. On vérifie alors si le nouvel état abstrait d'entrée I_{new}^\sharp et l'état abstrait dans le cache I^\sharp coïncident sur les emplacements mémoires lus $I_{new}^\sharp \downarrow R = I^\sharp \downarrow R$, ce qui est le cas. On peut donc réutiliser le résumé présent dans le cache de Memexec pour calculer le résultat d'analyse de la fonction f au point 5 sans ré-analyser son corps. Le résultat d'analyse est le suivant :

$$\{p \rightarrow \top, a \rightarrow [1; 1], b \rightarrow \top\} \sqcap^\sharp \{p \rightarrow \{\&a\}, a \rightarrow \top, b \rightarrow [0; 0]\} = \\ \{p \rightarrow \{\&a\}, a \rightarrow [1; 1], b \rightarrow [0; 0]\}$$

Enfin, lors de l'analyse du dernier appel à f , à la ligne 6, la condition de réutilisation n'est pas vérifiée $\{p \rightarrow \{\&b\}, a \rightarrow \top, b \rightarrow \top\} \neq \{p \rightarrow \{\&a\}, a \rightarrow \top, b \rightarrow \top\}$. Le corps de la fonction f est donc ré-analysé, puis le nouveau résumé est ajouté dans le cache.

Correction de l'analyse avec utilisation du cache Soit I^\sharp un état abstrait d'entrée d'analyse de F et O^\sharp un état abstrait de sortie de F . Lors de l'analyse de F , en plus de O^\sharp , l'analyseur fournit un ensemble d'emplacements mémoires lus et écrits R et W respectivement, qui vérifient que pour tout état abstrait I_{new}^\sharp , si les emplacements mémoires nécessaires pour analyser F avec I_{new}^\sharp et I^\sharp sont les mêmes (2), alors les états abstraits modifiés par F du résumé O^\sharp doit sur-approximer les états abstraits modifiés par F dans O_{new}^\sharp (3) et que les états abstraits non modifiés par F dans I_{new}^\sharp sur-approximent les états abstraits non modifiés par F dans O_{new}^\sharp (4).

$$I_{new}^\sharp \downarrow R = I^\sharp \downarrow R \Rightarrow \quad (2)$$

$$O_{new}^\sharp \downarrow W \sqsubseteq O^\sharp \downarrow W \quad (3)$$

$$\wedge O_{new}^\sharp \downarrow \overline{W} \sqsubseteq I_{new}^\sharp \downarrow \overline{W} \quad (4)$$

La vérification de ces conditions lors de la première analyse de F avec I^\sharp permet d'ajouter le résumé dans le cache de Memexec. Lors d'une nouvelle analyse avec un nouvel état abstrait S^\sharp , il suffit de vérifier $S^\sharp \downarrow R = I^\sharp \downarrow R$ pour réutiliser le résumé.

Précision de l'analyse Pour un état abstrait S^\sharp , une mémoire M , un ensemble d'emplacements mémoires L et une fonction de concrétisation γ , le résultat de la projection de S^\sharp sur L est une sur-approximation de l'ensemble de tous les états concrets possibles dans $\gamma(S^\sharp)$ qui ont la même valeur que M sur L , c'est-à-dire $\gamma(S^\sharp \downarrow L) \supseteq \{M \mid \exists M' \in \gamma(S^\sharp), M' \downarrow L = M\}$. On en déduit que la réutilisation du cache avec (1) est une sur-approximation du résultat d'analyse sans cache $F^\sharp(I_{new}^\sharp) : \gamma(O^\sharp \downarrow W) \sqcap^\sharp I_{new}^\sharp \downarrow \overline{W} \supseteq \gamma(F^\sharp(I_{new}^\sharp))$. Cela est dû à la perte d'information lors de la projection des états abstraits sur les emplacements mémoire modifiés. En effet, cette opération perd d'éventuelles propriétés relationnelles entre les emplacements modifiés W et les autres. Considérons la fonction `void f() { x = y - 1; }`. Son analyse avec $I^\sharp = \{x < y\}$ produit $O^\sharp = \{x = y - 1\}$ ainsi que le résumé ($R = \{y\}, W = \{x\}, I^\sharp \downarrow R = \{\top\}, O^\sharp \downarrow W = \{\top\}$). Nous constatons que ce résumé ne contient aucune information sur les propriétés relationnelles de x et y , ce qui induit une perte de précision lors de la réutilisation du cache $\gamma(\top \sqcap^\sharp \top) \supseteq \gamma(O^\sharp)$. En pratique, Eva augmente R et W pour prévenir cette imprécision en prenant $R = W = \{x, y\}$. Nous n'avons cependant pas prouvé que le choix d'Eva garantissait ou ne garantissait pas l'absence de perte de précision.

4.2 AST Diff : Correspondance de programmes

Notre approche sur l'analyse statique incrémentale se fonde sur la réutilisation d'une manière sûre des résultats d'analyse précédents. Pour cela, il est nécessaire d'établir une correspondance entre les deux versions du programme. Cette opération est effectuée à l'aide d'AST Diff, un outil d'analyse syntaxique offert par Frama-C et comparant deux arbres de syntaxe abstraite (AST) raisonnablement proches, correspondant typiquement à deux versions successives d'un même programme. Cette comparaison permet de déterminer si une fonction F donnée a été modifiée ou non. Dans le cas où F n'a pas été modifiée, AST Diff fournit une correspondance des variables et des instructions de F . En revanche, si la fonction a été modifiée, nous perdons toute correspondance.

Pour mieux comprendre AST Diff, nous allons modéliser son fonctionnement. Soit AST_{old} et AST_{new} les arbres syntaxiques abstraits de deux versions successives d'un même programme. Pour chaque fonction F_{new} dans AST_{new} , on cherche une fonction F_{old} de même nom dans AST_{old} . L'équivalence entre F_{new} et F_{old} est déterminée en comparant l'équivalence structurelle des deux fonctions modulo renommage des variables locales(6) et en comparant l'égalité (de nom et de valeur) des variables globales qu'elles utilisent(7) et en tenant compte

des fonctions appelées qui doivent être elles-mêmes deux à deux égales(8).

$$F_{new} \equiv F_{old} \Leftrightarrow \quad (5)$$

$$\text{AST}(F_{new}) \equiv \text{AST}(F_{old}) \quad (6)$$

$$\wedge \text{Globals}(F_{new}) = \text{Globals}(F_{old}) \quad (7)$$

$$\wedge \forall f_{new} \in \text{Calls}(F_{new}), \exists f_{old} \in \text{Calls}(F_{old}), f_{new} \equiv f_{old} \quad (8)$$

Si l'équivalence (5) est vérifiée, on considère que F_{new} est une fonction qui n'a pas été modifiée. Dans ce cas, on remplit une table de correspondance entre les noeuds de F_{new} correspondant aux instructions et aux utilisations de variables et ceux similaires dans F_{old} . Sinon, on considère F_{new} comme une fonction qui a été modifiée. On note que deux variables globales ne sont équivalentes que si elles ont strictement le même nom.

À l'état de l'art [DP16, FMB⁺14], les algorithmes de comparaison d'AST calculent un ensemble minimal de modifications nécessaires pour transformer un AST en un autre. Certes, un ensemble vide permet d'inférer que les deux ASTs sont identiques. En revanche, l'application des modifications pour transformer l'AST AST_{old} en AST_{new} ne permet pas de garantir que les résultats d'analyse précédemment liés à AST_{old} soient toujours valides pour AST_{new} . En effet, nous ne cherchons pas à transformer AST_{old} en AST_{new} , nous cherchons à établir une correspondance entre les fonctions dans les deux ASTs. Cela constitue déjà une base solide de garantie de sûreté pour la réutilisation des résultats d'analyse précédents pour les fonctions non modifiées, tout en nous permettant d'identifier déjà les potentielles extensions de cette technique qui pourraient nous être utiles.

5 Approche incrémentale

Cette section présente les approches que nous avons étudiées. Nous nous concentrons sur une réutilisation sûre des résultats d'analyse précédemment sauvegardés. La section 5.1 décrit l'approche consistant à recharger et réutiliser les résumés sauvegardés d'une fonction qui *n'a pas été modifiée*. La section 5.2 propose une extension d'AST Diff pour établir une correspondance entre boucles. La section 5.3 décrit la réutilisation des invariants de boucle.

5.1 Réutilisation des résumés de fonctions

Memexec non incrémental efface ses résultats en fin d'analyse. Nous étendons Memexec pour les sauvegarder. Pour cela, nous procédons en deux étapes : nous rechargeons les résumés de fonctions sauvegardés d'une analyse précédente seulement pour les fonctions qui n'ont pas été modifiées, puis nous réutilisons ces résumés de fonctions pour accélérer l'analyse du programme modifié. Distinguer ces deux étapes est important, car *recharger* le cache d'une fonction est une opération coûteuse qui n'implique pas forcément la *réutilisation* des résumés de fonction dans le cache : il faut donc éviter de la réaliser trop souvent.

Nous détaillons ces étapes dans les paragraphes suivants. Soit un programme P et son cache C , on suppose que le cache est sauvegardé après l'analyse du programme P . Notons \mathcal{F}_P l'ensemble des fonctions du programme P . On appelle un résumé d'une fonction le quadruplet $(R, W, I^\# \downarrow R, O^\# \downarrow W)$. On suppose également que C contient les résumés de toutes les fonctions analysées du programme P . Ainsi, C est formellement défini comme l'ensemble des paires composées chacune d'une fonction $F \in \mathcal{F}_P$ et de l'ensemble des résumés de F que l'on note S_F , c'est-à-dire $C = \{(F, S_F) \mid \forall F \in \mathcal{F}_P\}$.

Conditions de rechargement du cache Soit un programme P_m , correspondant à une version modifiée de P . Pour analyser P_m , nous rechargeons le cache de résumés de fonction C de P . Cependant, si $P_m \neq P$, alors tous les résumés de fonctions dans C ne sont pas forcément valides pour P_m . En effet, supposons que F soit une fonction de P et F_m la version modifiée de F dans P_m . Nous pouvons alors distinguer deux cas. Si $F \equiv F_m$, alors les résumés de F dans C sont valides pour F_m . On peut donc recharger ces derniers. Sinon, les

résumés de F dans C sont potentiellement invalides pour F_m . On évite de les recharger pour garantir la sûreté. On initialise ensuite le cache C_{new} du programme P_m avec les résumés de fonctions rechargeables de C . Enfin, on analyse le programme P_m avec le cache C_{new} .

Conditions de réutilisation du cache et accélération de l'analyse Soit le programme P_m avec le cache C_{new} initialisé à partir des caches rechargeables de C . La condition de réutilisation des résumés de fonction dans C_{new} est la même que la condition d'utilisation du cache dans Memexec non incrémental : il faut que le nouvel état abstrait d'entrée $I_{new}^\#$ coïncide avec un état abstrait $I^\#$ présent dans le cache, mais uniquement sur les emplacements mémoires lus R , c'est-à-dire $I_{new}^\# \downarrow R = I^\# \downarrow R$. Ainsi l'accélération de l'analyse de P_m dépend du nombre de résumés de fonctions réutilisables dans C_{new} .

Exemple Considérons le programme suivant :

```

1     int foo(int *p) { if(*p > 0) *p = 1; else *p = -1; }
2     void main() {
3     -   int a = 0;
4     +   int a = 1;
5         int b = 0;
6         foo(&a);
7         foo(&b);
8     }
```

En analysant le programme non modifié, nous obtenons les résumés suivants pour les deux appels à la fonction `foo` :

$$(\text{foo}, (R = \{p, a\}, W = \{a\}, I^\# \downarrow R = \{p \rightarrow \{\&a\}, a \rightarrow [0; 0], b \rightarrow \top\}, O^\# \downarrow W = \{p \rightarrow \top, a \rightarrow [-1; -1], b \rightarrow \top\})) \quad (9)$$

$$(\text{foo}, (R = \{p, b\}, W = \{b\}, I^\# \downarrow R = \{p \rightarrow \{\&b\}, a \rightarrow \top, b \rightarrow [0; 0]\}, O^\# \downarrow W = \{p \rightarrow \top, a \rightarrow \top, b \rightarrow [-1; -1]\})) \quad (10)$$

Nous omettons le résumé de la fonction `main`, car la modification de la variable `a` invalide le cache du résumé de cette fonction. En analysant le programme modifié, nous rechargeons seulement le cache de la fonction `foo`. Nous distinguons deux cas. Le résumé (9) de la fonction `foo` n'est pas réutilisé à la ligne 6, car l'état abstrait d'entrée $I_{new}^\#$ ne coïncide pas avec l'état abstrait $I^\#$ sur les emplacements mémoires R présent dans le cache, c'est-à-dire $\{p \rightarrow \{\&a\}, a \rightarrow [1; 1], b \rightarrow \top\} \neq \{p \rightarrow \{\&a\}, a \rightarrow [0; 0], b \rightarrow \top\}$. Le résumé (10) de la fonction `foo` est réutilisé à la ligne 7, car l'état abstrait d'entrée $I_{new}^\#$ coïncide avec l'état abstrait $I^\#$ sur les emplacements mémoires R présent dans le cache, c'est-à-dire $\{p \rightarrow \{\&b\}, a \rightarrow \top, b \rightarrow [0; 0]\} = \{p \rightarrow \{\&b\}, a \rightarrow \top, b \rightarrow [0; 0]\}$.

Impact des modifications sur l'efficacité en temps de l'analyse Le bénéfice attendu de cette approche varie selon la nature des modifications apportées au programme. Nous distinguons deux cas de modifications :

1. **Modifications mineures** : ces modifications ne sont pas visibles dans l'AST du programme ou n'invalident pas plusieurs caches de résumés de fonctions.
 - Modifications qui n'impactent pas la sémantique du programme (formatage du code, ajout de commentaires, réorganisation des fonctions, etc.).
 - Petites modifications localisées dans une fonction qui n'est pas profondément imbriquée dans la pile d'appel.
2. **Modifications majeures** : Ces modifications peuvent invalider plusieurs caches de résumés de fonctions.
 - Modifications qui impactent la sémantique du programme (ajout, suppression, modification de variables globales utilisées par plusieurs fonctions).

- Petites modifications localisées dans une fonction qui est profondément imbriquée dans la pile d'appel.

Pour les modifications mineures, nous pouvons recharger un grand nombre de résultats d'analyse dans le cache, ce qui augmente la probabilité de réutiliser les résumés de fonctions et donc d'accélérer l'analyse du programme modifié. Pour les modifications majeures, il est très probable que la plupart des caches seront invalidés directement ou indirectement à cause des modifications dans les fonctions profondément imbriquées dans la pile d'appel qui invalident transitivement le cache de toutes les fonctions appelantes, ce qui augmente considérablement le temps d'analyse du programme modifié.

Cette approche hérite des mêmes limitations que Memexec non incrémental. Memexec ne peut ni sauvegarder ni réutiliser les résultats d'analyse d'une fonction qui alloue de la mémoire dynamiquement. En effet, Eva adopte différentes stratégies d'analyse des allocations dynamique qui peuvent dépendre de plusieurs paramètres, tels que le contexte d'appel et le nombre de bases mémoire déjà allouées pour analyser de telles fonctions. Cela signifie que l'hypothèse de base – l'analyse de la fonction ne dépend que de l'état abstrait d'entrée – n'est plus valide. Cela est également valable pour tout analyseur statique qui analyse une fonction avec d'autres paramètres que l'état d'entrée. Nous ne pouvons donc pas recharger les résultats d'analyse de ces fonctions, car ils n'ont pas été sauvegardés dans l'analyse précédente. Par conséquent, le temps d'analyse incrémental du programme modifié augmente en fonction du nombre de résultats d'analyse de fonctions qui ne peuvent pas être rechargés. Nous verrons dans la section 6 l'impact de cette limitation sur l'efficacité de l'analyse en pratique.

5.2 Extension d'AST Diff

Dans cette section, nous proposons une extension d'AST Diff pour établir une correspondance entre les conditions de boucle de deux fonctions modifiées. Cette condition contrôlant l'accès au corps de la boucle est généralement associé à l'invariant de boucle. Ainsi, établir une correspondance nous permet de recharger l'invariant inféré dans la version précédente du programme d'une fonction modifiée. Cependant, la version actuelle d'AST Diff ne permet pas d'établir une correspondance entre les variables locales et les instructions de deux fonctions modifiées. Nous proposons donc une extension d'AST Diff pour établir une correspondance partielle entre les variables locales et les boucles des deux fonctions modifiées. Pour les variables locales, celles de mêmes noms sont ajoutées à une table. Pour les boucles, il est nécessaire de définir un critère d'équivalence entre les instructions de boucle, car ces dernières peuvent avoir subi une modification. Nous utilisons à cet effet une heuristique fondée sur l'ordre d'apparition des boucles. La correspondance entre les boucles nous permet d'identifier rapidement un invariant de boucle potentiellement réutilisable dans la version actuelle du programme et la correspondance entre les variables locales nous permet d'assurer qu'un invariant réutilisé porte sur les mêmes variables que celui à calculer.

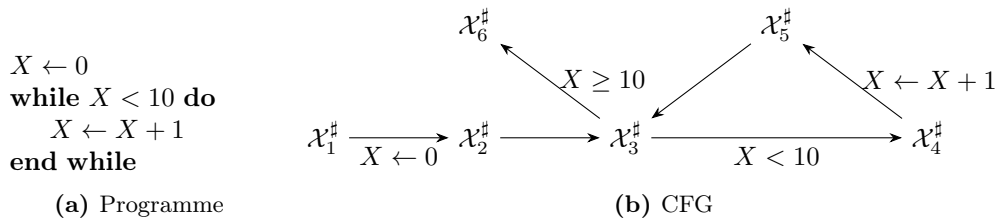
Toutefois, il est important de souligner que cette méthode peut générer de fausses correspondances, notamment si une boucle a été ajoutée ou supprimée. Nous verrons par la suite que ces erreurs n'affectent pas la sûreté de l'analyse. Cependant, elles peuvent tout de même conduire à des faux positifs et donc à une perte de précision. Illustrons cette extension sur l'exemple de la figure 1 qui montre deux versions d'une fonction f contenant deux boucles $L1$ et $L2$ et leurs invariants. Soit $L1_{new}$ et $L2_{new}$ les boucles de la fonction modifiée. Initialement, la correspondance entre les deux versions de f est perdue, car f a été modifiée. Nous cherchons donc à établir une correspondance partielle entre les variables locales et les boucles des deux versions. Ici, nous obtenons $L1 \equiv L1_{new}$ et $L2 \equiv L2_{new}$ grâce à cette heuristique même si, syntaxiquement, $L2 \neq L2_{new}$. Ensuite, la correspondance entre les variables locales garantit que les invariants liés à $L1$ et $L2$ portent sur les mêmes variables que ceux de $L1_{new}$ et $L2_{new}$, même si les valeurs de x diffèrent.


```

1      int f() {
2 -     int x = 0;
3 +     int x = 1;
4       while (x < 10) { // L1, x = [0; 10]
5         x++;
6       }
7       ... // Code
8       int y = 0;
9       while (y < 10) { // L2, y = [0; 10]
10 -      y = y + 1;
11 +      y++;
12     }
13   }

```

Figure 1. Exemple de correspondance de boucle.

Figure 2. Exemple de programme incrémentant X et son CFG.

5.3 Réutilisation des invariants de boucle

Les modifications apportées à une fonction profondément imbriquée dans la pile d'appel ou sur plusieurs fonctions peuvent entraîner l'invalidation de nombreux caches de résumés de fonctions, rendant ainsi l'approche précédente inefficace. Nous souhaitons réutiliser d'autres résultats d'analyse pour accélérer l'analyse incrémentale. Pour cela, nous réutilisons les invariants de boucle précédemment sauvegardés, ce qui est possible grâce à notre extension d'AST Diff. Cette approche permet d'accélérer l'analyse des boucles dans une fonction qui a été modifiée. Plus précisément, au lieu de commencer l'itération de la boucle à partir d'un état initial $\mathcal{X}^\#$, nous la commençons à partir de l'invariant de boucle sauvegardé combiné avec l'état abstrait d'entrée de la boucle. Cela permet d'éviter des itérations abstraites si l'invariant de boucle sauvegardé est proche de l'invariant de boucle du programme actuel.

La figure 2 introduit un programme incrémentant une variable X et le graphe de flot de contrôle (CFG) correspondant dont chaque nœud est étiqueté par l'état abstrait obtenu après analyse. Soit l'invariant $\mathcal{X}_j^{\#i}$, avec i le nombre d'itérations (omis lorsque l'état stable est atteint) et j un point du programme. On note $\mathcal{X}_2^\#$ et $\mathcal{X}_5^\#$ les états abstraits d'entrée et respectivement de sortie de la boucle. L'opérateur d'élargissement standard est défini par :

$$\begin{aligned}
\mathcal{X}_3^{\#0} &= \mathcal{X}_2^\# \\
\mathcal{X}_3^{\#n+1} &= \mathcal{X}_3^{\#n} \nabla^\# (\mathcal{X}_2^\# \sqcup^\# \mathcal{X}_5^{\#n})
\end{aligned} \tag{11}$$

On suppose que l'invariant de boucle $\mathcal{X}_3^{\#old}$ est sauvegardé dans un cache. Nous modifions l'opérateur d'élargissement standard pour réutiliser l'invariant de boucle sauvegardé $\mathcal{X}_3^{\#old}$ combiné avec l'état abstrait d'entrée de la boucle $\mathcal{X}_2^\#$:

$$\begin{aligned}
\mathcal{X}_3^{\#0} &= \mathcal{X}_3^{\#old} \sqcup^\# \mathcal{X}_2^\# \\
\mathcal{X}_3^{\#n+1} &= \mathcal{X}_3^{\#n} \nabla^\# (\mathcal{X}_2^\# \sqcup^\# \mathcal{X}_5^{\#n})
\end{aligned} \tag{12}$$

On rappelle que nous pouvons initialiser l'analyse à partir d'un autre état que $\mathcal{X}_2^\#$. Nous

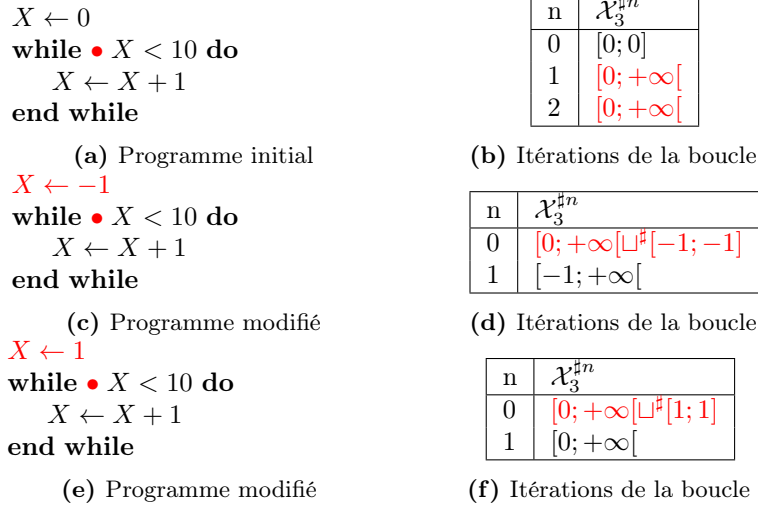


Figure 3. Analyse d'une boucle sans (en haut) et avec (milieu) réutilisation d'invariant, et avec réutilisation mais perte de précision (bas).

obtiendrons un invariant correct à la fin, tant que le test de stabilité $\mathcal{X}_3^{\#n+1} = \mathcal{X}_3^{\#n}$ permettant de savoir si le point-fixe est atteint ou non, est réalisé correctement.

Considérons l'exemple en haut de la figure 3 pour lequel nous souhaitons analyser la boucle avec l'opérateur d'élargissement standard (11). Après $n = 2$ itérations, l'invariant de boucle $\mathcal{X}_3^{\#}$ converge vers $[0; +\infty[$. Cet invariant de boucle est sauvegardé dans un cache pour une réutilisation ultérieure. Considérons à présent l'exemple du milieu, où le programme a été modifié en changeant $X \leftarrow 0$ par $X \leftarrow -1$. Soit $\mathcal{X}_3^{\#old}$ l'invariant de boucle du programme initial. En analysant la boucle avec l'opérateur d'élargissement incrémentale (12), nous commençons l'itération avec $\mathcal{X}_3^{\#old} \sqcup^{\#} \mathcal{X}_2^{\#} = [0, +\infty[\sqcup^{\#}[-1, -1] = [-1; +\infty[$. Ainsi, nous constatons que les itérations convergent en une seule itération, ce qui évite une itération lors du calcul de point-fixe. Cela peut paraître peu, mais en pratique, le temps de calcul induit peut être long, surtout si le corps de la boucle est complexe.

Si l'invariant sauvegardé $\mathcal{X}_3^{\#old}$ est suffisamment proche de l'invariant de la boucle du nouveau programme, alors nous évitons des itérations abstraites et nous convergeons plus rapidement vers l'invariant de la nouvelle boucle. Cependant, dans le cas où cet invariant sauvegardé est différent du nouveau, nous pouvons introduire des faux positifs, et donc une perte de précision. Considérons maintenant l'exemple du bas, où l'on a modifié le programme en changeant $X \leftarrow 0$ par $X \leftarrow 1$. Ici aussi, les itérations convergent en une seule itération. Cependant, l'invariant calculé avec notre opérateur d'élargissement est moins précis que s'il avait été calculé avec l'élargissement standard, avec lequel il serait $[1; +\infty[$. Nous avons donc introduit une imprécision, car le nouvel invariant est plus précis que celui sauvegardé.

Amélioration de la précision L'invariant sauvegardé d'une boucle dans une fonction donnée peut ne pas être unique : il peut varier en fonction du contexte d'appel de la fonction ou même de la trace d'exécution, surtout si des techniques avancées comme le partitionnement de trace [MR05] sont employées. Par conséquent, il est possible et souvent avantageux de sauvegarder plusieurs invariants de boucle pour une seule et même fonction. Ainsi, le choix de l'invariant de boucle à réutiliser a un effet direct sur la précision de l'analyse. Pour identifier l'invariant le plus approprié, il est nécessaire de retrouver le contexte dans lequel la fonction est appelée. Plus précisément, des facteurs tels que les valeurs des arguments d'entrée, la pile d'appel et même les propriétés du chemin d'exécution peuvent influencer le choix de l'invariant de boucle à réutiliser. Nous verrons dans la section 6 comment on retrouve le bon contexte d'appel avec *Eva* pour limiter la perte de précision. On note que, malgré ces améliorations, cette approche d'analyse incrémentale avec réutilisation des invariants

de boucle ne peut pas toujours garantir le même niveau de précision qu’une analyse non incrémentale. Nous verrons néanmoins dans la section 6 que les pertes de précision sont limitées en pratique si les évolutions successives du code le sont aussi.

6 Implémentation et évaluation expérimentale

Implémentation L’implémentation de la recharge des résumés de fonction est relativement simple. En effet, Frama-C permet déjà de sauvegarder les résultats d’analyse dans un fichier dédié. Ainsi, la première étape a été de l’étendre pour y inclure également le cache de Memexec. Finalement, ce dernier est rechargé après la comparaison des arbres syntaxiques abstraits des deux versions du programme, permettant ainsi l’invalidation du cache pour les fonctions qui ont subi une modification.

Pour les invariants de boucle, l’implémentation nécessite une considération particulière du contexte d’appel des fonctions pour le stockage des invariants de boucle. Cela est crucial pour augmenter la précision de l’analyse incrémentale comme expliqué en fin de la section 5.3. Dans ce contexte, nous proposons de sauvegarder les invariants de boucle en se fondant sur la relation fonction \rightarrow pile d’appel \rightarrow arguments \rightarrow partition \rightarrow $\sqcup^{\#}$ invariants de boucle. Ici, le contexte d’appel d’une fonction est représenté par sa pile d’appel, ses arguments et le partitionnement. L’opérateur $\sqcup^{\#}$ fusionne de multiples invariants en présence du même contexte d’appel. Cet opérateur résout le dilemme du choix de l’invariant lorsque plusieurs invariants de boucle coexistent dans un contexte d’appel identique.

Expérimentations Nos expérimentations visent à répondre aux questions de recherche suivantes :

- RQ1** Est-ce que l’analyse incrémentale est plus rapide que l’analyse normale ?
- RQ2** Est-ce que l’analyse incrémentale consomme plus de mémoire que l’analyse normale ?
- RQ3** Est-ce que l’analyse incrémentale est aussi précise que l’analyse normale ?
- RQ4** Est-ce que la réutilisation des invariants de boucle permet d’accélérer l’analyse des boucles dans les fonctions qui ont subi des modifications ?
- RQ5** Quel est l’impact du temps de chargement du cache de Memexec et du calcul d’AST Diff sur l’analyse incrémentale ?

Pour cela, nous avons évalué la performance et la précision de notre approche en nous concentrant sur des critères tels que le temps d’analyse, la consommation mémoire et le nombre d’alarmes générées. Nos expérimentations reposent sur l’analyse de deux programmes : PolarSSL¹ et Monocypher² issus de la suite Open Source Case Studies (OSCS)³. Ces programmes sont chacun analysés successivement sur 10 commits. Seuls les commits incluant des modifications affectant les fichiers sources sont considérés. Il est à noter que ces programmes, écrits en langage C, avaient été préalablement paramétrés et modifiés pour pouvoir être analysés avec Eva. Les modifications consistent en l’ajout d’annotations de code permettant de guider l’analyse afin qu’elle termine en un temps raisonnable.

Pour chaque programme, nous avons d’abord effectué une analyse initiale avec Eva à partir d’un commit initial. Ensuite, nous avons lancé notre analyse incrémentale avec Eva sur les commits suivants. Les tableaux 1a et 1b résumant respectivement les versions des commits analysés et les modifications (nombre de fichiers modifiés, nombre de lignes ajoutées et supprimées par diff) entre les commits pour les programmes PolarSSL et Monocypher. Nos approches ont été implémentées sur le greffon Eva dans une branche publique de la version 26.1 de la plateforme Frama-C. Les codes sources de Frama-C et les scripts utilisés pour les expérimentations sont disponibles ici⁴. Enfin, toutes ces expérimentations ont été menées sur un ordinateur équipé d’un processeur Intel Core i7-12800H cadencé à 2.4 GHz, avec 64 Go de RAM et fonctionnant sous le système d’exploitation Linux Manjaro 6.1.51-1.

1. <https://git.frama-c.com/pub/open-source-case-studies/-/tree/master/polarssl>
2. <https://git.frama-c.com/pub/open-source-case-studies/-/tree/master/monocypher>
3. <https://git.frama-c.com/pub/open-source-case-studies>
4. <https://gitlab.com/nyandrianamamy/jfla2024>

Commit	LoC	Diff	Commit	LoC	Diff
a465d75	28784	1 file/+1	ea79193	27700	36 files/+1937
90f242b	28784	2 files/+3/-2	33ab0fe	27700	1 file/+1/-1
68514b0	28784	6 files/+10/-10	c3c74e2	27710	1 file/+57/-73
3f5b753	28784	2 files/+2/-1	2c6b521	27710	1 file/+7/-7
8648f04	28784	2 files/+6/-1	310aab8	27708	3 files/+30/-33
16e5f81	28794	2 files/+14/-2	57bacd2	27711	1 file/+5
df177ba	28799	2 files/+10	94d0f70	27713	3 files/+33/-30
3081ba1	28801	2 files/+6/-1	f5d90ef	27713	1 file /+6/-6
3513868	28807	2 files/+7	0bfb92	27735	4 files/+146
62dfcf0	28807	6 files/+10/-10	a6fe62b	27759	4 files/+2802

(a) PolarSSL

(b) Monocypher

Table 1. Résumé des modifications. **LoC** indique le nombre de lignes de code du programme analysé et **Diff** indique le nombre de fichiers modifiés, ainsi que le nombre de lignes de code ajoutées et, le cas échéant, supprimées.

Évaluation Soit un ensemble de commits $C = \{c_1, \dots, c_n\}$ et un ensemble de résultats d'analyse $R = \{r_1, \dots, r_n\}$, où c_i est un commit d'un programme P , n est le nombre de commits et r_i est le résultat de l'analyse de c_i . Quatre résultats sont distingués pour les différentes analyses :

- **Normale N** : analyse normale de c_i sans réutilisation de résultats précédents.
- **Incrémentale IO** : analyse incrémentale de c_i avec réutilisation des résumés de fonctions et des invariants de boucle dans r_i .
- **Incrémentale I1** : analyse incrémentale de c_i avec réutilisation des résumés de fonctions dans r_{i-1} .
- **Incrémentale I2** : analyse incrémentale de c_i avec réutilisation des résumés de fonctions et des invariants de boucle dans r_{i-1} .

Nous évaluons l'efficacité de nos approches en comparant les résultats d'analyse incrémentale avec une analyse normale. Pour les programmes PolarSSL et Monocypher, la figure 4 montre le temps d'analyse et la consommation en mémoire pour chacune des quatre configurations. Ensuite, les tableaux 2a et 2b montrent le nombre d'alarmes générées par chacun. Après, les tableaux 3a et 3b montrent le nombre d'itérations total par programme et le nombre d'invariants de boucle réutilisés. Enfin, les tableaux 3c et 3d montrent le nombre de résumés de fonctions rechargés par une analyse **I1** et le taux d'équivalence de fonctions non modifiées inféré par AST Diff entre chaque commit. L'analyse incrémentale **IO** représente le cas où l'on analyse un programme sur lui-même, c'est-à-dire que l'on réutilise les résultats d'une analyse normale pour l'analyse incrémentale du même programme sans modification. Ce cas permet d'obtenir une estimation du temps d'analyse minimal attendu pour une analyse incrémentale, en réutilisant le maximum de résultats sauvegardés. **IO** illustre également l'impact des limitations du cache de Memexec. L'interprétation des résultats est donnée en réponse aux questions de recherche ci-dessous.

RQ1 Pour le programme PolarSSL, nous obtenons en moyenne une analyse incrémentale **IO : 9x**, **I1 : 7x** et **I2 : 8x** plus rapide que l'analyse normale **N**. Avec **IO**, le temps d'analyse est constitué en grande partie par le temps de chargement du cache de Memexec et du calcul AST Diff. Pour le programme Monocypher, nous obtenons en moyenne une analyse incrémentale **IO : 2x**, **I1 : 1.8x** et **I2 : 1.9x** plus rapide que l'analyse normale **N**. Nous constatons avec **IO** que les résultats sont moins bons pour Monocypher, même si le programme n'a pas été modifié. En effet ce programme contient beaucoup de fonctions qui allouent de la mémoire dynamiquement. On rappelle que ces fonctions et les fonctions qui les appellent ne bénéficient pas du cache de Memexec incrémental et non incrémental. Inévitablement, chaque analyse incrémentale doit ré-analyser ces fonctions en entier.

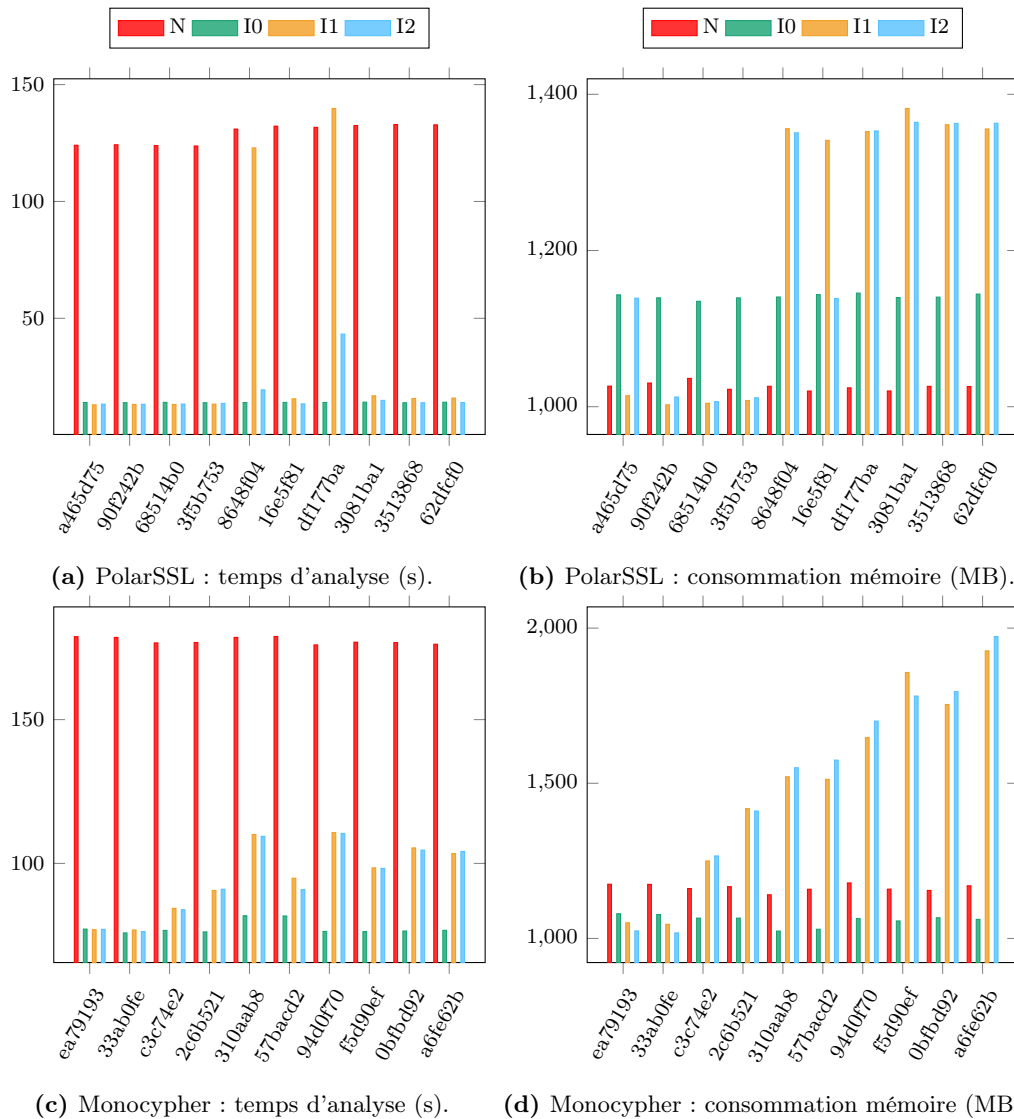


Figure 4. Temps d'analyse et consommation mémoire sur PolarSSL et Monocypher.

RQ2 Pour le programme PolarSSL, nous obtenons une analyse incrémentale qui consomme au maximum pour **I0** : **1.1x**, **I1** : **1.33x** et **I2** : **1.31x** plus de mémoire que l'analyse normale **N**. Pour le programme Monocypher, nous obtenons une analyse incrémentale qui consomme au maximum pour **I0** : **0.91x**, **I1** : **1.63x** et **I2** : **1.67x** plus de mémoire que l'analyse normale **N**. Nous pouvons donc conclure que l'analyse incrémentale consomme plus de mémoire que l'analyse normale. Nous supposons que cette augmentation est due au rapport entre le nombre de résultats rechargés et le taux de réutilisation des résultats. Un taux de réutilisation plus faible implique un nombre plus élevé de création de nouveaux résultats, ce qui augmente la consommation mémoire. Le tableau 3 montre le nombre de résumés de fonctions rechargés.

RQ3 L'analyse incrémentale **I2** des commits `df177ba` et `2c6b521` pour les programmes PolarSSL et Monocypher introduisent chacun une alarme en plus (respectivement 93 contre 92, et 179 contre 178) par rapport à une analyse normale. Nous pouvons donc conclure que l'analyse incrémentale n'est pas aussi précise que l'analyse normale. Cependant, la différence est minime, avec de l'ordre de un pourcent d'alarmes supplémentaires au maximum.

Commit	N	I0	I1	I2	Commit	N	I0	I1	I2
df177ba	92	92	92	93	2c6b521	178	178	178	179

(a) PolarSSL

(b) Monocypher

Table 2. Alarmes générées par les analyses. Seuls les commits qui génèrent plus d’alarmes que l’analyse non incrémentale sont affichés.

Commit	I1	I2	Reuse	Commit	I1	I2	Reuse
a465d75	9	9	0	ea79193	29713	29170	218
90f242b	9	9	0	33ab0fe	29713	29170	218
68514b0	9	9	0	c3c74e2	33042	32167	281
3f5b753	58	21	1	2c6b521	48406	44838	774
8648f04	181133	9170	35	310aab8	46899	45832	488
16e5f81	9	9	0	57bacd2	31179	30488	243
df177ba	200416	46271	477	94d0f70	44927	43796	461
3081ba1	9	9	0	f5d90ef	29713	29170	218
3513868	9	9	0	0bfb92	29713	29170	218
62dfcf9	9	9	0	a6fe62b	31950	31328	57

(a) PolarSSL

(b) Monocypher

Commit	Reloaded	Equiv(%)	Commit	Reloaded	Equiv(%)
a465d75	12898	100.0	ea79193	43537	100
90f242b	12909	100.0	33ab0fe	44027	100
68514b0	12920	100.0	c3c74e2	43567	94
3f5b753	12908	95.0	2c6b521	41802	77
8648f04	12774	90.0	310aab8	42007	89
16e5f81	21202	100.0	57bacd2	44361	96
df177ba	21190	95.0	94d0f70	43436	89
3081ba1	32937	100.0	f5d90ef	46019	100
3513868	32948	100.0	0bfb92	46509	100
62dfcf0	32959	100.0	a6fe62b	46992	97

(c) PolarSSL

(d) Monocypher

Table 3. En haut, nombre d’itérations pour **I1** et **I2** et nombre d’invariants réutilisés par **I2**. En bas, nombre de résumés rechargés par **I1** et taux d’équivalence de fonctions non modifiées inféré par AST Diff entre chaque commit.

RQ4 L’analyse incrémentale **I2** est en moyenne **1.8x** plus rapide que l’analyse incrémentale **I1** de PolarSSL et pas de différence en temps pour Monocypher. Nous constatons en particulier pour les commits **8648f04** et **df177ba** de PolarSSL une différence significative car les modifications apportées à ces commits affectent une fonction profondément imbriquée dans la pile d’appel, ce qui invalide plusieurs caches de résumés de fonctions et rend l’analyse incrémentale **I1** moins rapide. L’analyse incrémentale **I2**, quant à elle, est plus rapide, car elle réutilise les invariants de boucle pour accélérer l’analyse de ces fonctions modifiées.

RQ5 Le temps de chargement maximal du cache de Memexec et le calcul d’AST Diff sont respectivement de **9s** et **6s** pour PolarSSL, et de **32s** et **3s** pour Monocypher. Le chargement des emplacements mémoires lus et écrits par chaque fonction est particulièrement long pour Monocypher en raison des allocations dynamiques de mémoire, augmentant le nombre d’emplacements mémoires et le temps de chargement du cache.

Facteurs de risque pour la validité La validité des résultats de cette étude peut être affectée de plusieurs manières. D’abord, l’analyse n’a été effectuée que sur des modifications qui affectent des fichiers sources entre chaque commit. Par conséquent, l’efficacité de l’analyse pour de grandes différences de versions n’a pas été vérifiée, ce qui limite la généralisabilité des résultats, surtout pour des scénarios où les changements sont importants. Ensuite, les effets des modifications des paramètres d’analyse n’ont pas été explorés. Les résultats obtenus

pourraient donc varier significativement en les modifiant. Enfin, l'étude n'a été réalisée que sur deux programmes, ce qui soulève des questions sur sa généralisabilité. Les caractéristiques spécifiques de ces programmes pourraient avoir influencé les résultats, rendant ainsi leur applicabilité à d'autres programmes incertaine.

7 Conclusion et perspectives

Cet article démontre l'intérêt de l'analyse incrémentale pour optimiser le temps d'analyse et la consommation mémoire. Nous avons proposé deux approches différentes, mais complémentaires pour réutiliser de manière sûre et efficace les résultats d'analyse précédents. La première réutilise les résumés de fonction pour les fonctions qui n'ont pas subi de modifications et n'appellent pas des fonctions qui ont été modifiées. La seconde, permet de réutiliser des invariants de boucle pour accélérer l'analyse des boucles d'une fonction modifiée. Nous avons également présenté une technique de comparaison d'AST pour identifier les fonctions non modifiées, ainsi que la correspondance entre les boucles des fonctions modifiées. Ces approches ont été implémentées sur le greffon *Eva* de la plateforme *Frama-C*. Les résultats expérimentaux montrent que l'analyse incrémentale permet d'améliorer l'efficacité en temps de l'analyse statique tout en garantissant la sûreté et en conservant une grande précision.

Suite à ces travaux, plusieurs axes de recherche sont possibles. D'abord, nous souhaitons formaliser les approches présentées dans cet article afin d'établir une base théorique solide. Cette formalisation permettra de démontrer la correction des méthodes proposées et leur applicabilité dans un contexte plus large.

Ensuite, nous prévoyons d'appliquer la même approche que celle utilisée pour les résumés de fonctions aux résumés de boucles, ou plus généralement à n'importe quel bloc de code. Cette extension devrait permettre d'éviter d'analyser le corps d'une boucle (ou d'un bloc de code) lorsqu'un état d'entrée identique est rencontré. Le choix de la boucle ou du bloc de code à résumer pourra être fait à l'aide d'heuristique choisit automatiquement, ou manuellement par des experts en analyse statique.

De plus, les résultats d'analyse sont actuellement rechargés au début de chaque nouvelle analyse, ce qui peut être coûteux en mémoire et en temps. Pour y remédier, une recharge partielle ou différée des résultats d'analyse du cache est envisagée, où seul le cache de la fonction en cours d'analyse serait rechargé.

Par ailleurs, les analyseurs statiques sont également utilisés après la phase initiale du développement, généralement par une équipe distincte dont l'objectif est de certifier l'absence d'erreur à l'exécution. Pour cela, les paramètres de l'analyseur sont finement ajustés de manière incrémentale afin d'améliorer la précision. Le coût de ce processus étant d'un ordre de grandeur supérieure à celui de la précédente analyse, nous souhaitons adapter notre approche à cette méthodologie reposant sur un paramétrage incrémental de l'analyseur. Toutefois, il est crucial non seulement d'éviter une perte de précision résultant de l'incrémentalité, comme dans le cas actuel, mais également de chercher à améliorer la précision.

Par la suite, nous envisageons d'étendre l'utilisation du cache de résumé de fonction pour les fonctions qui allouent dynamiquement de la mémoire. Cette extension permettra de couvrir un plus grand nombre de fonctions afin d'augmenter la réutilisation du cache.

Enfin, notre analyse incrémentale est actuellement dépendante de l'analyse complète du programme afin d'assurer la sûreté de l'analyse. Nous cherchons à intégrer une approche plus fondamentale fondée sur l'analyse modulaire, inspirée de travaux de P. et R. Cousot [CC02], pour réduire cette dépendance. Cette approche vise à définir une analyse incrémentale tirant parti de la modularité de l'analyse, afin de réduire les parties du programme à ré-analyser.

Remerciements. Ce travail a bénéficié d'une aide de l'État gérée par l'Agence Nationale de la Recherche au titre de France 2030 portant la référence 'ANR-22-PECY-0005'. Nous remercions également les relecteurs anonymes pour leurs remarques bienveillantes ayant permises d'améliorer notre article.

Références

- [AB14] Steven ARZT et Eric BODDEN : Reviser : efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. *In International Conference on Software Engineering (ICSE)*, Hyderabad India, mai 2014.
- [ALL96] Martin ABADI, Butler LAMPSON et Jean-Jacques LÉVY : Analysis and caching of dependencies. *SIGPLAN Not.*, jun 1996.
- [BBB⁺21] Patrick BAUDIN, François BOBOT, David BÜHLER, Loïc CORRENSON, Florent KIRCHNER, Nikolai KOSMATOV, André MARONEZE, Valentin PERRELLE, Virgile PREVOSTO, Julien SIGNOLES et Nicky WILLIAMS : The Dogged Pursuit of Bug-Free C Programs : The Frama-C Software Analysis Platform. *Communications of the ACM*, 2021.
- [BBY17] Sandrine BLAZY, David BÜHLER et Boris YAKOBOWSKI : *In International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 2017.
- [CB16] Maria CHRISTAKIS et Christian BIRD : What developers want and need from program analysis : an empirical study. *In International Conference on Automated Software Engineering (ASE)*, août 2016.
- [CC77] Patrick COUSOT et Radhia COUSOT : Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Symposium on Principles of programming languages (POPL)*, 1977.
- [CC02] Patrick COUSOT et Radhia COUSOT : Modular Static Program Analysis. *In International Conference on Compiler Construction (CC)*. 2002.
- [CDD⁺15] Cristiano CALCAGNO, Dino DISTEFANO, Jeremy DUBREIL, Dominik GABI, Martino LUCA, Peter O’HEARN, Irene PAPAKONSTANTINO et Dulma RODRIGUEZ : Moving Fast with Software Verification. 2015.
- [CDOY11] Cristiano CALCAGNO, Dino DISTEFANO, Peter O’HEARN et Hongseok YANG : Compositional Shape Analysis by means of Bi-Abduction. *Journal of the ACM*, décembre 2011.
- [Cou21] Patrick COUSOT : Principles of Abstract Interpretation. sep 2021.
- [DAL⁺17] Lisa Nguyen Quang DO, Karim ALI, Benjamin LIVSHITS, Eric BODDEN, Justin SMITH et Emerson MURPHY-HILL : Just-in-time static analysis. *In International Symposium on Software Testing and Analysis (ISSTA)*, juillet 2017.
- [DM19] David DELMAS et Antoine MINÉ : Analysis of Software Patches Using Numerical Abstract Interpretation. *In Static Analysis*, volume 11822. Springer International Publishing, Cham, 2019.
- [DP16] Georg DOTZLER et Michael PHILIPPSEN : Move-optimized source code tree differencing. *In International Conference on Automated Software Engineering (ASE)*, 2016.
- [EST⁺22] Julian ERHARD, Simmo SAAN, Sarah TILSCHER, Michael SCHWARZ, Karoline HOLTER, Vesal VOJDANI et Helmut SEIDL : Interactive abstract interpretation : Reanalyzing whole programs for cheap, 2022.
- [FMB⁺14] Jean-Rémy FALLERI, Floréal MORANDAT, Xavier BLANC, Matias MARTINEZ et Martin MONPERRUS : Fine-grained and accurate source code differencing. *In International Conference on Automated Software Engineering (ASE)*, 2014.

- [FT90] John FIELD et Tim TEITELBAUM : Incremental reduction in the lambda calculus. *In Conference on LISP and Functional Programming (LFP)*, mai 1990.
- [JSMHB13] Brittany JOHNSON, Yoonki SONG, Emerson MURPHY-HILL et Robert BOWDIDGE : Why don't software developers use static analysis tools to find bugs? *In International Conference on Software Engineering (ICSE)*, San Francisco, CA, USA, mai 2013.
- [MD15] Antoine MINÉ et David DELMAS : Towards an industrial use of sound static analysis for the verification of concurrent embedded avionics software. *In International Conference on Embedded Software (EMSOFT)*, octobre 2015.
- [MGR13] Scott MCPPEAK, Charles-Henri GROS et Murali Krishna RAMANATHAN : Scalable and incremental software bug detection. *In Joint Meeting on Foundations of Software Engineering (FSE)*, 2013.
- [MOJ18] Antoine MINÉ, Abdelraouf OUADJAOUT et Matthieu JOURNAULT : Design of a Modular Platform for Static Analysis. *In Workshop on Tools for Automatic Program Analysis (TAPAS)*, 2018.
- [MOM20] Raphaël MONAT, Abdelraouf OUADJAOUT et Antoine MINÉ : Static Type Analysis by Abstract Interpretation of Python Programs. *In European Conference on Object-Oriented Programming (ECOOP)*, 2020.
- [MR05] Laurent MAUBORGNE et Xavier RIVAL : Trace partitioning in abstract interpretation based static analyzers. *In Mooly SAGIV, éditeur : Symposium on Programming Languages and Systems (APLAS)*, 2005.
- [NEH19] Lawton NICHOLS, Mehmet EMRE et Ben HARDEKOPF : Fixpoint reuse for incremental JavaScript analysis. *In International Workshop on State Of the Art in Program Analysis (SOAP)*, juin 2019.
- [O'H18] Peter W. O'HEARN : Continuous Reasoning : Scaling the impact of formal methods. *In Symposium on Logic in Computer Science (LICS)*, juillet 2018.
- [Ryd83] Barbara G. RYDER : Incremental data flow analysis. *In Symposium on Principles of programming languages (POPL)*, 1983.
- [SCS21] Benno STEIN, Bor-Yuh Evan CHANG et Manu SRIDHARAN : Demanded abstract interpretation. *In International Conference on Programming Language Design and Implementation (PLDI)*, juin 2021.
- [SEV16] Tamás SZABÓ, Sebastian ERDWEG et Markus VOELTER : IncA : a DSL for the definition of incremental program analyses. *In International Conference on Automated Software Engineering (ASE)*, août 2016.
- [SEV20] Helmut SEIDL, Julian ERHARD et Ralf VOGLER : Incremental Abstract Interpretation. *In From Lambda Calculus to Cybersecurity Through Program Analysis*, volume 12065. 2020.
- [SV21] Helmut SEIDL et Ralf VOGLER : Three improvements to the top-down solver. *Mathematical Structures in Computer Science*, octobre 2021.
- [VdPSVEDR20] Jens Van der PLAS, Quentin STIEVENART, Noah VAN ES et Coen DE ROOVER : Incremental Flow Analysis through Computational Dependency Reification. *In International Working Conference on Source Code Analysis and Manipulation (SCAM)*, septembre 2020.
- [VJL07] Jan Wen VOUNG, Ranjit JHALA et Sorin LERNER : RELAY : static race detection on millions of lines of code. *In Joint Meeting of European Software Engineering Conference and the Symposium on The Foundations of Software Engineering (ESEC-FSE)*, septembre 2007.

- [Yak15] Boris YAKOBOWSKI : Fast whole-program verification using on-the-fly summarization. *In International Workshop on Tools for Automatic Program Analysis (TAPAS)*, 2015.

Alias: pointeurs espionnés en série

Tristan Le Gall², Jan Rochel², Florian Faissole¹, Julien Signoles² et
Denis Cousineau¹

¹Mitsubishi Electric R&D Centre Europe, Rennes, France

²Université Paris-Saclay, CEA, List, Palaiseau, France

Cet article présente un nouveau greffon de Frama-C, appelé *Alias*. Il s'agit d'une analyse de pointeurs légère destinée à être intégrée d'une manière transparente dans d'autres analyseurs. Elle doit donc être à la fois correcte, efficace et ne requérir aucun paramétrage de la part de l'utilisateur. À cette fin, nous avons adapté l'algorithme de Steensgaard pour développer une analyse sensible au contexte à l'aide de résumés de fonction, sensible aux champs de structure, mais insensible aux tableaux et, plus généralement, à l'arithmétique de pointeurs. Nos évaluations expérimentales sur un *benchmark* représentatif montrent qu'elle est bien plus rapide qu'Eva, l'analyse de valeurs de Frama-C préexistante qui fournit aussi une analyse d'alias, tout en gardant une précision suffisante aux besoins.

1 Introduction

En programmation, les pointeurs sont très utiles. Ils dénotent en effet une adresse mémoire du programme, l'« adresse pointée », contenant une autre valeur, la « valeur pointée », ce qui permet d'effectuer simplement des opérations bas-niveau liées à la mémoire, en particulier lorsque cette dernière est allouée dynamiquement pendant l'exécution du programme. On peut les manipuler directement dans des langages permettant des opérations dites de bas-niveau, comme le langage C, ou bien indirectement, sous forme de références dans les langages fonctionnels comme OCaml, ou d'objets dans les langages orientés objets comme Java. L'expressivité ainsi permise a néanmoins un prix : elle rend la compréhension des programmes plus compliquée, que ce soit par un humain ou par un outil automatique. Cette difficulté provient en particulier du problème de l'*aliasing*, c'est-à-dire lorsque deux pointeurs dénotent la même adresse mémoire à un point de programme donné pendant son exécution. Ainsi, dans l'exemple suivant, les pointeurs `p` et `&x` sont aliasés à partir de la ligne 2.

```
int x = 0;
int *p = &x;      // p et &x sont maintenant en alias
x = 1;           // *p est également modifié
printf("%d", *p); // affiche 1
```

Détecter les pointeurs en alias et quelles adresses mémoires sont pointées par quels pointeurs est indécidable dans le cas général, alors même que ce type d'information est nécessaire à la correction de beaucoup de compilateurs ou d'analyseurs de code. Ainsi, dans la plateforme Frama-C dédiée à l'analyse de code C [BBB⁺21], une telle analyse de pointeurs serait utile pour des tâches diverses, comme optimiser du code généré par transformation de programmes ou des obligations de preuve émises par un outil de vérification déductive, ou encore analyser des propriétés sur des programmes concurrents.

Rappelons que Frama-C est composée d'un noyau (qui fournit un parseur du langage C et divers moyens de parcourir et de transformer l'AST obtenu) et de multiples greffons qui sont spécialisés dans un type d'analyse ou de transformation. Frama-C dispose de son langage de spécification formelle : ACSL [BCF⁺]. C'est une plateforme *open-source* et qui permet à tout un chacun de développer de nouveaux greffons, chaque greffon pouvant utiliser les fonctions du noyau ainsi que celles des autres greffons (via leur API).

Frama-C dispose donc déjà d'une analyse d'alias. En effet, son greffon Eva [BBY17] fournit une analyse de valeurs par interprétation abstraite [Cou22, CC77] qui calcule une sur-approximation correcte des valeurs possibles pour chaque expression C en tout point du programme : les adresses mémoires et les pointeurs étant des expressions C particulières, cette analyse est donc, entre autres, une analyse de pointeurs. Elle est très puissante, pouvant notamment passer à l'échelle sur des programmes de grande taille [Our15] tout en donnant des résultats précis. Néanmoins, pour y parvenir, l'utilisateur doit paramétrer finement l'analyseur, ce qui requiert une expertise certaine, afin de régler au mieux les nombreux compromis possibles entre précision d'un côté et rapidité et consommation mémoire de l'autre. Ces paramétrages et cette expertise font que Eva *n'est pas* l'outil adéquat lorsqu'on souhaite *uniquement* s'en servir, dans une autre analyse, pour obtenir les alias et/ou les adresses pointées par un pointeur. En effet, dans un tel contexte, on souhaite en général d'une part une analyse transparente pour l'utilisateur, ne requérant en particulier aucun paramétrage spécifique, et d'autre part, une analyse particulièrement rapide, puisqu'elle est destinée à être englobée dans une analyse plus importante, souvent elle-même coûteuse en temps d'exécution.

Cet article présente un nouveau greffon Frama-C, appelé Alias, qui répond à ce besoin en proposant une nouvelle analyse de pointeurs dite « légère », c'est-à-dire rapide et ne requérant aucun paramétrage utilisateur. Cette nouvelle analyse de pointeurs est principalement destinée à être utilisée par d'autres analyses *via* son API. Elle adapte l'algorithme de Steensgaard [Ste96]. Elle est sensible au contexte (*context sensitive*) pour être raisonnablement précise en présence d'appels de fonction, tout en demeurant efficace grâce à des résumés de fonction. Elle est aussi sensible aux champs des structures (*field sensitive*) pour rester précise lorsqu'on y accède. Elle est en revanche insensible aux tableaux (*array insensitive*) et, plus généralement, aux décalages calculés *via* de l'arithmétique de pointeurs, afin de beaucoup gagner en efficacité tout en ne subissant qu'une perte de précision limitée. L'efficacité en temps et en mémoire, ainsi que la précision, de ce nouveau greffon ont été évaluées sur un *benchmark* existant, déjà régulièrement utilisé pour évaluer Eva.

État de l'art L'analyse de pointeurs est un problème qui a fait l'objet de nombreux travaux de recherche (voir par exemple [SB15]), avec de grandes familles d'algorithmes bien identifiés. La première de ces familles est celle fondée sur l'algorithme d'Andersen [And94]. L'algorithme d'Andersen travaille avec des contraintes sur des sous-ensembles, tandis que Steensgaard [Ste96] utilise des contraintes d'égalité. Par conséquent, l'algorithme d'Andersen est généralement plus précis, mais aussi plus coûteux, que celui de Steensgaard. Quelques années plus tard, [DMM98] a proposé une analyse d'alias reposant sur le typage (statique) d'un programme, et qui n'est donc pas tout à fait adaptée à l'analyse du langage C, du fait des opérations de *casts* permettant des conversions dynamiques de types. Il existe également des analyses de pointeurs "*demand-driven*" [HT01] qui ne cherchent qu'à calculer une partie du graphe de pointeurs [SGSB05], parfois en le formulant en terme d'analyse d'atteignabilité des CFL (*Context-Free Languages*) [ZR08]. Plus récemment, [TLM⁺21] augmente la précision de l'analyse en utilisant la sensibilité au contexte de façon sélective. Parmi les autres approches, il faut également citer les analyses de formes (*shape analyses*) [RS01, JLRS04] utilisant la théorie de l'interprétation abstraite et qui ont pour but de faire une analyse de pointeur très précise [Min06]. Plus généralement, quand on veut faire une analyse statique d'un programme, il est judicieux, dans un certain nombre de cas, de faire une analyse de pointeur précise, car la précision supplémentaire apportée par celle-ci simplifie ensuite l'analyse

```

int* jfla (int *fst, int *snd, int **i1, int **i2, int bo) {
    *snd = *fst;
    if (bo) { fst = *i1; return *i2; } else { fst = *i2; return *i1; }
}
void main(void) {
    int u = 11, v = 12, t[3] = {0,1,2};
    int *a = &t[1], *b = &u, *c = &v;
    int **x = &a, **y = &b, **z = &c;
    struct str_t {int *fst; int *snd; } s = { c , t }, *s1 = &s, *s2 = &s;
    c = jfla(s1->fst, s1->snd, x, y, 0);
    a = jfla(s2->fst, s2->snd, y, z, 1);
}

```

FIGURE 1. Exemple introductif

statique du programme qui est alors plus précise et moins coûteuse que si on s'était contenté d'une analyse de pointeur imprécise [CLHY05]. C'est pourquoi la plupart des travaux récents s'orientent vers ces analyses de pointeurs très précises, malgré la difficulté théorique du problème : [Hor97] a en effet démontré que l'analyse précise de may-alias est NP-Hard (même pour une analyse *flow-insensitive*). Cependant, on peut parfois se contenter d'analyses plus légères. C'est ce que propose par exemple l'outil RTC [MVT⁺16] ; nous avons le même but, mais nous avons implémenté une méthode différente dans Frama-C avec le greffon Alias.

Plan La Section 2 introduit un exemple présentant notre approche. La Section 3 présente l'algorithme de Steensgaard sur lequel repose notre travail. La Section 4 est le cœur de l'article décrivant Alias, la nouvelle analyse d'alias légère de Frama-C. La Section 5 explicite nos résultats expérimentaux. Dans la Section 6, nous discutons quelques applications concrètes envisagées pour cette analyse. Enfin, la Section 7 conclut et propose quelques perspectives.

2 Exemple introductif

Pour illustrer notre méthode, voici un exemple de programme écrit en langage C (Figure 1). Ce programme commence par déclarer dans la fonction `main` deux variables entières `u` et `v`, un tableau d'entier `t`, puis trois pointeurs `a`, `b` et `c` sur ces valeurs. En particulier, `a` pointe vers la seconde case, d'indice 1, du tableau, qui contient initialement la valeur 1. Ensuite, on déclare des pointeurs `x`, `y` et `z` qui pointent vers les adresses auxquelles `a`, `b` et `c` sont stockées en mémoire, ainsi que deux pointeurs `s1` et `s2` vers l'adresse d'une même structure `s`.

Enfin, le programme principal fait appel à la fonction `jfla` à deux reprises. Cette fonction `jfla` prend comme arguments deux pointeurs d'entiers `fst` et `snd`, deux pointeurs de pointeurs `i1` et `i2`, et un booléen codé par un entier `bo`. Elle commence par copier la valeur pointée par `fst` dans la variable pointée par `snd` (copie d'entier), puis, selon la valeur booléenne, copie `*i1` ou `*i2` dans `fst` (copie de pointeur), et renvoie le pointeur non copié.

Cet exemple jouet a été choisi en raison de la complexité de son graphe de pointeurs (Figure 2). Il offre en effet deux niveaux d'indirections, un tableau, une structure et deux appels de fonctions. Surtout, il présente un grand nombre d'alias. Certains sont assez évidents, comme `s1`, `s2` et `&s` qui sont tous les trois aliasés. D'autres sont moins évidents, car ils résultent de l'application de la fonction `jfla`. Par exemple, après l'instruction `c = jfla(s1->fst, s1->snd, x, y, 0)`, les pointeurs `s1->fst` et `y` sont en alias (et pointent vers la variable `u`) ; de même, les pointeurs `*x` et `c` sont aliasés (et pointent vers la case du tableau `t[1]`). Puis le second appel à la fonction `jfla` va encore modifier certains de ces alias. Comme dit en introduction, la recherche exacte de tous les alias à tous les points du programme est donc un problème complexe, qui ne peut être résolu que de manière approchée. Le but de notre analyse sera donc d'obtenir un graphe de pointeurs qui représente une sur-approximation des relations entre les différents pointeurs et permet d'en déduire

ceux qui sont potentiellement aliasés.

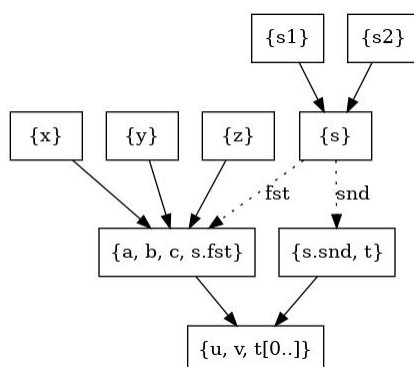


FIGURE 2. Graphe de pointeurs obtenu en fin d'analyse.

3 Une adaptation de l'algorithme de Steensgaard

L'analyse de pointeurs présenté dans cet article est une adaptation de l'algorithme de Steensgaard [Ste96]. C'est l'un des algorithmes les plus connus et les plus efficaces pour réaliser une analyse de *may-alias*, c'est-à-dire une analyse d'alias qui sur-approxime ses résultats, de sorte que l'ensemble concret des alias du programme est toujours inclus dans l'ensemble calculé par l'algorithme : ce critère est essentiel pour la sûreté de l'approche. Cet algorithme construit un graphe de pointeurs, c'est-à-dire un graphe orienté dont les nœuds représentent des pointeurs (ou des valeurs finales) et les arcs du type $a \rightarrow b$ signifient que le pointeur a pointe vers b . Nous rappelons ici brièvement les principes de cet algorithme.

L'article de Steensgaard [Ste96] ne parle qu'implicitement de graphe de pointeurs, car son algorithme est donné en terme d'un problème de typage et d'inférence de types. Pour le greffon Alias, cette formulation ne permettait pas de profiter pleinement des fonctionnalités déjà présentes dans Frama-C, aussi nous avons choisi de reformuler cet algorithme dans une version "analyse flot de données" et de l'implémenter. C'est donc cette version que nous présentons ici.

Une analyse "flot de données" est une analyse statique qui parcourt le graphe de flot de contrôle (CFG) du programme P analysé [NNH10]. Dans notre cas, le CFG est un graphe orienté $(\mathcal{S}, \mathcal{T}, \mathcal{E}, \mathcal{I}, \mathcal{F})$, où :

- à chaque instruction de P on associe deux nœuds de \mathcal{S} , représentant respectivement l'état avant et après l'instruction ;
- $\mathcal{T} \subset (\mathcal{S} \times \mathcal{S})$ est l'ensemble des arcs ;
- $\mathcal{E} : \mathcal{T} \mapsto \{lv := e \mid ?(e) \mid \text{skip} \mid \text{return}(e)\}$ associe une étiquette à chaque arc ;
- $\mathcal{I} \in \mathcal{S}$ représente le nœud initial ;
- $\mathcal{F} \in \mathcal{S}$ représente le nœud final.

Pour les étiquettes des arcs, $lv := e$ représente une affectation dans laquelle e désigne n'importe quelle expression du langage C (y compris des appels de fonctions) et lv est une *valeur-gauche* (*lval*) représentant une zone mémoire dans laquelle on peut écrire (typiquement, une variable). $?(e)$ représente une condition : l'arc n'est franchissable que si la condition booléenne e est vraie, c'est-à-dire est un entier différent de 0 en C, `skip` ne fait aucune action, et `return`(e) représente un retour d'une fonction renvoyant la valeur de l'expression e .

Le CFG d'un programme peut être calculé automatiquement à partir du code source, par exemple en utilisant Frama-C [BBB⁺21]. Frama-C effectue en outre certaines normalisations avant de le calculer, garantissant en particulier que, pour toute fonction f , il n'y a qu'un `return`(`_retres`), systématiquement placé en dernière instruction de f ¹. Ainsi, la Figure 3

1. Dans le cas où la fonction retourne `void`, l'instruction est en réalité `return ;`, mais ce détail est omis dans cet article pour simplifier la présentation.

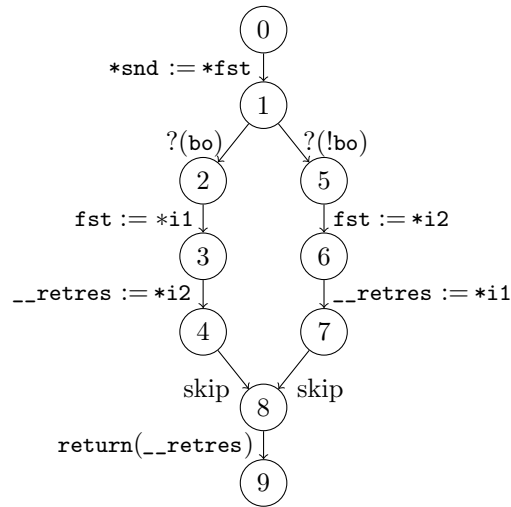


FIGURE 3. CFG de la fonction jfla.

donne le CFG de la fonction `jfla` de l'exemple introductif (cf Section 2).

Une analyse flot de données se caractérise par son sens (avant ou arrière), sa fonction de transfert (comment, à partir d'un état de l'analyseur et d'une transition du CFG, calculer l'état suivant ou précédent) et la manière de fusionner ses états dans le cas où plusieurs arcs arrivent dans un même nœud. Dans le cas de l'algorithme de Steensgaard, les états sont des graphes de pointeurs, c'est-à-dire des graphes non étiquetés ayant les deux propriétés suivantes :

1. Les nœuds du graphe de pointeurs contiennent des ensembles de *lval*. Chaque *lval* apparaît dans au plus un seul nœud. De fait, ces nœuds représentent des *classes d'équivalences* de *lval* potentiellement, du fait des sur-approximations, en alias les unes avec les autres, qui sont progressivement fusionnées.
2. Chaque nœud a au plus un successeur. En effet, l'algorithme détaillé ci-dessous assure que lors des fusions, tous les successeurs potentiels sont fusionnés en un nœud unique, et ce récursivement.

Par conséquent, on définit les fonctions suivantes sur un graphe de pointeurs (G' désigne le graphe modifié à partir de G).

- **find_or_create**(lv, G) = n, G' : renvoie le nœud n de G contenant lv (s'il existe) ; sinon le crée, ce qui donne le graphe G' . Une extension de cette fonction à n'importe quelle expression e du langage C sera détaillée en Section 4.2.
- **fusion_rec**(n_1, n_2, G) = n, G' : fusionne récursivement n_1 et n_2 en un nœud n ; si n_1 et/ou n_2 ont des successeurs, les fusionnent récursivement en un nœud n' (qui sera donc l'unique successeur de n).

Avec ces fonctions, nous pouvons caractériser l'analyse flot de donnée qui implémente l'algorithme de Steensgaard. C'est une analyse en avant dont l'état initial est le graphe vide. La fusion des états est une union des graphes de pointeurs, de façon à toujours sur-approximer. Pour une transition $t \in \mathcal{T}$ donnée, la fonction de transition $F_t(G)$ est calculée de la façon suivante.

- Si $\mathcal{E}(t) = \text{skip}$, alors $F_t(G) = G$ (fonction identité).
- Si $\mathcal{E}(t) = lv := e$, alors $F_t(G)$ est donné par l'Algorithme 1 ; dans cet algorithme, **is_pointer**(lv) vérifie que la *lval* lv est bien un pointeur. En pratique, Frama-C fournit cette information.
- Si $\mathcal{E}(t) = ?(e)$, alors $F_t(G) = G$ sauf si l'expression e est la constante 0, auquel cas $F_t(G) = \perp$, ce qui signifie que la branche du CFG qui suit cette transition est inatteignable.

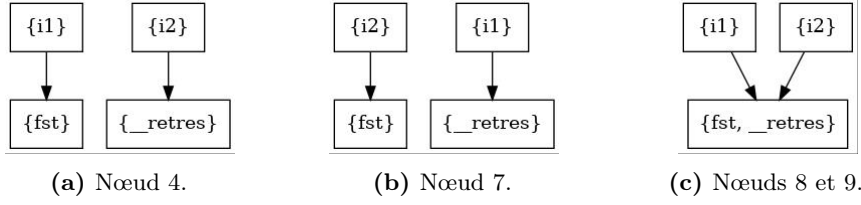


FIGURE 4. Graphes calculés pour différents nœuds du CFG.

- Si $\mathcal{E}(t) = \text{return}(e)$, alors $F_t(G) = G$. En outre, le graphe G est enregistré comme résumé de la fonction (qui sera utilisé lors de l'appel de la fonction, cf Section 4.3).

Algorithme 1 $F_t(G)$ pour une affectation $lv := e$

```

function: fonction_transfert( $lv, e, G$ )
  if is_pointer( $lv$ ) then
    let  $n_1, G := \text{find\_or\_create}(lv, G)$  in
    let  $n_2, G := \text{find\_or\_create}(e, G)$  in
    let  $n, G := \text{fusion\_rec}(n_1, n_2, G)$  in
    return  $G$ 
  else
    return  $G$ 

```

L'Algorithme 1 présente deux simplifications par rapport à l'algorithme original. En effet, pour une affectation scalaire (c'est-à-dire lorsque la variable affectée n'est pas un pointeur), il n'y a pas lieu de procéder à la fusion de nœuds dans le graphe de pointeurs. Cette simplification n'est pas possible dans l'algorithme de Steensgaard original dans la mesure où l'information de scalarité n'est pas directement présente dans l'état de l'analyseur, c'est-à-dire le graphe de pointeurs : on sait uniquement que les nœuds avec successeurs représentent des pointeurs. Ceux sans successeurs représentent potentiellement des scalaires, mais on ne peut en être certains en cours d'analyse, tant que les fusions ne sont pas toutes effectuées. Dans notre cas, Frama-C nous fournit gratuitement, et instantanément, cette information.

L'autre simplification est que l'algorithme original distingue plusieurs types d'affectations, tandis que notre généralisation de la fonction `find_or_create` est applicable à n'importe quelle expression du langage C : la distinction de cas est opérée au moment d'un appel à cette fonction, comme expliqué plus en détail en Section 4.2.

Appliquons donc notre algorithme de Steensgaard modifié à la fonction `jfla` dont le CFG est donné Figure 3. Au nœud 0, le graphe est vide. La première affectation est une affectation scalaire, et ne modifie pas le graphe, tout comme les conditions : le graphe est donc toujours vide aux nœuds 1, 2 et 5. Puis, les deux affectations `fst := *i1` et `__retres := *i2` introduisent de nouveaux nœuds et arcs : la Figure 4a représente le graphe de pointeurs au nœud 4. D'une manière similaire, on obtient la Figure 4b au nœud 7. Enfin, une union des deux graphes est réalisée pour obtenir le graphe final, présenté Figure 4c ; l'algorithme réalisant cet union sera expliqué en Section 4.3.

L'algorithme de Steensgaard a donc de nombreux avantages, ce qui explique son efficacité. D'abord, le graphe est construit de façon paresseuse, un nœud n'y apparaissant que s'il représente un pointeur pour lequel il existe une information d'alias le concernant. Ensuite, il fonctionne par fusion de classes d'équivalence et permet donc d'utiliser des structures de données très efficaces, comme *Union-find* [GF64]. Enfin, l'algorithme assure que chaque nœud a au plus un arc, ce qui permet une exploration linéaire du graphe.

4 Alias, une nouvelle analyse d'alias légère pour Frama-C

Cette section présente le cœur de notre contribution. Nous détaillons d'abord en Section 4.1 le cahier des charges que notre analyse doit satisfaire. Ensuite, la Section 4.2 présente quelques détails sur la façon dont les valeurs gauches et les expressions sont globalement prises en compte. Puis, la Section 4.3 présente la manière dont nous gérons les appels de fonction. Enfin, nous discutons de quelques détails d'implémentation dans la Section 4.4.

4.1 Cahier des charges

Comme nous l'avons dit en introduction, nous voulons implémenter une analyse d'alias « légère » dans Frama-C, d'une manière indépendante à l'analyse de valeur Eva déjà existante de façon, notamment, à être plus efficace sans nécessiter de paramétrages particuliers à spécifier par un expert en analyse de code, tout en gardant une précision raisonnable.

Pour être précis, nous souhaitons avoir une analyse sensible au champ (*field-sensitive analysis*), c'est-à-dire que les champs de structures ne doivent pas être systématiquement fusionnés, de façon à pouvoir spécifier qu'on est en alias avec le champ **a** mais pas avec le champ **b** d'une structure donnée, et sensible au contexte (*context-sensitive analysis*) : le contexte des appels de fonctions doit être pris en compte.

Néanmoins, pour gagner en efficacité, nous sommes prêts à certains sacrifices. En particulier, l'analyse est insensible au tableau (*array-insensitive*) et, plus généralement, à l'arithmétique de pointeurs : deux cases d'un même tableau et, plus généralement, deux pointeurs p et $p \pm i$ ($i \in \mathbb{N}$) sont systématiquement fusionnés dans la même classe d'équivalence.

Enfin, la version courante du prototype ne supportent pas certaines constructions du langage C, dont l'étude est laissée pour des travaux ultérieurs. En particulier, les conversions de types (*cast*) hétérogènes, c'est-à-dire changeant le nombre de déréférencements nécessaires pour atteindre le scalaire final, ne sont, pour l'instant, pas supportées, par exemple celles convertissant un pointeur de type `int*` en `int**` ou encore un scalaire en un pointeur ; pas plus que les fonctions récursives et les types unions. Nous allons maintenant détailler les différents algorithmes que nous avons employés.

4.2 Valeurs gauches et expressions généralisées

Représentation des valeurs gauches L'algorithme de Steensgaard ne considèrerait que deux types d'objets pouvant se trouver à gauche d'une affectation : une variable ($\mathbf{x}=\mathbf{e}$) ou un pointeur déréférencé ($\ast\mathbf{x}=\mathbf{e}$). Néanmoins, le langage C autorise d'autres sortes de valeurs gauches (*lval*). En particulier, une *lval* peut désigner le champ d'une structure ou une case d'un tableau. Plus généralement, une *lval* peut être modélisée par un couple (h, o) composé d'un hôte h (soit le nom d'une variable, soit un emplacement mémoire) et d'un décalage (*offset*) o qui peut être :

- None : pas de décalage supplémentaire ;
- Field(f, o) : accès à un champ f d'une structure, précédé d'un autre décalage o ;
- Index(i, o) : accès à la case d'indice i d'un tableau, précédé d'un autre décalage o .

Cette définition est récursive, car tableaux et structures peuvent être imbriqués. Par exemple, l'expression `t[2].fst` désigne le champ `fst` de la structure se trouvant dans la case d'indice 2 du tableau `t`, ce qui est représenté par le couple hôte-offset $(t, \text{Field}(\text{fst}, \text{Index}(2, \text{None})))$. Cette représentation correspond à celle utilisée par Frama-C.

Généraliser ainsi les valeurs gauches complexifie le graphe de pointeurs et la recherche d'alias. Pour reprendre l'exemple introductif, `s1->fst` et `s2->fst` désignent le même champ `fst` de la structure `s`, puisque `s1` et `s2` sont en alias. Il faut donc ajouter au graphe de pointeurs les informations sur les décalages en plus d'avoir des nœuds pour les hôtes.

Tableaux et arithmétique de pointeurs Pour les décalages de type *Index*(i, o), il suffit d'ajouter un arc supplémentaire au graphe du fait de vouloir une analyse insensible au décalage, car `t[i]` est alors équivalent à `*(t+i)` du point de vue de l'analyse d'alias.

Plus précisément, tout décalage $\tau[i]$ sera considéré comme un décalage vers la première case du tableau $\tau[0]$, et sera donc traité dans le graphe de pointeurs comme un simple arc du nœud contenant τ vers le nœud contenant $\tau[0..]$. Cette dernière notation est inspirée d'ACSL [BCF⁺], le langage de spécification formelle de *Frama-C*. Elle permet d'expliciter le fait qu'un tel nœud représente l'ensemble des cases du tableau τ , dont la taille n'est pas nécessairement statiquement connue en C.

Le fait d'être insensible à l'arithmétique de pointeurs et au décalage lors d'un accès dans un tableau rend l'analyse plus imprécise, mais nous permet de ne pas avoir à dépendre d'une analyse de valeurs sur-approximant ces calculs arithmétiques, ce qui rend l'analyse beaucoup plus efficace.

Structures Pour les décalages de type $Field(f,o)$, que nous souhaitons conserver dans l'état de l'analyse pour être sensible aux structures, on peut soit les coder dans le graphe par des arcs spéciaux $n_1 \xrightarrow{f} n_2$, soit, d'une manière équivalente, les conserver à part, par exemple dans une table modifiée en même temps que le graphe. Nous supposons par la suite la première représentation (ajout d'un arc spécial). En outre, lors d'une fusion de deux nœuds n_1 et n_3 via la fonction $\mathbf{fusion_rec}(n_1, n_3, G)$ introduite en Section 2, et que $n_1 \xrightarrow{f} n_2$ et $n_3 \xrightarrow{f} n_4$, il faut aussi fusionner récursivement n_2 et n_4 pour préserver l'invariant que chaque nœud n'a au plus qu'un seul successeur. Ici, le nœud résultant de la fusion de n_1 et n_3 sera suivi du nœud issu de la fusion de n_2 et n_4 via un arc \xrightarrow{f} .

Fonction `find_or_create` généralisée Comme déjà expliqué, dans l'algorithme de Steensgaard originel [Ste96], chaque nœud du graphe de pointeurs contient un ensemble de variables équivalentes, c'est-à-dire possiblement en alias. La fonction **find_or_create**, présentée en Section 3, a pour but d'associer un nœud du graphe à une expression, ou de créer un tel nœud s'il n'existe pas encore. Les expressions du langage C, même limitées à celles supportées ici, sont nettement plus expressives que celles considérées par Steensgaard. La fonction **find_or_create** doit donc être adaptée. En particulier, l'expression e en argument, est simplifiée de la façon suivante : toute valeur scalaire est ignorée ou traitée comme zéro ; toute arithmétique de pointeurs est simplifiée pour ne retenir que le pointeur de base : par exemple, si p est un pointeur, l'expression $*(p+4)$ sera simplifiée en $*(p+0) = *p$; toute conversion de type (`cast`) est considérée comme idempotente, ce qui est possible tant que les conversions hétérogènes ne sont pas supportées. Cette procédure de simplification, notée **simplify**(e), permet de garantir qu'une expression est soit une *lval* $Lv(lv)$, soit une adresse $Addr(lv)$ d'une *lval* lv . Par conséquent, **find_or_create_expr**(e, G) doit chercher récursivement le nœud correspondant à **simplify**(e) dans G , et le crée s'il n'existe pas encore. L'Algorithme 2 présente cette recherche sous la forme d'un ensemble de fonctions mutuellement récursives, du fait de la structure inductive de notre représentation des valeurs gauches. Il fait appel aux fonctions auxiliaires suivantes :

- **find_or_create_var**(v, G) : trouve le nœud correspondant à la variable v (et ajoute la variable v à l'ensemble des *lval* de ce nœud), ou le crée s'il n'existe pas encore ;
- **add_edge**(n, n', G) ajoute un arc de n à n' ;
- **add_field**(n, f, n', G) ajoute l'arc spécial $n \xrightarrow{f} n'$ à G .

4.3 Inter-procédurabilité

Jusqu'à présent, nous avons parlé d'une analyse de pointeurs intra-procédurale, c'est-à-dire à l'intérieur d'une même fonction. La gestion de l'inter-procéduralité est souvent technique en analyse de code et cette section explique notre façon de procéder pour *Alias*.

Le corps de la fonction est analysé comme précédemment expliqué. Lorsque l'on atteint l'instruction `return(_retres)`, le graphe de pointeurs obtenu est transformé en un *résumé* pour cette fonction. Pour cela, nous filtrons du graphe toutes les variables locales pour ne

Algorithme 2 Les différentes fonctions `find_or_create`

```

function: find_or_create_expr( $e, G$ )
  match simplify( $e$ ) :
    | Lv( $lv$ ) : return find_or_create_lv( $lv, G$ )
    | Addr( $lv$ ) :
      let  $n_2, G :=$  find_or_create_lv( $lv, G$ ) in
      let  $n_1 :=$  new_node() in
      return  $n_1, \text{add\_edge}(n_1, n_2, G)$ 

function: find_or_create_lv( $lv, G$ )
  let  $h, o := lv$  in
  let  $n, G :=$  find_or_create_host( $h, G$ ) in
  return find_or_create_offset( $n, o, G$ )

function: find_or_create_host( $h, G$ )
  match  $h$  :
    | Var( $v$ ) : return find_or_create_var( $v, G$ )
    | Mem( $e$ ) :
      let  $n_1, G :=$  find_or_create_expr( $e, G$ ) in
      if  $\exists(n_1 \rightarrow n_2) \in G$  then return  $n_2, G$ 
      else
        let  $n_2 :=$  new_node() in
        return  $n_2, \text{add\_edge}(n_1, n_2, G)$ 

function: find_or_create_offset( $n, o, G$ )
  match  $o$  :
    | None : return  $n, G$ 
    | Field( $f, o'$ ) :
      if  $\exists(n \xrightarrow{f} n') \in G$  then return find_or_create_offset( $n', o', G$ )
      else
        let  $n' :=$  new_node() in
        return find_or_create_offset( $n', o', \text{add\_field}(n, f, n', G)$ )
    | Index( $0, o'$ ) :
      if  $\exists(n \rightarrow n') \in G$  then return find_or_create_offset( $n', o', G$ )
      else
        let  $n' :=$  new_node() in
        return find_or_create_offset( $n', o', \text{add\_edge}(n, n', G)$ )

```

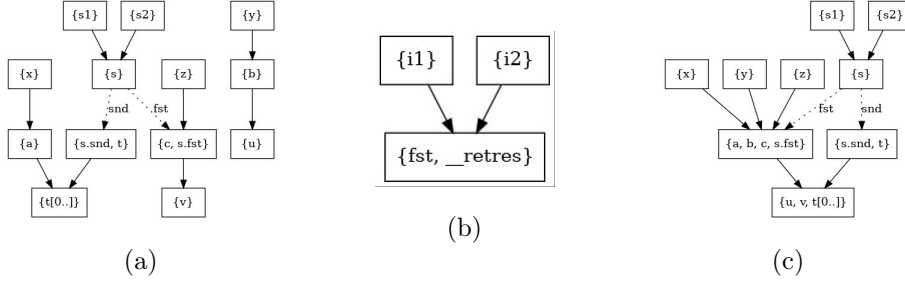


FIGURE 5. Graphe de pointeurs avant (a) et après (c) l'appel à la fonction `jfla` de résumé (b).

conserver que les variables globales et ses paramètres formels, ainsi que la variable distinguée `__retres`.

Lors de l'appel d'une fonction, le résumé est inséré dans le graphe de la fonction appelée en unifiant chaque paramètre formel avec l'argument réel correspondant. De même, le nœud correspondant à la (classe d'équivalence de la) variable `__retres` est unifié avec le nœud de la `lval` affectée lorsqu'elle existe. Les nœuds contenant les variables globales sont eux aussi unifiés avec ceux déjà existants dans le graphe de la fonction appelée.

Plus précisément, pour une fonction g avec un graphe G , appelant une fonction f :

- la normalisation du code par Frama-C assure que chaque appel de la fonction f est fait par une instruction spéciale `Call(r,f,args)` où r est la `lval` optionnelle dans laquelle est stocké le résultat de f et $args$ est la liste des arguments de la fonction f ;
- s'il n'existe pas encore un résumé R pour la fonction f , le corps de cette fonction est alors analysé s'il existe et ce résumé est créé ;
- pour chaque argument arg_i de la liste $args$, on identifie le nœud correspondant dans G grâce à la fonction `find_or_create_expr`, et on le fusionne avec celui correspondant à l'argument formel dans R ;
- on procède de même pour la `lval` r lorsqu'elle existe et le nœud correspondant à la variable `__retres` dans R .

La principale différence avec la fusion récursive `fusion_rec` employée dans l'Algorithme 1 est que ces fusions sont effectuées simultanément et qu'elles peuvent interférer entre elles, certains nœuds pouvant être partagés. En outre, les fusions faites lors d'un appel de fonction sont elles aussi récursives pour préserver l'invariant d'au plus un seul arc sortant pour chaque nœud. Ainsi, le premier appel à la fonction `jfla` donne lieu à la fusion des graphes telle que le montre la Figure 5.

4.4 Implémentation

Comme tous les autres greffons de Frama-C, `Alias` est écrit en langage OCaml. Il utilise les fonctionnalités offertes par le noyau de Frama-C. En particulier, il utilise le module `Dataflow` qui implémente une analyse "flot de données" générique, instantiable grâce à un foncteur. Ceci permet de se concentrer sur la définition des éléments propres à notre analyse, notamment son état, c'est-à-dire ici le graphe de pointeurs, et les opérations associées, comme l'union de deux graphes, ainsi que les fonctions de transfert. L'implémentation précise de ses éléments sur un graphe de flot de contrôle, comme par exemple le calcul de point-fixe en présence de cycle, est laissé à la charge du moteur de `Dataflow`.

`Alias` peut être appelé par d'autres greffons Frama-C grâce à son API. Cette dernière offre à l'utilisateur diverses fonctions et itérateurs, dont :

- `fold_aliases_stmt(f, acc, kf, s, lv)` itère la fonction OCaml f à partir de l'accumulateur acc sur les alias de la `lval` lv avant l'instruction s de la fonction $C kf$
- `fold_new_aliases_stmt(f, acc, kf, s, lv)` itère la fonction f à partir de l'accumulateur acc sur les alias de la `lval` lv après l'instruction s de la fonction $C kf$

- **fold_aliases_kf**(f, acc, kf, lv) itère la fonction f à partir de l’accumulateur acc sur les alias de la *lval* lv juste avant l’instruction `return(__retres)` de la fonction $C\ kf$
- **are_aliases_stmt**(kf, s, lv_1, lv_2) renvoie vrai si et seulement si les deux *lval* lv_1 et lv_2 sont en alias avant l’instruction s de la fonction $C\ kf$
- **fold_vertex_closure**(f, acc, kf, s, lv) itère récursivement f à partir de l’accumulateur acc sur toutes les *lval* contenues dans le nœud i et, récursivement, dans tous ses successeurs, du graphe de pointeurs avant l’instruction s de la fonction $C\ kf$
- **get_state_before_stmt**(kf, s) offre un accès direct à l’état de l’analyse, c’est-à-dire à la totalité du graphe de pointeurs et des informations associées, avant l’instruction s de la fonction $C\ kf$. Cette fonctionnalité permet aux développeurs audacieux de coder des opérations complexes de bas niveau, sans exposer dans l’API des opérations inutiles pour la quasi-totalité des développeurs.

L’analyse est faite une seule fois : les résultats sont stockés de manière persistente et peuvent ensuite être utilisés via l’API autant de fois que nécessaire. Cette fonctionnalité est classique dans *Frama-C* et offre un confort appréciable. Elle contraint néanmoins certains choix d’implémentation. Nous avons implémenté la structure de graphe de pointeurs grâce à la bibliothèque **OCamlGraph** [CFS05, CFS07]. Cette bibliothèque offre à la fois la possibilité des structures de graphes persistantes ou non-persistantes. Nous avons choisi les premières car cela nous permet de conserver un graphe en chaque point du programme, ce qui est requis par l’API ci-dessus, sans avoir besoin de copier le graphe, en temps linéaire, à chaque instruction le modifiant. Parmi les autres choix effectués, nous avons choisi de continuer l’analyse en émettant un avertissement lorsqu’une construction non supportée est rencontrée. Cela permet de passer à l’échelle, au prix d’une perte potentielle de correction, car le comportement par défaut choisi peut ignorer un certain nombre d’alias. Par exemple, lorsqu’une fonction est déclarée mais pas définie, nous supposons implicitement que cet appel ne modifie pas le graphe de pointeurs. À terme, nous pourrions nous servir des contrats ACSL pour raffiner ce comportement, mais cela ne garantirait pas pour autant la correction. C’est également le cas du code assembleur. Aussi, les conversions de type (*cast*) hétérogènes sont ignorées.

Pour finir, nous signalons qu’Alias sera intégré à la prochaine distribution publique de *Frama-C* (version *Frama-C* 28.0(Nikel)), mais est d’ores et déjà accessible à partir du répertoire de développement public de *Frama-C*². Il est distribué sous licence libre LGPLv2.1.

5 Évaluation expérimentale

Comme le greffon *Alias* a été conçu comme une alternative à *Eva*, nous évaluons sa performance et son exactitude en le comparant à *Eva*. À cette fin, nous avons effectué des expérimentations sur une collection d’exemples, nommée *Open source case studies*³ (OSCS). Cette collection contient 36 projets open-source de tailles diverses mais comprenant au total environ 270 000 lignes de code C, déjà préparé pour pouvoir être facilement traités avec *Frama-C* en général et *Eva* en particulier.

5.1 Performance

Pour chaque projet dans OSCS nous avons mesuré le temps d’exécution et le pic de consommation de mémoire en utilisant trois modes opératoires différents : d’abord, avec le greffon *Eva*, donnant le temps d’exécution t_e et la mémoire consommée m_e ; ensuite, avec le greffon *Alias*, donnant t_a et m_a ; enfin, sans aucun greffon, donnant t_k et m_k . Ce dernier mode nous permet de déterminer le temps et la mémoire consommé par le noyau de *Frama-C* pour calculer son AST. Ce prétraitement est une précondition à toutes analyses, notamment *Eva* et *Alias*. Ce temps et la mémoire nécessaire sont donc inclus dans les mesures ci-dessus.

2. <https://git.frama-c.com/pub/frama-c/-/tree/master/src/plugins/alias>

3. <https://git.frama-c.com/pub/open-source-case-studies>

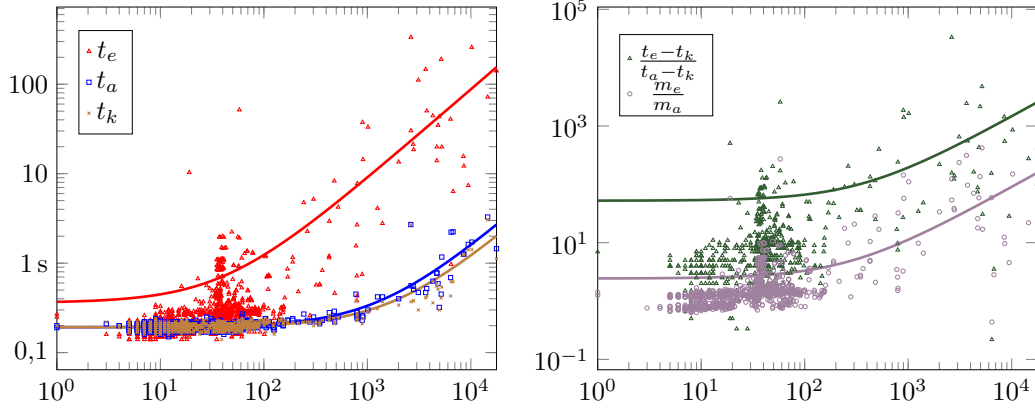


FIGURE 6. Indicateurs de performance en fonction du nombre de lignes de code des sous-projets ; à gauche : temps d'exécution des différentes analyses (noyau en marron, Alias en bleu, Eva en rouge) ; à droite : l'accélération d'Alias par rapport à Eva (en vert) et la réduction de l'empreinte mémoire (en violet).

Comme t_e et t_a incluent tous les deux le temps de prétraitement t_k par le noyau de Frama-C, nous soustrayons t_k pour obtenir le seul temps d'analyse, sans prétraitement. Ainsi, nous pouvons calculer le facteur d'accélération d'Alias par rapport à Eva avec la formule $\frac{t_e - t_k}{t_a - t_k}$. Pour l'empreinte mémoire, ce genre de soustraction n'est pas pertinent. En effet, nous ne calculons qu'un *pic* de consommation de mémoire, sans savoir où il se situe précisément. En outre, il est très probable qu'il soit dans les analyseurs et non dans le noyau⁴. Pour le *facteur de réduction de l'empreinte mémoire* d'Alias par rapport à Eva, nous nous contentons donc de l'approximation $\frac{m_e}{m_a}$.

Les résultats sur OSCS sont présentés dans les figures 6 et 7, en fonction du nombre de lignes de code. Notez que quelque projets d'OSCS sont découpés en sous-projets car ils s'agit en réalité de *benchmarks* incluant eux-mêmes plusieurs programmes, donnant un nombre de points de données bien supérieur à 36. L'ordinateur avec lequel le test de performance a été effectué a comme processeur un Intel Core i7-8700 avec 3.20GHz et 16 Go de RAM.

En totalisant toutes les valeurs pour tous les projets on obtient une accélération moyenne de 256 et en se limitant aux sous-projets de plus de mille lignes de code 235 :

$$\frac{\sum t_e - \sum t_k}{\sum t_a - \sum t_k} \approx 256 \qquad \frac{\sum_{>999} t_e - \sum_{>999} t_k}{\sum_{>999} t_a - \sum_{>999} t_k} \approx 235$$

Comme ce n'est pas pertinent d'additionner des pics de consommation de mémoire nous considérons seulement la moyenne de la réduction d'empreinte de mémoire, qui est d'environ 1,5. En se limitant aux sous-projets de plus de mille lignes de code, la moyenne est de 3,6 :

$$\text{moyenne}\left(\frac{m_e}{m_a}\right) \approx 1,5 \qquad \text{moyenne}_{>999}\left(\frac{m_e}{m_a}\right) \approx 3,6$$

5.2 Exactitude et complétude

Nous nous intéressons à la question de savoir à quel point l'ensemble d'alias A calculé par Alias est une bonne approximation du *vrai ensemble d'alias* V . Comme l'algorithme de Steensgaard calcule une sur-approximation de V , on pourrait définir l'*exactitude* d'Alias comme $\frac{|V|}{|A|}$: quel taux d'alias calculés par Alias sont de vrais alias ?

Néanmoins, comme l'implémentation actuelle d'Alias est un prototype ne traitant pas encore certains constructions syntaxiques, il peut également manquer de vrais alias dans A . Cela fausse la pertinence de $\frac{|V|}{|A|}$, car alors, plus le nombre d'éléments manquants serait

4. Dans le cas contraire, on aurait $m_a \approx m_k$ et nous obtiendrions des valeurs démesurées pour $\frac{m_e - m_k}{m_a - m_k}$.

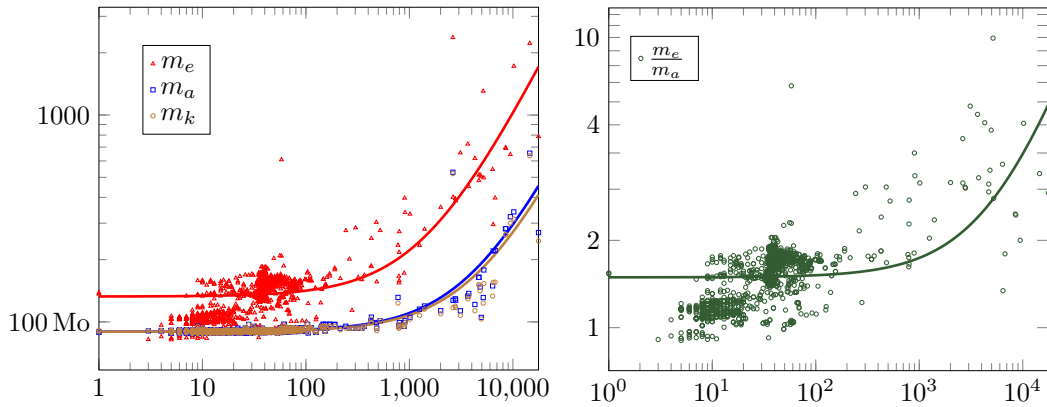


FIGURE 7. Indicateurs de performance en fonction du nombre de lignes de code des sous-projets ; à gauche : pics de consommation de mémoire en Mo ; à droite : facteurs d'épargne de mémoire d'Alias par rapport à Eva

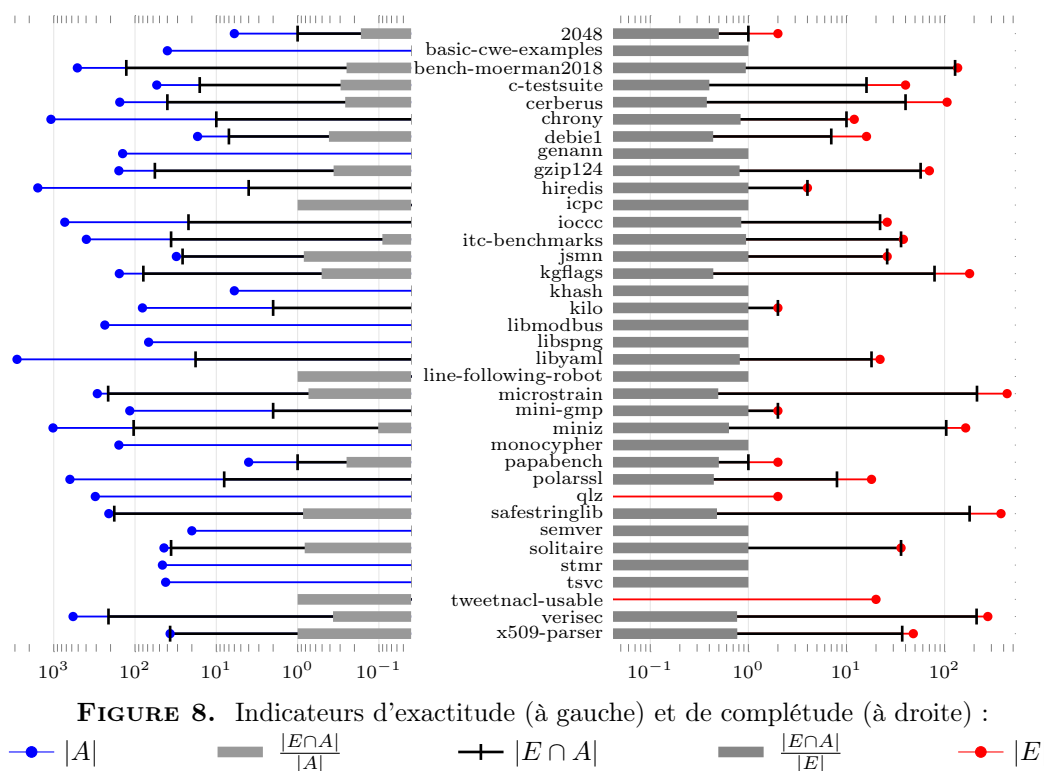
grand, plus l'exactitude serait élevée. Nous pouvons restaurer une notion d'exactitude viable en prenant en compte les alias manquants comme suit : $\frac{|V \cap A|}{|A|}$. Toutefois, dans le cadre de cette expérimentation, l'ensemble V est inconnu. En particulier, les projets analysés sont trop gros pour pouvoir le calculer « à la main ». Néanmoins, nous pouvons utiliser Eva pour connaître l'ensemble des alias E calculé par Eva. Comme $E \supseteq V$, nous pouvons utiliser comme approximation (et borne supérieure) pour $\frac{|V \cap A|}{|A|}$ la formule $\frac{|E \cap A|}{|A|}$. De manière analogue, on peut définir la *complétude* d'Alias comme $\frac{|V \cap A|}{|V|}$: combien de vrais alias ont été trouvés par Alias ? De nouveau nous pouvons approximer V par E en calculant $\frac{|E \cap A|}{|E|}$.

Notez quelques choix qui ont été faits pour cette évaluation expérimentale. D'une part, elle repose sur une analyse d'alias et non une analyse de pointeurs, car cette expérimentation était nettement plus facile à réaliser avec Eva. D'autre part, les alias sont comparés seulement à la fin de chaque fonction, et non pour chaque instruction. Cette évaluation a nécessité le développement d'un greffon Frama-C dédié calculant les ensembles A et E pour toutes les *lval* du programme, à la fin de chaque fonction. La figure 8 présente les résultats d'exactitude et de complétude d'Alias pour tous les projets d'OSCS. En totalisant toutes les valeurs sur tous les projets, on obtient les résultats globaux suivants :

$$|E| = 2058 \quad |A| = 11774 \quad |E \cap A| = 1240 \quad \frac{|E \cap A|}{|A|} \approx 0,11 \quad \frac{|E \cap A|}{|E|} \approx 0,6$$

$$\text{moyenne} \left(\frac{|E \cap A|}{|A|} \right) \approx 0,29 \quad \text{moyenne} \left(\frac{|E \cap A|}{|E|} \right) \approx 0,77$$

Limitations de l'expérimentation Les résultats relativement faibles pour l'exactitude et la complétude s'expliquent par de nombreuses raisons. D'abord, on ne dispose pas des vrais ensembles d'alias pour les exemples traités. L'utilisation d'Eva comme cadre de référence mène à des chiffres imprécis pour l'exactitude et la complétude. Ensuite, grâce à la structure de certains exemples d'OSCS, Eva peut utiliser des contrats attachés aux fonctions des libraries standard pour son analyse abstraite et, certaines de ces fonctions sont même traitées de manières natives, comme des *built-ins*, ce qui améliore grandement leur précision (et leur efficacité). Un tel traitement n'existe pas à ce jour pour Alias. Puis, Alias fonctionne très mal sur certains exemples spécifiques, en général à cause d'une construction spécifique non encore supportée, ce qui fausse fortement le calcul des moyennes. Enfin, certaines causes peuvent être encore inconnues : pour mieux les comprendre, il faudrait scruter plus en détails les résultats exacts d'Eva et Alias sur nos exemples. Cette étude n'a pas encore été effectuée.



6 Applications potentielles

Cette section présente quatre applications potentielles d'Alias. En effet, même si ce nouveau greffon a été développé très récemment, plusieurs applications potentielles sont prévues ou en cours. Cette section en présente quatre, deux pour de nouveaux analyseurs et deux pour des analyseurs existants : d'une part, les Sections 6.1 et 6.2 introduisent respectivement des applications à la génération de programmes concurrents et à une analyse pour l'intégrité du flot de contrôle, tandis que les Sections 6.3 et 6.4 montrent l'intérêt d'Alias pour optimiser d'une manière correcte des traitements dans WP, le greffon de vérification déductive de Frama-C, et dans E-ACSL, celui dédié à la vérification à l'exécution. La première application, pour la concurrence, est plus détaillée que les autres.

6.1 Génération automatique de programmes concurrents

Les programmes concurrents sont, par nature, plus complexes que les programmes séquentiels et, donc, sont davantage sujets aux erreurs. Certaines classes d'erreurs leur sont propres, comme les *data races*, les *deadlocks* ou les violations de comportements attendus par l'utilisateur, *e.g.* l'atomicité d'une section de code, c'est-à-dire le fait que d'autres instructions peuvent s'exécuter en parallèle sans que leurs effets n'impactent ceux de la section considérée, ou l'ordonnancement entre instructions de tâches parallèles. Vérifier l'absence d'erreur s'avère particulièrement difficile dans ce contexte. Non seulement il est nécessaire de raisonner suffisamment finement sur le modèle mémoire sous-jacent, mais il faut aussi tenir compte de tous les entrelacements possibles entre les tâches du programme : la multitude de combinaisons possibles rend inefficace tout effort de validation à base de tests.

Certains interpréteurs abstraits, comme le greffon MThread [KKP⁺15] de Frama-C ou AstréeA [Min15] permettent de vérifier l'absence de *data races* et de *deadlocks* dans les programmes C concurrents. Ces approches, fondées sur une abstraction des interférences entre tâches d'un programme, présentent plusieurs inconvénients : difficulté à corriger les erreurs identifiées, impossibilité de traiter les erreurs complexes (violations d'atomicité ou

```

1 int *out, in[10], x;
2 void * t1(void * args) {
3     int *tmp1 = &x;
4     for (int i = 0; i < 10; i+=2) *tmp1+=in[i];
5 }
6 void * t2(void * args) {
7     int *tmp2 = &x;
8     for (int i = 1; i < 10; i+=2) *tmp2+=in[i];
9 }
10 void main(void) {
11     out = &x;
12     *out = 0;
13     _task_(t1);
14     _task_(t2);
15     printf ("Current output = %d\n", *out);
16     _after_task_(t1,t2);
17 }

```

FIGURE 9. Exemple de programme en entrée

d'ordonnement) et coût important de l'analyse. Les langages corrects par construction, comme Rust [YSZ19] ou Concurrent Cyclone [GPS10], garantissent par typage l'absence de *data races*. Cependant, les erreurs plus complexes ne leur sont pas accessibles. Ils sont par ailleurs difficiles à adopter dans certains contextes industriels, de part leur difficulté d'appréhension et le nombre important de programmes existants écrits dans des langages traditionnels. Autolocker [MZGB06] offre un compromis intéressant : l'outil est fondé sur un langage proche du C, avec des directives de parallélisme de haut niveau. Les programmes y sont écrits en C séquentiel et des directives permettent de déclarer le lancement de tâches ou l'atomicité de sections de code. L'outil place des verrous dans le code de manière à assurer l'atomicité et l'absence de *deadlocks*. En revanche, il est de la responsabilité de l'utilisateur de déclarer quels verrous protègent quelles variables. De plus, Autolocker ne permet pas de spécifier de propriétés d'ordonnement temporelles entre instructions.

Nous proposons une approche plus complète, fondée sur un langage d'entrée proche d'Autolocker. Le langage d'entrée est augmenté de la possibilité de placer des directives d'ordonnement entre des instructions à l'intérieur des tâches parallèles. La Figure 9 présente ce langage d'entrée à travers un exemple. Le programme a pour but de sommer les éléments d'un tableau. Les deux tâches `t1` et `t2` sont respectivement chargées des éléments d'indices pairs et impairs. Nous complexifions le cas d'usage en introduisant des alias entre pointeurs. La directive `_task_(t)` spécifie le lancement d'une tâche `t`. La directive `_after_task_(t)` spécifie un point d'ordonnement après lequel les instructions doivent attendre la fin de l'exécution de la tâche `t` avant de s'exécuter.

À partir du programme en entrée, nous identifions les mécanismes de protection adéquats pour chaque instruction dans le programme. La stratégie est déployée en trois grandes étapes (identification des variables partagées, ensemble de verrous à acquérir, placement correct des verrous). Ici, nous nous intéressons uniquement à la première étape de cette stratégie. Nous assumons, en outre, certaines restrictions sur le langage C, les mêmes que les restrictions actuelles du greffon Alias (discutées en Section 4.1), et sur les directives de parallélisme : une tâche ne peut pas en lancer une autre et chaque tâche ne peut être lancée qu'une seule fois dans le programme.

Commençons par poser le problème à résoudre : pour chaque instruction du programme, nous voulons déterminer les noms de variables du programme qui pourraient référencer une zone de mémoire partagée. La présence de potentiels alias dans le programme complexifie fortement l'analyse. Le greffon Alias présenté dans les précédentes sections étant léger et ne nécessitant aucun paramétrage, il répond parfaitement à nos besoins. Prenons pour exemple

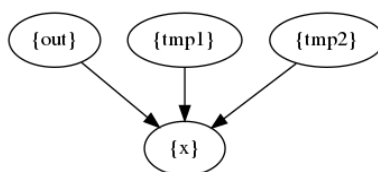


FIGURE 10. Graphe de pointeurs fusionné.

l’instruction `printf` à la ligne 20 du programme de la Figure 9.

Premièrement, nous identifions syntaxiquement les variables accédées, ainsi que les modes d’accès associées. Ici, il y a un seul accès en lecture, à la zone mémoire `*out`. Ensuite, nous identifions l’ensemble des tâches (et des instructions du `main` si l’instruction considérée est à l’intérieur d’une tâche) qui peuvent être exécutées en parallèle. Les éléments de cet ensemble sont appelés les compétiteurs de l’instruction. Ici, les deux compétiteurs sont `t1` et `t2`.

Nous utilisons le greffon `Alias` pour obtenir le graphe de pointeurs correspondant au point de programme situé avant l’exécution de l’instruction considérée. Si l’instruction fait partie de la fonction `main`, l’analyse a pour point de départ le début du programme et ne tient pas compte des tâches parallèles, sauf celles dont l’exécution est entièrement terminée. Si l’instruction fait partie du corps d’une tâche `t`, le point de départ de l’analyse est le début de `t`, dont le contexte d’appel est considéré comme vide (pas d’aliasing). Ici, au point de programme considéré, la variable `out` pointe vers `x`. Nous calculons ensuite les graphes de pointeurs après exécution des différents compétiteurs de l’instruction, considérés isolément les uns des autres. Dans le cas d’une instruction située à l’intérieur d’une tâche `t`, nous collectons également le graphe de pointeurs correspondant au point de programme situé avant le lancement de `t`. Ici, `tmp1` pointe vers `x` pour `t1` et `tmp2` pointe vers `x` pour `t2`.

Nous avons défini une notion de fusion de graphes de pointeurs, qui permet de sur-approximer les possibles zones mémoires pointées par les différentes variables en fonction de l’entrelacement considéré. Nous fusionnons le graphe collecté avant l’instruction considérée, les graphes collectés ci-dessus pour les compétiteurs et, si nécessaire, le graphe au point de lancement de la tâche dans laquelle se trouve l’instruction. La Figure 10 illustre la notion de graphe fusionné dans le cas de notre exemple. Ensuite, nous collectons les ensembles de variables syntaxiquement accédées par les différents compétiteurs, en tenant compte des modes d’accès mais en ignorant leurs compétiteurs respectifs. Ici, `t1` (resp. `t2`) accède à `tmp1` (resp. `tmp2`) en écriture et au tableau `in` en lecture. À partir de ces informations, nous identifions les variables à protéger, c’est-à-dire accédées par au moins un compétiteur, avec un accès au moins en écriture parmi l’instruction et les compétiteurs, en tenant compte de potentiels alias. Pour notre exemple, la variable `out` est identifiée car accédée en lecture dans l’instruction et en écriture par `t1` et `t2` via `tmp1` et `tmp2`, toutes deux en alias avec `out`.

6.2 Intégrité du flot de contrôle avec attestation à distance

L’*attestation à distance* permet la détection des comportements inattendus dans un composant logiciel. C’est une technique critique pour l’informatique de confiance, qui donne des garanties fortes concernant l’intégrité d’un composant logiciel. L’*intégrité du flot de contrôle* (*control flow integrity*, CFI) est une technique qui vérifie qu’un programme suit le flot de contrôle attendu pendant son exécution, par observation (*monitoring*). Le *flot de contrôle attendu* est une sur-approximation de tous les chemins d’exécution possible d’un programme qui s’exécute correctement. Il est déterminé par une analyse statique qui, en présence de pointeurs de fonction, dépend d’une analyse de pointeurs. Des déviations du flot de contrôle par rapport à celui attendu peuvent indiquer des manipulations du programme par un attaquant ou des défaillances matérielles. En attestant l’intégrité du flot de contrôle, CFI permet d’augmenter la confiance dans la sécurité et la sûreté d’un logiciel.

Nous sommes en train de développer un outil logiciel permettant l’attestation à distance de l’intégrité du flot de contrôle pour des programmes implémentés en C. Les composants

de cette architecture sont les suivants.

- Une analyse statique qui extrait le graphe de flot de contrôle du programme.
- Une instrumentation de code source qui insère, à certains points de programme (par exemple, avant les appels de fonction), des *points de contrôle*. Ces points de contrôle journalisent, pendant l'exécution du programme, le comportement de ce dernier (par exemple : le fait que la fonction f a été appelée depuis une certaine position p) et les transmettent au vérificateur.
- Un *vérificateur*, c'est-à-dire un serveur qui dispose du graphe de flot de contrôle statique et qui peut attester que le comportement journalisé et transmis depuis le programme respecte bien le flot de contrôle attendu.
- Un protocole de communication sécurisée qui assure aussi l'intégrité du code ajouté par l'instrumentation en s'appuyant sur un TPM (*Trusted Platform Module*), c'est-à-dire un module de stockage matériel sécurisé.
- Une architecture de déploiement qui facilite l'ensemble des étapes nécessaires pour exécuter un programme : l'analyse et l'instrumentation du programme, l'exécution du vérificateur et la transmission du graphe de flot de contrôle au vérificateur, la compilation et l'exécution du programme instrumenté.

L'analyse et l'instrumentation ont été implémentées comme un greffon pour Frama-C, appelé rCFI, non encore publiquement disponible. Ce greffon utilise au choix Eva ou Alias comme analyse de pointeurs. L'exactitude de l'analyse de pointeurs a un impact direct sur l'exactitude du graphe de flot de contrôle généré. Néanmoins, ici, les enjeux principaux du projet sont d'une part la minimisation des surcoûts en temps d'exécution et, d'autre part, la maximisation de l'exactitude du graphe de flot de contrôle. En effet une meilleure exactitude restreint la surface d'attaque, c'est-à-dire les comportements possibles qu'un attaquant peut exploiter pour arriver à son but. De plus il faut que les coûts d'analyse restent raisonnables. Il est donc intéressant d'avoir à disposition deux analyses avec des points forts différents et de déployer l'une ou l'autre selon le cas d'utilisation.

6.3 Optimisation pour la vérification déductive

WP est le greffon dédié à la vérification déductive de Frama-C. Il prend en entrée un code C annoté avec des spécifications écrites dans un langage de spécification formelle appelé ACSL [BCF⁺] dans le but de démontrer que ce code satisfait les spécifications. Pour cela, il génère des propriétés, appelées obligations de preuve : si l'ensemble de ces propriétés sont démontrées, alors le code C satisfait les spécifications [HH19]. Démontrer les obligations de preuve repose sur l'usage de prouveurs automatiques comme Alt-Ergo ou d'assistants à la preuve comme Coq. Un des enjeux pour ce type d'outils consiste à avoir le taux d'automatisation le plus élevé possible, c'est-à-dire à limiter au maximum l'usage des assistants à la preuve.

Générer d'une manière correcte les obligations de preuve demande une interprétation fine des sémantiques de C et d'ACSL. Il convient en outre de définir de bonnes abstractions pour représenter les opérations de ces langages. Ces abstractions permettent notamment de représenter les objets mathématiques comme les nombres entiers ou les nombres flottants, ainsi que les objets écrits en mémoire par le programme. En particulier, pour ces derniers objets, les abstractions liées à la gestion de la mémoire, appelées modèles mémoires, sont critiques dans le cas de l'analyse d'un code C : un modèle dit « de bas niveau » permet d'exprimer finement les propriétés sur la mémoire, mais a tendance à générer des obligations de preuve difficiles, voire impossibles, à prouver automatiquement, alors qu'un modèle dit de « haut niveau » autorise un taux élevé d'automatisation mais exclut la vérification de programmes contenant certaines opérations non représentables dans le modèle. Notamment, certains modèles ne fonctionnent pas en présence d'alias dans le code C .

Un courant de recherche actuel consiste donc à définir de nouveaux modèles mémoires permettant la vérification automatique de programmes de bas niveau, notamment en présence d'alias. Les techniques actuelles reposent notamment sur de l'analyse de régions

visant à séparer la mémoire en zones disjointes (appelées régions) permettant de garantir des propriétés d'isolation inter-régions augmentant l'automatisation des preuves. Ainsi, *Alias* sera utilisé dans le cadre de la définition de nouveaux modèles mémoires bas niveau qui seront développés dans l'année qui vient.

6.4 Optimisation pour la vérification à l'exécution

E-ACSL est le greffon de *Frama-C* dédié à la vérification à l'exécution. Il prend en entrée un programme *C* annoté avec des annotations ACSL et génère un nouveau programme *C* dans lequel les annotations ont été converties en code *C* : l'exécution du programme est fonctionnellement équivalente au programme *C* d'origine lorsque toutes les annotations sont correctes, ou sinon arrête l'exécution sur la première annotation incorrecte (ce comportement est celui par défaut, mais il est modifiable) [SKV17]. L'enjeu, ici, consiste à générer un code correct et le plus efficace possible.

Le langage ACSL permet notamment à l'utilisateur d'exprimer des propriétés sur la mémoire, par exemple le fait que des accès mémoires soient valides, c'est-à-dire que le programme a le droit d'y accéder en lecture et/ou en écriture, ou sur le fait que certaines données ont été correctement initialisées. Vérifier de telles propriétés pendant l'exécution du programme est compliqué et requiert, là encore, l'utilisation d'un modèle mémoire, cette fois dynamique, c'est-à-dire évoluant en cours d'exécution [VSK17]. Son usage requiert néanmoins une instrumentation invasive du code, requérant notamment d'ajouter de nouvelles instructions pour toutes les écritures mémoires de façon à mettre à jour le modèle mémoire d'E-ACSL. Or, si les annotations à vérifier sont indépendantes de certaines écritures mémoires, l'instrumentation de ces dernières est en réalité inutile. Ainsi, une analyse statique dédiée a été développée dans E-ACSL permettant de calculer une sur-approximation correcte des zones mémoires à surveiller [LKLS18], ce qui permet d'améliorer grandement l'efficacité du code produit [JKS16].

Cette analyse statique repose en réalité sur une analyse de pointeurs qui est, dans l'implémentation actuelle d'E-ACSL, effectuée en même temps que le reste de l'analyse. Ceci pose un certain nombre de problèmes pratiques, aussi bien au niveau du passage à l'échelle que de la correction, ce qui est problématique. Une étude formelle de cette analyse [LKLS18] a permis de démontrer qu'elle pouvait être paramétrée par l'analyse de pointeurs sous-jacente. Ainsi, nous envisageons d'implémenter une nouvelle version correcte et plus efficace de l'analyse actuelle, qui reposerait sur *Alias* pour effectuer son analyse de pointeurs.

7 Conclusion et perspectives

Cet article a présenté *Alias*, une nouvelle analyse de pointeurs fonctionnant sur du code *C* et implémentée comme un greffon de la plateforme *Frama-C* dédiée à l'analyse de code *C*. Il s'agit d'une adaptation de l'algorithme de Steensgaard [Ste96]. Elle est rapide et légère, dans le sens où elle demande moins de ressources (temps, mémoire) que l'analyse du greffon *Eva*, et ne nécessite aucun paramétrage de la part de l'utilisateur.

Cette caractéristique permet d'envisager de l'intégrer dans des analyseurs plus importants. Nous envisageons ainsi de l'utiliser dans quatre contextes différents, aussi bien liés au développement de nouveaux analyseurs pour la génération de programmes concurrents et la vérification de l'intégrité du flot de contrôle, que pour améliorer des analyseurs existants, comme *WP* dédié à la vérification déductive et E-ACSL dédié à la vérification à l'exécution.

L'efficacité et la précision d'*Alias* ont été mesurées sur un *benchmark* existant et représentatif. Les résultats démontrent qu'elle répond au besoin, en les comparant notamment à l'analyse d'*Alias* incluse dans le greffon *Eva* de *Frama-C*.

L'implémentation actuelle reste néanmoins un prototype. Les travaux futurs incluent donc l'extension de l'analyse pour supporter l'ensemble des constructions du langage *C*, par exemple les casts hétérogènes ou les types unions. Ils visent également à augmenter encore l'efficacité de l'analyseur.

Références

- [And94] Lars Ole ANDERSEN : *Program analysis and specialization for the C programming language*. Thèse de doctorat, University of Copenhagen, 1994.
- [BBB⁺21] Patrick BAUDIN, François BOBOT, David BÜHLER, Loïc CORRENSON, Florent KIRCHNER, Nikolai KOSMATOV, André MARONEZE, Valentin PERRELLE, Virgile PREVOSTO, Julien SIGNOLES et Nicky WILLIAMS : The Dogged Pursuit of Bug-Free C Programs : The Frama-C Software Analysis Platform. *Communications of the ACM*, 2021.
- [BBY17] Sandrine BLAZY, David BÜHLER et Boris YAKOBOWSKI : Structuring Abstract Interpreters through State and Value Abstractions. *In International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'17)*, janvier 2017.
- [BCF⁺] Patrick BAUDIN, Pascal CUOQ, Jean-Christophe FILLIÂTRE, Claude MARCHÉ, Benjamin MONATE, Yannick MOY et Virgile PREVOSTO : ACSL : ANSI/ISO C Specification Language.
- [CC77] Patrick COUSOT et Radhia COUSOT : Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *In Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '77*, 1977.
- [CFS05] Sylvain CONCHON, Jean-Christophe FILLIÂTRE et Julien SIGNOLES : Le foncteur somme toujours deux fois. *In Journées Francophones des Langages Applicatifs (JFLA)*, mars 2005.
- [CFS07] Sylvain CONCHON, Jean-Christophe FILLIÂTRE et Julien SIGNOLES : Designing a generic graph library using ML functors. *In Trends in Functional Programming (TFP)*, avril 2007.
- [CLHY05] Tong CHEN, Jin LIN, Wei-Chung HSU et Pen-Chung YEW : An empirical study on the granularity of pointer analysis in C programs. *In Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2005.
- [Cou22] Patrick COUSOT : *Principles of Abstract Interpretation*. MIT Press, 2022.
- [DMM98] Amer DIWAN, Kathryn S. MCKINLEY et J. Eliot B. MOSS : Type-Based Alias Analysis. *In International Conference on Programming Languages, Design and Implementation (PLDI)*, 1998.
- [GF64] Bernard A. GALLER et Michael J. FISHER : An improved equivalence algorithm. *Communications of the ACM*, mai 1964.
- [GPS10] Prodromos GERAKEIOS, Nikolaos PAPASPYROU et Konstantinos SAGONAS : Race-free and memory-safe multithreading : Design and implementation in cyclone. *In Workshop on Types in Language Design and Implementation (TLDI)*, 2010.
- [HH19] R. HÄHNLE et M. HUISMAN : *Deductive Software Verification : From Pen-and-Paper Proofs to Industrial Tools*. 2019.
- [Hor97] Susan HORWITZ : Precise flow-insensitive may-alias analysis is np-hard. *Transactions on Programming Languages and Systems (TOPLAS)*, 1997.
- [HT01] Nevin HEINTZE et Olivier TARDIEU : Demand-Driven Pointer Analysis. *In Conference on Programming Language Design and Implementation (PLDI)*, 2001.
- [JKS16] Arvid JAKOBSSON, Nikolai KOSMATOV et Julien SIGNOLES : Fast as a Shadow, Expressive as a Tree : Optimized Memory Monitoring for C. *Science of Computer Programming*, octobre 2016.
- [JLRS04] Bertrand JEANNET, Alexey LOGINOV, Thomas REPS et Mooly SAGIV : A relational approach to interprocedural shape analysis. *In International Static Analysis Symposium (SAS)*, 2004.

- [KKP⁺15] Florent KIRCHNER, Nikolai KOSMATOV, Virgile PREVOSTO, Julien SIGNOLES et Boris YAKOBOWSKI : Frama-c : A software analysis perspective. *Formal aspects of computing*, 2015.
- [LKLS18] Dara LY, Nikolai KOSMATOV, Frédéric LOULERGUE et Julien SIGNOLES : Soundness of a dataflow analysis for memory monitoring. In *Workshop on Languages and Tools for Ensuring Cyber-Resilience in Critical Software-Intensive Systems (HILT)*, novembre 2018.
- [Min06] Antoine MINÉ : Field-sensitive value analysis of embedded c programs with union types and pointer arithmetics. In *Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES)*, 2006.
- [Min15] Antoine MINÉ : Astréa : A static analyzer for large embedded multi-task software. In *16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'15)*, volume 8931, page 3. Springer, 2015.
- [MVT⁺16] Reed MILEWICZ, Rajesh VANKA, James TUCK, Daniel QUINLAN et Peter PIRKELBAUER : Lightweight runtime checking of C programs with RTC. *Computer Languages, Systems & Structures*, 2016.
- [MZGB06] Bill MCCLOSKEY, Feng ZHOU, David GAY et Eric BREWER : Autolocker : synchronization inference for atomic sections. In *symposium on Principles of Programming Languages (POPL)*, 2006.
- [NNH10] Flemming NIELSON, Hanne R. NIELSON et Chris HANKIN : *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010.
- [Our15] Alain OURGHANLIAN : Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nucl. Eng. Technol.*, 2015.
- [RS01] Noam RINETZKY et Mooly SAGIV : Interprocedural shape analysis for recursive programs. In *International Conference on Compiler Construction (CC)*, 2001.
- [SB15] Yannis SMARAGDAKIS et George BALATSOURAS : Pointer Analysis. *Foundations and Trends in Programming Languages*, 2015.
- [SGSB05] Manu SRIDHARAN, Denis GOPAN, Lexin SHAN et Rastislav BODÍK : Demand-Driven Points-to Analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '05*, page 59–76, New York, NY, USA, 2005. Association for Computing Machinery.
- [SKV17] Julien SIGNOLES, Nikolai KOSMATOV et Kostyantyn VOROBYOV : E-acsl, a runtime verification tool for safety and security of c programs. In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, septembre 2017.
- [Ste96] Bjarne STEENSGAARD : Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages (POPL)*, 1996.
- [TLM⁺21] Tian TAN, Yue LI, Xiaoxing MA, Chang XU et Yannis SMARAGDAKIS : Making Pointer Analysis More Precise by Unleashing the Power of Selective Context Sensitivity. 2021.
- [VSK17] Kostyantyn VOROBYOV, Julien SIGNOLES et Nikolai KOSMATOV : Shadow state encoding for efficient monitoring of block-level properties. In *International Symposium on Memory Management (ISMM)*, juin 2017.
- [YSZ19] Zeming YU, Linhai SONG et Yiyang ZHANG : Fearless concurrency ? Understanding concurrent programming safety in real-world Rust software. *arXiv preprint arXiv :1902.01906*, 2019.
- [ZR08] Xin ZHENG et Radu RUGINA : Demand-Driven Alias Analysis for C. *SIGPLAN Not.*, janvier 2008.

À la recherche de tous les vrais bugs

Verification formelle d'un détecteur de bugs automatique

Arthur Correnson¹

¹CISPA Helmholtz Center for Information Security

Pour assurer le bon fonctionnement des logiciels, il est crucial de les tester pendant leur développement. Pour faciliter cette tâche, de nombreuses méthodes de test automatique existent et permettent de détecter rapidement des erreurs de programmation. Pour être efficace et sûr, un détecteur de bugs automatique se doit d'être aussi précis et exhaustif que possible : il ne doit détecter que des *vrais bugs* et, si possible, être capable de détecter *tous* les *bugs*. Dans cet article, nous présentons un détecteur automatique de *bugs* formellement vérifié en Coq. En particulier nous prouvons que, sous certaines hypothèses, notre détecteur est précis et exhaustif.

1 Introduction

Contexte. Pour vérifier ou valider les logiciels, il existe plusieurs méthodes, principalement la preuve et le test. Les outils de preuve permettent d'établir, a posteriori, qu'un logiciel est conforme à une certaine spécification (c'est à dire, prouver que le logiciel *ne se trompe pas*). D'autre part, les outils de test aident à identifier des défauts dans les logiciels pendant leur conception en détectant des scénarios d'utilisation dans lesquels un logiciel *se trompe*. Qu'il s'agisse de preuve ou de test, l'usage d'outils automatiques pour valider la qualité des logiciels est de plus en plus fréquent. Cela amène à se poser la question de la fiabilité de ces outils. En réponse à cette question, beaucoup de travaux se sont concentrés sur l'étude de la correction des outils de preuve automatique comme par exemple les vérificateurs de modèles, les démonstrateurs SMT, ou les analyseurs statiques par interprétation abstraite. Dans ce contexte, la question posée est claire : si un vérificateur prétend qu'un programme respecte sa spécification, est-ce vraiment le cas ?

Motivations. Contrairement aux vérificateurs automatiques, les approches de **test** automatiques visent à exposer des défauts plutôt que de prouver leur absence. En conséquence, il est souvent admis que ces outils n'ont pas besoin d'être très fiables. Cela est dommage car, en raison de sa simplicité de mise en œuvre, le test reste la méthode la plus largement utilisée pour vérifier la robustesse des logiciels. Utilisé à bon escient, le test peut donner des garanties formelles. Par ailleurs les méthodes de test automatique sont aussi sujettes à de nombreuses erreurs qui peuvent avoir des conséquences néfastes [God05]. Deux types d'erreurs peuvent se produire : le signalement de *faux bugs*, et le manquement de *bugs* que l'on espérait être détectés. Le premier type d'erreurs peut, à l'extrême, rendre un outil de test complètement inutile. En effet, inonder les utilisateurs de fausses alarmes ne fait que déplacer le problème de trouver des erreurs dans un logiciel vers celui de trouver des erreurs dans l'outil de recherche de *bugs* lui-même. Symétriquement, un détecteur de *bugs* qui manque trop d'erreurs donne de très faibles garanties et peut donner la fausse illusion qu'un

logiciel est robuste. Pour palier ces deux problèmes, il est désirable de pouvoir interpréter précisément les résultats d'un détecteur de *bugs* automatique. Si un *bug* est signalé, qu'est-ce que cela veut dire ? Si aucun *bug* n'est signalé, quelles garanties avons-nous obtenues ?

Contributions. Dans cet article, nous proposons de prouver la correction d'un détecteur de *bugs* à l'aide de l'assistant de preuve Coq. L'objectif est d'être capable d'interpréter formellement les verdicts de l'outil, qu'ils soient positifs ou négatifs. Plus précisément, nous prouvons un résultat de *précision* : tout *bug* détecté est un véritable *bug* dans le programme analysé ; ainsi qu'un résultat d'exhaustivité : tout *bug* finira éventuellement par être détecté. L'outil est fondé sur des méthodes d'exécution symbolique et permet d'analyser des programmes impératifs simples.

Tous les résultats présentés dans cet article sont issus d'un travail plus approfondi et déjà soumis en anglais à la conférence *Foundations of Software Engineering* [CS23]. L'objectif principal de cet article est de faire une synthèse de ces travaux en Français. L'accent est mis sur la présentation de l'approche plutôt que sur les détails techniques des preuves.

2 Vérifier les vérificateurs

Comment faire confiance aux méthodes formelles ? Établir la correction d'un vérificateur automatique peut passer par plusieurs approches. Par exemple, on peut exiger que les vérificateurs produisent des preuves en plus d'un simple résultat binaire "oui/non". Ces preuves jouent le rôle de certificat de correction et peuvent être relues par un expert, ou par un validateur automatique. Par exemple, les démonstrateurs automatiques VeriT [BCBdODF09] et CVC4 [BCD⁺11] produisent des preuves qui peuvent être validées par un validateur de preuves fiable comme SMTCoq [AFG⁺11]. D'autres outils de preuve comme les vérificateurs de modèles peuvent suivre cette approche [Nam01]. Par exemple, le vérificateur SLAB [DKFW10] produit des certificats. Une autre approche est de vérifier une fois pour toute la correction des vérificateurs en utilisant un autre vérificateur. Cette approche peut sembler vaine à première vue. Pourquoi ferions-nous plus confiance au vérificateur du vérificateur qu'au vérificateur lui-même ? Pour éviter cette spirale sans fin, il existe une solution *simple* qui fait consensus : utiliser un *assistant de preuve* comme second vérificateur.

Les assistants de preuves comme base de confiance. Les assistants de preuve sont des outils de vérification *minimalistes* et *robustes*. Ils permettent d'écrire des programmes et leur preuve de correction dans un langage formel. Contrairement à d'autres outils plus automatiques (par exemple, les démonstrateurs SMT), les preuves sont écrites manuellement par un expert et sont ensuite mécaniquement vérifiées par l'assistant. La tâche de valider la correction des preuves est suffisamment simple et élémentaire pour que l'on puisse faire l'hypothèse que l'assistant de preuve ne se trompe jamais. Par ailleurs, comme les preuves sont faites par l'utilisateur, on peut se permettre d'écrire des théorèmes et de définir des objets plus complexes. En particulier, on peut implémenter et prouver la correction de logiciels aussi complexes qu'un compilateur C [LBK⁺16] ou un analyseur statique [JLB⁺15] à l'aide d'un assistant à la démonstration. Cette méthode, comme toutes les autres, n'est pas parfaite (un assistant de preuve pourrait toujours contenir un *bug* !), mais elle donne un niveau de garanties difficilement égalable.

Vérifier les prouveurs. Les assistants de preuve ont été utilisés avec succès pour prouver la correction de divers vérificateurs automatiques. Par exemple, Verasco est un interprète abstrait pour le langage C dont la correction a été entièrement vérifiée en Coq [JLB⁺15]. D'autres méthodes de vérification automatiques comme la *vérification de modèles* pour les propriétés temporelles ont également été prouvées correctes à l'aide de l'assistant de preuve

Isabelle/HOL [ELN⁺13]. Ces projets démontrent qu'il est possible de prouver formellement la correction d'outils de vérification automatique réalistes.

Vérifier les détecteurs de bugs. Des travaux récents se sont intéressés à la formalisation des méthodes de test. On peut notamment citer les travaux sur *incorrectness logic* [O'H19], qui proposent d'utiliser des logiques de programmes pour prouver l'existence d'erreurs plutôt que de prouver leur absence. D'autres approches, plus automatiques, ont également été explorées. Par exemple, la plateforme Gillian a une fonction de recherche de bugs qui a été formalisée rigoureusement [FSMAG20]. En revanche, les preuves n'ont pas été mécanisées dans un assistant à la démonstration. À notre connaissance, le seul effort de mécanisation d'un détecteur de *bugs* automatique dans un assistant à la démonstration est *QuickChick*, une bibliothèque logicielle pour tester la correction de programmes Coq en utilisant des méthodes de génération d'entrées aléatoires [PHD⁺14].

3 Détection automatique de bugs par exécution symbolique

Qu'est-ce qu'un bug ? Selon le contexte, il existe plusieurs manières de définir ce qu'est un "*bug*" dans un programme informatique. Par exemple, il peut s'agir d'une simple erreur à l'exécution comme une division par zéro ou un accès hors des bornes d'un tableau. Plus généralement, les erreurs à l'exécution sont toutes les situations dans lesquelles un programme est contraint d'être interrompu prématurément. Un *bug* peut aussi désigner une erreur de programmation qui n'entraîne pas nécessairement d'erreur à l'exécution mais qui conduit un programme à produire des résultats incorrects ou inattendus. Dans la suite de cet article, nous nous concentrons sur la détection des erreurs à l'exécution. Cela n'est pas forcément une limitation car on peut détecter certaines erreurs de correction en les ramenant à des erreurs à l'exécution en utilisant des *assertions* (c'est à dire, en forçant l'interruption d'un programme lorsque certaines conditions de correction ne sont pas satisfaites pendant l'exécution).

Trouver les bugs en exécutant les programmes. Pour détecter des erreurs à l'exécution, une méthode naïve est d'employer la force brute. On peut exécuter un programme un très grand nombre de fois en essayant un maximum de combinaisons d'entrées possibles (cette méthode est appelée *fuzzing* [FMEH20]). Si le programme part en erreur pour certains choix d'entrées, il faut le mettre à jour en conséquence. Cette approche peut s'avérer très efficace pour détecter rapidement des cas particuliers qui ont été oubliés lors du développement d'un programme. En revanche, la force brute a plusieurs limitations. Tout d'abord, il est impossible de couvrir toutes les entrées possibles en pratique. Par ailleurs, certaines erreurs sont presque impossible à identifier simplement en testant des combinaisons d'entrées au hasard. Considérons par exemple le programme suivant :

```
if (x == 42) {
    fail();
}
return 2 * x;
```

Ce programme peut partir en erreur si on choisit $x = 42$. En revanche, la probabilité de choisir la bonne valeur de x sans inspecter le code source est presque nulle (si on considère des entiers 64 bits, il y a seulement $\frac{1}{2^{64}}$ chances de trouver l'erreur!).

Trouver les bugs par exécution symbolique. Une solution aux limites des approches de test par force brute est d'avoir recours à des techniques d'*exécution symbolique* pour essayer de prédire l'ensemble de toutes les issues possibles d'un programme [Kin76]. L'idée

est d'exécuter les programmes en traitant les entrées comme des *symboles opaques*. Lors de l'exécution symbolique d'un programme, quand un branchement gardé par une condition (par exemple un `if` ou une boucle `while`) est exécuté, on envisage tous les scénarios possibles selon que la condition est supposée satisfaite ou non. On prend soin de mémoriser les conditions que l'on a supposées satisfaites tout au long de l'exécution. Le résultat d'une telle *exécution symbolique* est un ensemble représentant toutes les issues possibles d'un programme. Chaque issue est exprimée comme une fonction des entrées et est annotée par l'ensemble des hypothèses qui doivent être satisfaites pour atteindre le résultat associé. Les issues possibles sont typiquement "le programme part en erreur" (on note `fail`) ou bien "le programme termine et retourne un résultat" (on note `return`). On note $\langle H, r \rangle$ une issue r étiquetée par un ensemble d'hypothèses H . Par exemple, l'exécution symbolique de l'exemple précédent donne les deux issues $\{\langle x = 42, \text{fail} \rangle, \langle x \neq 42, \text{return}(2 * x) \rangle\}$.

Si une issue possible est une erreur et est étiquetée par un ensemble d'hypothèses *satisfiables*, alors le programme contient un *bug*. Cela donne une méthode systématique pour tester un programme automatiquement. On commence par exécuter le programme symboliquement, puis on parcourt exhaustivement les issues obtenues. Pour chaque issue représentant une erreur, on peut faire appel à un solveur de contraintes pour vérifier que les hypothèses conduisant à l'erreur en question sont réalisables.

```

E ← exécution-symbolique(P)
for all  $\langle H, r \rangle \in E$  do
  if  $r$  est une erreur et  $H$  est satisfiable then
    return  $P$  part en erreur pour toutes entrées satisfiant  $H$ 
  end if
end for
return  $P$  ne part jamais en erreur

```

FIGURE 1. Un algorithme simple pour détecter les bugs dans un programme P

Si l'exécution symbolique de P est effectuée rigoureusement (nous reviendrons sur ce point dans la partie 4) et sous l'hypothèse que le test de satisfiabilité des hypothèses est correct, cet algorithme ne détecte que des vrais bugs. On dit qu'il est *précis*. En revanche, l'ensemble des issues symboliques d'un programme n'est pas toujours fini. Par ailleurs, l'exécution symbolique d'un programme qui ne termine pas peut diverger. En général, on ne peut donc calculer qu'un sous-ensemble des issues possibles. Par conséquent, l'algorithme 1 n'est pas *exhaustif* et il peut manquer des bugs. Toutefois, en calculant l'ensemble des issues de façon *paresseuse*, il est possible d'obtenir une implémentation *relativement exhaustive* : toute erreur à l'exécution sera éventuellement découverte pourvu que l'on laisse l'algorithme tourner suffisamment longtemps et avec suffisamment de ressources mémoire. Dans la suite de cet article, nous proposons une telle implémentation de cet algorithme en Coq. Nous effectuons aussi la preuve formelle de précision et d'exhaustivité relative.

4 Formaliser l'exécution symbolique

Dans cette partie, nous nous penchons sur la formalisation de la notion d'exécution symbolique. Nous commençons par introduire un langage de programmation simple que nous équipons avec une sémantique formelle pour décrire son modèle d'exécution. Puis, nous introduisons une deuxième sémantique formelle pour décrire un modèle d'exécution symbolique pour ce langage. Enfin, nous établissons une connection entre la sémantique concrète et la sémantique symbolique. Cette connection servira de fondation pour justifier la correction d'un interprète symbolique.

Le langage d'étude BUG. On se donne un langage impératif à contrôle structuré (avec des constructions **if** et **while**) pouvant manipuler des expressions arithmétiques simples. Nous étendons ce langage avec une instruction **Error** qui provoque l'interruption volontaire de l'exécution. Cette instruction peut-être utilisée pour instrumenter les programmes. Par exemple, pour modéliser des régions inaccessibles ou des assertions logiques qui ne devraient jamais être invalidées à l'exécution. La syntaxe complète de ce langage, que l'on appellera BUG dans la suite, est définie comme suit.

Définition 1 (Syntaxe du langage BUG).

Variables	$var \in \mathbb{V}$
Arithmétique	$\mathbb{A}expr \ni aexpr ::= c \in \mathbb{Z} \mid var$ $\mid aexpr \ (+ \mid -) \ aexpr$
Booléens	$\mathbb{B}expr \ni bexpr ::= \mathbf{true} \mid \mathbf{false}$ $\mid aexpr \ (< \mid =) \ aexpr$ $\mid bexpr \ (\mathbf{and} \mid \mathbf{or}) \ bexpr$ $\mid \mathbf{not} \ bexpr$
Instructions	$\mathbb{I}nstr \ni instr ::= \mathbf{skip} \mid \mathbf{fail}$ $\mid var = aexpr$ $\mid instr ; instr$ $\mid \mathbf{if} \ bexpr \ \mathbf{then} \ instr \ \mathbf{else} \ instr$ $\mid \mathbf{while} \ bexpr \ \mathbf{do} \ instr$

Sémantique concrète. Les programmes BUG opèrent sur une mémoire qui associe à chaque variables une valeur entière. Dans la suite, on notera $\mathbb{M} = \mathbb{V} \rightarrow \mathbb{Z}$ l'ensemble des états mémoires. Étant donnée une mémoire $M \in \mathbb{M}$, une variable $v \in \mathbb{V}$ et une constante $z \in \mathbb{Z}$, on note $M[x \leftarrow z]$ la mémoire M où la valeur de v à été remplacée par z . Par ailleurs, étant donnée une expression $e \in \mathbb{A}expr \cup \mathbb{B}expr$ on note $\llbracket e \rrbracket_M$ la valeur de e dans la mémoire M . Dans le cas où e est une expression arithmétique on a $\llbracket e \rrbracket_M \in \mathbb{Z}$. Si e est une expression booléenne on a $\llbracket e \rrbracket_M \in \{\mathbf{true}, \mathbf{false}\}$.

La sémantique des instructions est donnée dans un style à *petits pas*. Un état d'exécution est une paire $\langle M, i \rangle \in \mathbb{M} \times \mathbb{I}nstr$ où M est l'état courant de la mémoire et i est la prochaine instruction à exécuter. On note $\langle M_1, i_1 \rangle \hookrightarrow \langle M_2, i_2 \rangle$ pour exprimer que l'exécution de l'instruction i_1 dans la mémoire M_1 mène à la mémoire M_2 et i_2 est la prochaine instruction à exécuter. Les transitions valides entre états sont définies par les règles d'inférence suivantes.

Définition 2 (Sémantique concrète de BUG).

$$\begin{array}{c}
\overline{\langle M, x = e \rangle \hookrightarrow \langle M[x \leftarrow \llbracket e \rrbracket_M], \mathbf{skip} \rangle} \\
\\
\frac{}{\overline{\langle M, \mathbf{skip} ; i \rangle \hookrightarrow \langle M, i \rangle}} \quad \frac{\langle M, i_1 \rangle \hookrightarrow \langle M, i_2 \rangle}{\overline{\langle M, i_1 ; i_3 \rangle \hookrightarrow \langle M, i_2 ; i_3 \rangle}} \\
\\
\frac{\llbracket b \rrbracket_M = \mathbf{true}}{\overline{\langle M, \mathbf{if} \ b \ \mathbf{then} \ i_1 \ \mathbf{else} \ i_2 \rangle \hookrightarrow \langle M, i_1 \rangle}} \quad \frac{\llbracket b \rrbracket_M = \mathbf{false}}{\overline{\langle M, \mathbf{if} \ b \ \mathbf{then} \ i_1 \ \mathbf{else} \ i_2 \rangle \hookrightarrow \langle M, i_2 \rangle}} \\
\\
\frac{\llbracket b \rrbracket_M = \mathbf{true}}{\overline{\langle M, \mathbf{while} \ b \ \mathbf{do} \ i \rangle \hookrightarrow \langle M, i ; \mathbf{while} \ b \ \mathbf{do} \ i \rangle}} \quad \frac{\llbracket b \rrbracket_M = \mathbf{false}}{\overline{\langle M, \mathbf{while} \ b \ \mathbf{do} \ i \rangle \hookrightarrow \langle M, \mathbf{skip} \rangle}}
\end{array}$$

À l'aide de cette sémantique, on peut définir formellement la notion d'erreur à l'exécution. Étant donné un état $\langle M_1, i_1 \rangle$, on note $\langle M_1, i_1 \rangle \hookrightarrow^* \langle M_2, i_2 \rangle$ si le nouvel état $\langle M_2, i_2 \rangle$ peut être atteint en 0, une ou plusieurs étapes d'exécution. En partant d'un état $\langle M_1, i_1 \rangle$, on dit qu'il y a une erreur à l'exécution s'il existe un état $\langle M_2, i_2 \rangle$ tel que $\langle M_1, i_1 \rangle \hookrightarrow^* \langle M_2, i_2 \rangle$ et $\langle M_2, i_2 \rangle$ est "bloqué" (c'est à dire, n'a aucun successeur valide d'après la relation \hookrightarrow). Les seuls états bloqués sont ceux de la forme $\langle M, \text{skip} \rangle$ et $\langle M, \text{fail} ; \dots \rangle$. Bien sûr, les états $\langle M, \text{skip} \rangle$ ne doivent pas être considérés comme des états d'erreur. On définit donc la notion de bug comme suit.

Définition 3. Un programme p possède un bug s'il existe des états mémoire M_1, M_2 tels que $\langle M_1, p \rangle \hookrightarrow^* \langle M_2, \text{fail} ; \dots \rangle$.

Sémantique symbolique. D'après la définition 3, trouver un bug revient à exposer un état mémoire M_1 conduisant à une erreur. Pour automatiser la recherche d'une telle mémoire initiale, on effectue l'exécution de p en suivant des règles d'exécution symbolique plutôt que la sémantique concrète donnée par la relation \hookrightarrow . Lors de l'exécution symbolique d'un programme, on remplace les états mémoire \mathbb{M} par des états mémoire symboliques $\mathbb{M}_{\text{sym}} = \mathbb{V} \rightarrow \mathbb{A}expr$ qui associent chaque variable à une expression arithmétique. Cela permet d'affecter aux variables des valeurs qui dépendent des entrées du programme. Pour évaluer une expression arithmétique ou booléenne dans une mémoire symbolique, il suffit de substituer chaque variable par son expression associée. On note $\llbracket e \rrbracket_{\hat{M}}^{\text{sym}}$ la valeur symbolique d'une expression dans une mémoire symbolique $\hat{M} \in \mathbb{M}_{\text{sym}}$. Au début de l'exécution symbolique d'un programme, on choisit comme mémoire symbolique $\hat{M}_{\text{init}} = x \mapsto x$ (chaque variable contient son propre nom comme valeur symbolique). Par exemple, si l'on exécute symboliquement la séquence d'instructions $x = 2 + y ; x = x + 1$, la mémoire symbolique obtenue en sortie associe l'expression $(2 + y) + 1$ à la variable x .

On formalise cette intuition en donnant une nouvelle sémantique aux programmes. Dans le contexte de l'exécution symbolique, les états sont des triplets $\langle \varphi, \hat{M}, i \rangle$ où $\varphi \in \mathbb{B}expr$ est une expression booléenne représentant un ensemble d'hypothèses qui sont faites sur les entrées du programme, $\hat{M} \in \mathbb{M}_{\text{sym}}$ est une mémoire symbolique, et $i \in \mathbb{I}nstr$ est une instruction à exécuter. On note \mathbb{S}_{sym} l'ensemble des états symboliques et les transitions valides entre états sont données par une relation $\hookrightarrow_{\text{sym}} \subseteq \mathbb{S}_{\text{sym}} \times \mathbb{S}_{\text{sym}}$. Cette sémantique suit exactement la même structure que celle de la définition 2. Toutefois, une différence notable est que les instructions `if` et `while` sont exécutées de manière *non-déterministe*. L'exécution de ces instructions peut se poursuivre dans l'une ou l'autre branche au choix. Dans tous les cas, les conditions booléennes (ou leurs négations) sont évaluées de manière symbolique avant d'être ajoutées à l'ensemble des hypothèses courantes.

Définition 4 (Sémantique symbolique de BUG).

$$\begin{array}{c}
\frac{}{\langle \varphi, \hat{M}, x = e \rangle \hookrightarrow_{\text{sym}} \langle \varphi, \hat{M}[x \leftarrow \llbracket e \rrbracket_{\hat{M}}^{\text{sym}}], \text{skip} \rangle} \\
\frac{}{\langle \varphi, \hat{M}, \text{skip} ; s \rangle \hookrightarrow_{\text{sym}} \langle \varphi, \hat{M}, s \rangle} \qquad \frac{\langle \varphi, \hat{M}, s_1 \rangle \hookrightarrow_{\text{sym}} \langle \varphi, \hat{M}, s_2 \rangle}{\langle \varphi, \hat{M}, s_1 ; s_3 \rangle \hookrightarrow \langle \varphi, \hat{M}, s_2 ; s_3 \rangle} \\
\frac{}{\langle \varphi, \hat{M}, \text{if } b \text{ then } i_1 \text{ else } i_2 \rangle \hookrightarrow_{\text{sym}} \langle \varphi \text{ and } \llbracket b \rrbracket_{\hat{M}}^{\text{sym}}, \hat{M}, i_1 \rangle} \\
\frac{}{\langle \varphi, \hat{M}, \text{if } b \text{ then } i_1 \text{ else } i_2 \rangle \hookrightarrow_{\text{sym}} \langle \varphi \text{ and not } \llbracket b \rrbracket_{\hat{M}}^{\text{sym}}, \hat{M}, i_2 \rangle} \\
\frac{}{\langle \varphi, \hat{M}, \text{while } b \text{ do } i \rangle \hookrightarrow_{\text{sym}} \langle \varphi \text{ and } \llbracket b \rrbracket_{\hat{M}}^{\text{sym}}, \hat{M}, i ; \text{while } b \text{ do } i \rangle} \\
\frac{}{\langle \varphi, M, \text{while } b \text{ do } i \rangle \hookrightarrow_{\text{sym}} \langle \varphi \text{ and not } \llbracket b \rrbracket_M^{\text{sym}}, M, \text{skip} \rangle}
\end{array}$$

Correction et complétude. Il existe un lien formel entre les sémantiques concrète et symbolique. D'une part, toute exécution symbolique simule un ensemble d'exécution concrètes. D'autre part, toute exécution concrète peut être simulée par une exécution symbolique. On dit que la sémantique symbolique est une abstraction correcte et complète de la sémantique concrète. Plus précisément, la correction assure que toute exécution symbolique peut être *transformée* en exécution concrète en choisissant une mémoire initiale qui satisfait les hypothèses collectées durant l'exécution symbolique.

Théorème 1 (Correction). *Soit i_1 et i_2 deux instructions, \hat{M} une mémoire symbolique, et φ une expression booléenne tels que $\langle \text{true}, \hat{M}_{\text{init}}, i_1 \rangle \hookrightarrow_{\text{sym}}^* \langle \varphi, \hat{M}, i_2 \rangle$. Alors, pour toute mémoire concrète M telle que $\llbracket \varphi \rrbracket_M = \text{true}$, on a $\langle M, i_1 \rangle \hookrightarrow^* \langle M \circ \hat{M}, i_2 \rangle$*

Ici, la notation $M \circ \hat{M}$ désigne la mémoire concrète obtenue en remplaçant toutes les variables libres dans la mémoire symbolique \hat{M} par leurs valeurs dans M . En corollaire du théorème 1, on obtient également que l'exécution symbolique peut être utilisée pour détecter précisément les bugs.

Corollaire 1. *Soit p un programme, \hat{M} une mémoire symbolique, et φ une expression booléenne tels que $\langle \text{true}, \hat{M}_{\text{init}}, p \rangle \hookrightarrow_{\text{sym}}^* \langle \varphi, \hat{M}, \text{fail} ; \dots \rangle$. Alors p part en erreur pour toute mémoire initiale qui satisfait φ .*

La complétude assure que toute exécution concrète à partir d'une mémoire donnée peut être simulée par une exécution symbolique de sorte que les hypothèses collectées *généralisent* la mémoire en question.

Théorème 2 (Complétude). *Soient i_1 et i_2 deux instructions et M_1 et M_2 deux mémoires tels que $\langle i_1, M_1 \rangle \hookrightarrow^* \langle i_2, M_2 \rangle$. Alors il existe une expression booléenne φ et une mémoire symbolique \hat{M} tels que $\langle \text{true}, i_1, \hat{M}_{\text{init}} \rangle \hookrightarrow_{\text{sym}}^* \langle \varphi, i_2, \hat{M} \rangle$ et $M_2 = M_1 \circ \hat{M}$ et $\llbracket \varphi \rrbracket_{M_1} = \text{true}$.*

En corollaire de 2, on obtient que toute exécution qui part en erreur peut être simulée par exécution symbolique.

Corollaire 2. *Soit p un programme qui part en erreur si il est exécuté depuis une certaine mémoire M . Alors, il existe une expression booléenne φ et une mémoire symbolique \hat{M} tels que $\langle \text{true}, \hat{M}_{\text{init}}, p \rangle \hookrightarrow_{\text{sym}}^* \langle \varphi, \hat{M}, \text{fail} ; \dots \rangle$ et M satisfait φ .*

L'ensemble de ces résultats de correction et de complétude peuvent être énoncés et prouvés en Coq. Un schéma de preuve est présenté dans l'article [dBB19] et une formalisation Coq est détaillée dans [CS23].

5 Implémentation vérifiée d'un détecteur de bugs

Un interpréteur symbolique correct et complet. La sémantique symbolique décrite dans la partie précédente n'est pas directement exécutable. En particulier, cette sémantique est non-déterministe puisque l'on peut choisir de poursuivre l'exécution de deux manières différentes lorsqu'un branchement est rencontré. Pour rendre la sémantique symbolique exécutable, il faut résoudre ce non-déterminisme. Pour se faire, on commence par définir une fonction `expand` : $\mathbb{S}_{\text{sym}} \rightarrow \text{list } \mathbb{S}_{\text{sym}}$ qui associe à chaque état symbolique la liste de tous ses successeurs possibles. L'implémentation d'une telle fonction est immédiate si l'on suit les règles décrites à la définition 4. Cette fonction d'*expansion* des états symboliques reflète la sémantique symbolique au sens du théorème suivant.

Théorème 3 (Correction et complétude de la fonction d'expansion). *Soient $s_1, s_2 \in \mathbb{S}_{\text{sym}}$, $s_1 \hookrightarrow_{\text{sym}} s_2$ si et seulement si $s_2 \in \text{expand } s_1$.*

Pour calculer l'ensemble des états (symboliques) accessibles d'un programme p , on itère simplement la fonction `expand` en partant de l'état $\langle \text{true}, \mathcal{M}_{\text{init}}, p \rangle$. En revanche, ce processus peut ne jamais terminer. En effet, il se peut qu'un état génère une suite infinie de successeurs si le programme analysé ne termine pas.

Une approche paresseuse pour résoudre le problème de la terminaison. Pour résoudre le problème de non-terminaison, on se contente d'énumérer l'ensemble des états accessibles sous la forme d'une séquence infinie évaluée de manière *paresseuse*. En langage OCaml, les séquences infinies peuvent être implémentées en utilisant des *suspensions* (grâce au module `Lazy` par exemple). En Coq, on utilise des listes co-inductives, aussi appelées *streams*. Ci-dessous, nous proposons une implémentation possible des *stream* en OCaml (à gauche) et en Coq (à droite).

<pre> type 'a cell = snil scon of 'a * 'a stream with 'a stream = 'a cell Lazy.t </pre>	<pre> CoInductive stream A := snil scon (x : A) (xs : stream A) . </pre>
---	--

En utilisant le type `stream`, on peut écrire en Coq une fonction récursive `reachable` : $\text{list } \mathbb{S}_{\text{sym}} \rightarrow \text{stream } \mathbb{S}_{\text{sym}}$ qui énumère tous les états symboliques accessibles en partant d'une liste d'états initiaux.

<pre> CoFixpoint reachable l := match l with [] => snil s::l => scon s (reachable (l ++ expand s)) end. </pre>
--

A l'aide du théorème 3, on peut prouver que le *stream* `reachable l` énumère bien exactement l'ensemble des états symboliques $\hookrightarrow_{\text{sym}}$ -accessibles depuis l . Notons toutefois que cela n'est pas vrai si la liste des états courants est étendue à gauche au lieu d'être étendue à droite d'un appel récursif à l'autre! En effet, en concaténant les successeurs à droite, on réalise un parcours en largeur de l'espace d'états. Cet argument joue un rôle crucial dans la preuve et garantit l'exhaustivité du parcours, même lorsqu'il existe une infinité d'états accessibles.

Théorème 4. *Soit l une liste d'états symboliques et $s_2 \in \mathbb{S}_{\text{sym}}$. s_2 apparaît dans la séquence `reachable l` si et seulement si il existe un état $s_1 \in l$ tel que $s_1 \hookrightarrow_{\text{sym}}^* s_2$.*

En conséquence de ce théorème et des théorèmes de correction et de complétude de la sémantique symbolique, on obtient le corollaire suivant.

Corollaire 3. *Un programme p part en erreur si et seulement s’il existe une mémoire symbolique \hat{M} et une expression booléenne satisfiable φ tels qu’un état de la forme $\langle \varphi, \hat{M}, \mathit{fail}; \dots \rangle$ apparaît dans la séquence `reachable` [`true`, \hat{M}_{init} , p]*

Implémentation finale d’un détecteur de bug. Le corollaire 3 suggère une manière fiable d’implémenter l’algorithme de détection de bug 1. Il suffit de balayer le *stream* des états accessibles jusqu’à trouver un état symbolique dont la condition associée est satisfiable. Dans ce cas, le programme analysé contient nécessairement un bug. Si aucun état d’erreur n’est rencontré et que le *stream* est épuisé, le programme analysé ne contient pas de bugs. Si la recherche n’aboutit pas, on peut laisser l’utilisateur interrompre l’algorithme après un certain temps.

En pratique, on se contente de vérifier la fonction `reachable` en Coq. On génère ensuite automatiquement une version OCaml de cette fonction en utilisant le mécanisme d’extraction de Coq [Let08]. La lecture d’un fichier source, l’appel à la fonction `reachable` et son parcours sont programmés en OCaml. Dans notre implémentation, nous utilisons le solveur de contrainte Z3 [dMB08] pour tester la satisfiabilité des conditions booléennes. La question de vérifier la correction du solveur de contrainte sort du cadre de cet article. Toutefois, une solution plus fiable serait de prouver la correction d’un solveur de contraintes en Coq et de faire en sorte que la fonction `reachable` filtre les états dont la condition est non-satisfiable. Prouver la correction des solveurs de contraintes est un sujet de recherche à part entière qui a déjà reçu beaucoup d’attention [AFG⁺11, Les11, CDG12, BFW17]. Exploiter l’état de l’art dans ce domaine est une piste de réflexion intéressante pour de futurs travaux.

6 Conclusion

S’il est admis qu’il est utile voire nécessaire de vérifier la fiabilité des outils de preuve, les outils de test ont reçu moins d’attention. Dans cet article, nous avons proposé de vérifier la correction d’un détecteur de *bugs* automatique à l’aide de l’assistant de preuve Coq. En particulier, nous avons prouvé qu’il énumère toutes les erreurs à l’exécution. En supposant l’accès à un solveur de contraintes fiable, nous avons également prouvé que notre algorithme ne détecte que des vraies erreurs. Ce premier prototype ouvre la voie à de nombreuses possibilités d’extensions futures. En particulier, il serait intéressant de lever l’hypothèse d’accès à un solveur correct. Un autre angle de recherche, orthogonal, serait de se concentrer sur l’analyse de langages plus réalistes pour lesquels il existe une sémantique formalisée en Coq comme par exemple C [BL09] ou JavaScript [BCF⁺14].

Accessibilité des sources. Les théorèmes et des définitions énoncés dans cet article ont tous été formalisés en Coq. Les preuves sont disponibles à l’adresse <https://github.com/acorrenson/minibug>. Une version plus complète comprenant l’extraction vers OCaml peut être consultée librement à l’adresse <https://github.com/acorrenson/wiSE>.

Remerciements. Je remercie Dominic Steinhöfel et Sébastien Bardin pour les nombreuses discussions que nous avons eu sur la *sémantique des bugs* et sur la correction des interprètes symboliques. Je remercie également mes camarades de promotion Charles de Haro et Naïm Moussaoui Remil pour les débats agités mais constructifs que nous avons eu sur les critères de *soundness* des outils d’analyse statique. Enfin, merci au comité des JFLA pour les commentaires détaillés qui ont guidés la rédaction de la version finale de cet article.

Financements. This work was supported by the European Research Council (ERC) Grant HYPER (No. 101055412). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Références

- [AFG⁺11] Michael ARMAND, Germain FAURE, Benjamin GRÉGOIRE, Chantal KELLER, Laurent THÉRY et Benjamin WERNER : A modular integration of sat/smt solvers to coq through proof witnesses. *In* Jean-Pierre JOUANNAUD et Zhong SHAO, éditeurs : *Certified Programs and Proofs*, pages 135–150, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [BCBdODF09] Thomas BOUTON, Diego Caminha B. de OLIVEIRA, David DÉHARBE et Pascal FONTAINE : verit : An open, trustable and efficient smt-solver. *In* Renate A. SCHMIDT, éditeur : *Automated Deduction – CADE-22*, pages 151–156, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [BCD⁺11] Clark W. BARRETT, Christopher L. CONWAY, Morgan DETERS, Liana HADAREAN, Dejan JOVANOVIĆ, Tim KING, Andrew REYNOLDS et Cesare TINELLI : CVC4. *In* Ganesh GOPALAKRISHNAN et Shaz QADEER, éditeurs : *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 de *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.
- [BCF⁺14] Martin BODIN, Arthur CHARGUERAUD, Daniele FILARETTI, Philippa GARDNER, Sergio MAFFEIS, Daiva NAUDZIUNIENE, Alan SCHMITT et Gareth SMITH : A trusted mechanised javascript specification. *SIGPLAN Not.*, 49(1):87–100, jan 2014.
- [BFW17] Jasmin Christian BLANCHETTE, Mathias FLEURY et Christoph WEIDENBACH : A verified sat solver framework with learn, forget, restart, and incrementality. *In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 4786–4790, 2017.
- [BL09] Sandrine BLAZY et Xavier LEROY : Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.
- [CDG12] Matthieu CARLIER, Catherine DUBOIS et Arnaud GOTLIEB : A certified constraint solver over finite domains. *In* Dimitra GIANNAKOPOULOU et Dominique MÉRY, éditeurs : *FM 2012 : Formal Methods*, pages 116–131, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [CS23] Arthur CORRENSON et Dominic STEINHÖFEL : Engineering a formally verified automated bug finder. *In* Satish CHANDRA, Kelly BLINCOE et Paolo TONELLA, éditeurs : *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1165–1176. ACM, 2023.
- [dBB19] Frank S. de BOER et Marcello BONSANGUE : On the nature of symbolic execution. *In* Maurice H. ter BEEK, Annabelle MCIVER et José N. OLIVEIRA, éditeurs : *Formal Methods – The Next 30 Years*, pages 64–80, Cham, 2019. Springer International Publishing.
- [DKFW10] Klaus DRÄGER, Andrey KUPRIYANOV, Bernd FINKBEINER et Heike WEHRHEIM : Slab : A certifying model checker for infinite-state concurrent systems. *In* Javier ESPARZA et Rupak MAJUMDAR, éditeurs : *Tools and*

- Algorithms for the Construction and Analysis of Systems*, pages 271–274, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [dMB08] Leonardo de MOURA et Nikolaj BJØRNER : Z3 : An efficient smt solver. In C. R. RAMAKRISHNAN et Jakob REHOF, éditeurs : *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [ELN⁺13] Javier ESPARZA, Peter LAMMICH, René NEUMANN, Tobias NIPKOW, Alexander SCHIMPF et Jan-Georg SMAUS : A fully verified executable ltl model checker. In Natasha SHARYGINA et Helmut VEITH, éditeurs : *Computer Aided Verification*, pages 463–478, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [FMEH20] Andrea FIORALDI, Dominik MAIER, Heiko EISSFELDT et Marc HEUSE : Afl++ : Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT’20, USA, 2020. USENIX Association.
- [FSMAG20] José FRAGOSO SANTOS, Petar MAKSIMOVIĆ, Sacha-Élie AYOUN et Philippa GARDNER : Gillian, part i : A multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 927–942, New York, NY, USA, 2020. Association for Computing Machinery.
- [God05] Patrice GODEFROID : The soundness of bugs is what matters (position statement). In *BUGS’2005 (PLDI’2005 Workshop on the Evaluation of Software Defect Detection Tools)*, 2005.
- [JLB⁺15] Jacques-Henri JOURDAN, Vincent LAPORTE, Sandrine BLAZY, Xavier LEROY et David PICHARDIE : A formally-verified c static analyzer. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’15, page 247–259, New York, NY, USA, 2015. Association for Computing Machinery.
- [Kin76] James C. KING : Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.
- [LBK⁺16] Xavier LEROY, Sandrine BLAZY, Daniel KÄSTNER, Bernhard SCHOMMER, Markus PISTER et Christian FERDINAND : Compcert – a formally verified optimizing compiler. In *ERTS 2016 : Embedded Real Time Software and Systems*. SEE, 2016.
- [Les11] Stephane LESCUYER : *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. Theses, Université Paris Sud - Paris XI, janvier 2011.
- [Let08] Pierre LETOUZEY : Extraction in coq : An overview. In Arnold BECKMANN, Costas DIMITRACOPOULOS et Benedikt LÖWE, éditeurs : *Logic and Theory of Algorithms*, pages 359–369, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [Nam01] Kedar S. NAMJOSHI : Certifying model checkers. In Gérard BERRY, Hubert COMON et Alain FINKEL, éditeurs : *Computer Aided Verification*, pages 2–13, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- [O’H19] Peter W. O’HEARN : Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.
- [PHD⁺14] Zoe PARASKEVOPOULOU, Catalin HRITCU, Maxime DÉNÈS, Leonidas LAMPROPOULOS et Benjamin C. PIERCE : A coq framework for verified property-based testing (extended abstract). 2014.

Correct tout seul, sûr à plusieurs

Clément Allain et Gabriel Scherer

INRIA

Le modèle mémoire concurrent de OCAML 5 est conçu pour garantir la sûreté du typage et la sûreté mémoire *même* en présence de courses critiques (*data races*) entre fils d'exécution. Ce choix de conception impose des choix techniques précis dans l'implémentation du compilateur et de l'environnement d'exécution du langage. Mais il impose aussi, et c'est le sujet du présent article, des changements dans la façon d'écrire les *programmes utilisateurs* qui utilisent des constructions non sûres comme `Array.unsafe_get` et `Array.unsafe_set` pour des raisons de performance. Même si une structure de données est conçue pour être utilisée de façon séquentielle uniquement, le documente et ne fournit aucune garantie sur un usage concurrent incorrect, elle doit faire les bons choix d'implémentation pour préserver la sûreté mémoire même dans ce cas d'usage incorrect. Des implémentations parfaitement correctes pour OCAML 4 deviennent invalides en OCAML 5.

Nous présentons ici une partie de l'implémentation du module `Dynarray` de tableaux redimensionnables qui a été intégré à la bibliothèque standard de OCAML 5.2. Nous expliquons comment plusieurs choix d'implémentation habituels chez les utilisateurs experts du langage deviennent invalides dans un contexte concurrent. Nous présentons une façon de raisonner sur ces implémentations («correct tout seul, sûr à plusieurs»), comme un modèle mental informel que nous avons utilisé pour écrire ce code, mais aussi dans une formalisation COQ utilisant la logique de séparation IRIS. Nous avons développé et vérifié le noyau de `Dynarray` en IRIS, et montrons comment organiser les spécifications pour exprimer et vérifier de façon simple et claire cette nouvelle propriété de correction.

1 Introduction

Le drame Un tableau OCAML (`'a array`) a une taille fixe qui ne change pas pendant l'exécution du programme. On a souvent besoin d'un tableau redimensionnable, auquel il est possible d'ajouter ou d'enlever des éléments.

La représentation classique d'un tableau redimensionnable dont la taille est actuellement n utilise un enregistrement contenant deux champs modifiables :

- Un *tableau support* `arr` : `'a array`, de taille au moins n , contenant tous les éléments du tableau redimensionnable et un peu d'espace libre pour ajouter de nouveaux éléments.
- Un entier `length` ou `position` qui stocke la valeur actuelle de n ou, de façon équivalente, la position où il faudra écrire le prochain élément ajouté au tableau.

```
type 'a dynarray = {  
  mutable arr: 'a array;  
  mutable length: int;  
}
```

Nous appelons *capacité* la taille du support, qui doit être au moins égale à la *taille* actuelle du tableau redimensionnable.

```
let capacity (a: 'a dynarray) : int = Array.length a.arr
let length (a: 'a dynarray) : int = a.length
```

Ajouter des élément utilise une fonction `ensure_capacity` : un appel à `ensure_capacity a k` s'assure que la capacité de `a` est au moins égale à `k`. Si cette condition n'est pas vérifiée, la fonction fait une modification en place en remplaçant `arr` par un nouveau tableau de taille suffisante – typiquement en multipliant sa taille par une constante, ce qui permet d'obtenir une bonne complexité amortie.

Ajouter un élément en fin de tableau demande la capacité pour 1 élément de plus :

```
let add_last a v =
  let len = a.length in
  ensure_capacity a (len + 1);
  Array.unsafe_set a.arr len v;
  a.length <- len + 1
```

Le code de `add_last` utilise la fonction `Array.unsafe_set` pour écrire directement dans le tableau sans tester que l'indice est dans les bornes, ce qui serait redondant avec les tests déjà effectués dans `ensure_extra_capacity`.

Ce code est parfaitement correct en OCAML 4, et parfaitement faux en OCAML 5. En effet, un utilisateur ou une utilisatrice pourrait utiliser d'autres fonctions qui modifient le tableau `a.arr` en parallèle avec l'appel à `add_last`. Si cette modification est observée entre l'appel à `ensure_capacity` et l'appel à `unsafe_set`, le tableau `a.arr` peut changer et devenir un tableau plus petit – par exemple un appel à `Dynarray.clear` qui met la taille à 0 et libère la mémoire du tableau de support précédent. `unsafe_set` casse alors la sûreté mémoire (et donc de typage) du langage.

C'est frustrant puisque cette structure de données *séquentielle*, pas concurrente, ne doit pas être utilisée en parallèle sans synchronisation. Du code qui ferait de tels accès est buggé, il ne respecte pas la spécification de la bibliothèque `Dynarray`. Mais comme on ne peut pas empêcher ce bug par typage, il faut gérer ce cas en garantissant au moins la sûreté mémoire. Le sujet de cet article n'est pas l'implémentation d'un tableau dynamique concurrent, mais d'un tableau dynamique *séquentiel* qui reste «sûr à plusieurs».

Par exemple, un correctif possible pour `add_last` serait la version suivante :

```
let add_last a v =
  let len = a.length in
  let arr = with_capacity a (len + 1) in
  Array.unsafe_set arr len v;
  a.length <- len + 1
```

avec une fonction `with_capacity a k` qui modifie `a` en place si nécessaire, mais qui en plus renvoie le tableau de support de taille au moins `k` – le tableau en entrée ou le nouveau tableau agrandi. L'écriture non sûre se fait alors sur un tableau dont on sait qu'il est de taille au moins `k`. Cette fonction n'est toujours pas correcte dans un contexte concurrent, mais elle est sûre. (Écrire une version concurrente serait utile dans plus de contextes, mais le code serait aussi sensiblement plus lent car les opérations de synchronisation coûtent plus cher.)

Nos questions Comment écrire des structures de données séquentielles en OCAML 5 qui ne cassent pas les garanties de sûreté dans un cadre concurrent ? Comment relire ce code pour s'assurer qu'il est correct ? Pouvons-nous formaliser précisément la situation pour nous assurer que les réponses aux questions précédentes sont correctes ?

(Question bonus : combien d'implémentations malines et rapides de structures de données OCAML sont complètement non-sûres en OCAML 5 à cause de ce genre de problèmes ?)

Contributions Les contributions présentées dans cet article sont les suivantes :

- Une implémentation du module `Dynarray` intégrée dans la bibliothèque standard OCAML. Nous ne décrivons pas l'ensemble de l'interface (qui vient en grande partie d'un travail amont de Simon Cruanes), mais nous concentrons sur le problème de la sûreté dans un contexte concurrent. Nous parlons aussi, moins en détail, de deux autres problèmes de conception délicats : l'emboîtement (*boxing*) des éléments du tableau et l'invalidation des itérateurs.
- Une formalisation dans la logique de séparation IRIS du critère de correction qui a guidé l'écriture du code et sa relecture, «correct tout seul, sûr à plusieurs», et une implémentation vérifiée d'un petit noyau de `Dynarray` qui établit ce critère de correction.

2 Dynarray dans OCaml 5

2.1 Usages

Il existe déjà de nombreuses implémentations OCAML de tableaux dynamiques, comme des bibliothèques séparées ou même cachées dans des projets plus importants. Nos deux implémentations de référence sont `Vector`¹ de Jean-Christophe Filliâtre et le module `CCVector` de la bibliothèque `containers` de Simon Cruanes. À l'heure où nous écrivons ces lignes, au début octobre 2023, les implémentations de `Vector` et `CCVector` sont d'ailleurs incorrectes en OCAML 5 : elles utilisent par exemple la version non-sûre de `add_last` que nous avons discutée en introduction.

Un tableau redimensionnable est une structure de données très polyvalente. On peut la voir comme un tableau dont le nombre d'éléments change au cours du temps, mais aussi comme une pile (ou une liste modifiée en place) qui permet en plus un accès en position arbitraire.

Dans certains idiomes de programmation très impératifs, les tableaux dynamiques remplacent les listes (persistantes) comme structure de données de base pour accumuler des éléments, itérer dessus, etc.

Pour des exemples plus spécifiques, c'est par exemple une bonne structure de données pour implémenter la trace (*trail*) d'un prouveur automatique (ou d'autres structures de journal avec *rebroussement* (*backtracking*), pour pouvoir empiler et dépiler efficacement des décisions, mais aussi remonter inspecter des décisions sans les dépiler. On peut aussi s'en servir pour stocker l'ensemble des clauses apprises, etc. Les prouveurs Zenon et Zipperposition, par exemple, font un usage très intensif de tableaux dynamiques.

C'est aussi une structure de données utilisée pour implémenter de nombreux algorithmes, par exemple pour stocker une *heap* (un arbre binaire de recherche stocké dans un tableau dynamique) pour implémenter le tri par tas ou une file de priorité.

2.2 API

Cette section propose un tour rapide de l'API de `Dynarray` telle que proposée dans la bibliothèque standard d'OCAML 5.2. On ne montre pas toutes les fonctions, seulement les exemples représentatifs de chaque classe de fonction. Nous donnons seulement le nom et le type de chaque fonction, laissant au lecteur le loisir d'inférer leur comportement – ou de lire la documentation rédigée avec amour.

Les fonctions de base d'un tableau dynamique sont similaires à celles d'un tableau de taille fixe, par exemple :

```
val create : unit -> 'a t
val make : int -> 'a -> 'a t
```

1. Le nom «vector» pour les tableaux dynamiques est fameusement utilisé par les bibliothèques standard C++ (`std::vector`) et RUST ().

```
val init : int -> (int -> 'a) -> 'a t
```

```
val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> unit
```

```
val length : 'a t -> int
```

Il est aussi possible d'utiliser le tableau comme une pile, en ajoutant et enlevant des éléments en fin de tableau, par exemple :

```
val get_last : 'a t -> 'a
val add_last : 'a t -> 'a -> unit
val pop_last_opt : 'a t -> 'a option
```

Cette fonction d'addition en fin de tableau se généralise à des fonctions pour ajouter diverses structures de données – permettant aux tableaux dynamique de jouer le rôle de *buffer* où l'on verse des données de formats variés. Par exemple :

```
val append_array : 'a t -> 'a array -> unit
val append_list : 'a t -> 'a list -> unit
val append : 'a t -> 'a t -> unit
val append_seq : 'a t -> 'a Seq.t -> unit
```

Un ensemble de fonctions assez riche est fourni pour itérer sur les éléments, par exemple :

```
val iteri : (int -> 'a -> unit) -> 'a t -> unit
val mapi : (int -> 'a -> 'b) -> 'a t -> 'b t
```

```
val fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a t -> 'acc
val fold_right : ('a -> 'acc -> 'acc) -> 'a t -> 'acc -> 'acc
```

```
val exists : ('a -> bool) -> 'a t -> bool
val for_all : ('a -> bool) -> 'a t -> bool
```

```
val filter : ('a -> bool) -> 'a t -> 'a t
val filter_map : ('a -> 'b option) -> 'a t -> 'b t
```

On peut convertir entre les tableaux dynamiques et différentes structures de données :

```
val of_array : 'a array -> 'a t
val to_array : 'a t -> 'a array
```

```
val of_list : 'a list -> 'a t
val to_list : 'a t -> 'a list
```

```
val of_seq : 'a Seq.t -> 'a t
val to_seq : 'a t -> 'a Seq.t
```

Enfin, un ensemble de fonctions plus avancées permet de jouer avec la taille du tableau de support, quand un expert veut faire mieux que la politique de redimensionnement par défaut pour un cas d'usage particulier.

```
val capacity : 'a t -> int
val ensure_capacity : 'a t -> int -> unit
```

```
val fit_capacity : 'a t -> unit
val set_capacity : 'a t -> int -> unit
```

Nous avons déjà parlé de la fonction `ensure_capacity a n`, qui garantit une capacité d'au moins `n` pour le tableau dynamique `a`. Elle est utile pour éviter les redimensionnements inutiles si on connaît à l'avance la taille finale qu'aura le tableau.

La fonction `fit_capacity` réduit le tableau de support du tableau dynamique pour que sa taille soit exactement la taille nécessaire et pas plus. Elle est utile pour éviter de conserver un espace mémoire supplémentaire si l'on sait statiquement que le tableau ne va plus recevoir de nouveaux éléments. (Notre implémentation ne réduit pas implicitement la taille du tableau de support quand le nombre d'élément diminue, donc ces fonctions pour réduire explicitement sa taille sont parfois utiles.)

La fonction `set_capacity a n` réalloue un tableau de support de capacité exactement `n`. On peut s'en servir comme une version plus précise de `ensure_capacity` pour agrandir le tableau, ou au contraire pour réduire un tableau en supprimant certains éléments. En particulier, `set_capacity a 0` libère tout l'espace mémoire occupé par un tableau dynamique.

2.3 Choix de conception : vivacité mémoire

Un choix de conception est le comportement des fonctions qui retirent des éléments au tableau dynamique. Une implémentation naturelle serait la suivante :

```
let pop_last_opt a =
  let n = a.length in
  if n = 0 then None
  else begin
    let v = a.arr.(n - 1) in
    a.length <- n - 1;
    Some v
  end
```

Mais cette version a le défaut que la valeur `v` en fin de tableau, bien que retirée du tableau tel que vu par l'utilisateur, est toujours accessible dans la mémoire du tableau de support, et sera conservée en vie par le glaneur de cellules.

Notre implémentation garantit une propriété de vivacité mémoire minimale : les valeurs accessibles dans le tableau de support `a.arr` sont exactement les valeurs des éléments aux positions de 0 à `a.length - 1`. En particulier, aucune valeur initialement fournie par l'utilisateur n'est gardée en vie indépendamment par le tableau de support dans l'espace «vide» après la fin du tableau.

Pour obtenir cette garantie il est nécessaire d'effectuer des écritures supplémentaires pour effacer les valeurs retirées du tableau. La question «avec quoi écraser ces valeurs?» nous amène à un deuxième choix de conception un peu délicat.

2.4 Choix de conception : la valeur du vide

Si le tableau de support d'un `'a dynarray` est de type `'a array`, pour que `pop_last_opt` préserve une vivacité minimale il faut «effacer» le dernier élément du tableau en y mettant une autre valeur de type `'a`. Quelle valeur choisir pour le vide ?

2.4.1 Un des éléments visibles

On peut utiliser un des éléments du tableau, par exemple celui en position 0 – à condition de remplacer le tableau de support par un tableau vide quand il n'a plus d'éléments. Mais ce choix demande de parcourir tout le tableau de support à chaque fois que l'élément en position 0 est modifié, donnant une mauvaise surprise pour la complexité de `set`.

2.4.2 Un élément fourni explicitement

On peut demander à l'utilisateur de fournir une valeur «par défaut» au moment de la création du tableau, stockée comme un troisième champ de l'enregistrement, qui est documentée comme restant vivante pendant toute la durée de vie du tableau. On relâche

un peu la promesse de vivacité minimale, mais de façon explicite et contrôlée. C'est le choix de la bibliothèque `Vector`. Nous n'avons pas fait ce choix, car il alourdit l'API, toutes les fonctions de création doivent demander une valeur supplémentaire. Si on veut implémenter `map : ('a -> 'b) -> 'a dynarray -> 'b dynarray` par exemple, faut-il dériver un élément par défaut dans 'b en appelant `f` sur l'élément par défaut dans 'a, ou demander un nouvel élément par défaut ? Enfin, certains usages de `Dynarray` impliquent des types d'éléments complexes. Il peut être pénible pour l'utilisateur de construire un élément par défaut.

2.4.3 Une valeur invalide

L'implémentation proposée par Simon Cruanes utilisait une valeur non-sûre de la forme `(Obj.magic () : 'a)` comme élément par défaut.²

Le problème de cette approche est qu'elle est incorrecte en OCAML 5. Considérons une implémentation courante de `get` :

```
let get a i =
  if i >= a.length then invalid_arg "Dynarray.get";
  a.arr.(i)
```

En présence d'accès concurrents imprévus, il n'y a aucune garantie que la taille `a.length` soit inchangée au moment où on accède à `a.arr`; un appel concurrent à `pop_last_opt` pourrait avoir modifié `a.length` et mis un élément par défaut en position `i`, qui peut être alors renvoyé par la fonction `get`. Si l'on utilise un élément par défaut qui n'est pas une valeur valide à ce type, on peut casser la sûreté du typage et donc la sûreté mémoire.³

Une façon de contourner ce problème serait de tester, dans `get`, si la valeur lue est la valeur par défaut, et d'échouer dans ce cas.

```
let get a i =
  if i >= a.length then invalid_arg "Dynarray.get";
  let v = a.arr.(i) in
  if is_dummy_value a v then invalid_arg "Dynarray.get";
  v
```

Ce n'est pas possible si l'on utilise `()` comme valeur par défaut, puisque cette valeur pourrait avoir été insérée par l'utilisateur. Quand nous avons fait remonter ce problème de concurrence avec l'implémentation de Simon Cruanes, une longue discussion a fait émerger deux choix possibles :

1. On peut utiliser une valeur allouée dynamiquement et jamais montrée à l'utilisateur, par exemple `ref ()`, stockée comme une constante (privée) du module ou dans un champ du tableau.
2. On pourrait aussi utiliser des valeurs magiques permises par le runtime mais qui ne représentent aucune valeur OCAML source valide.

2.4.4 Représentation emboîtante

La solution la plus simple, et celle que nous avons finalement adoptée dans notre proposition d'implémentation, est d'utiliser un type différent de 'a `array` pour le tableau de support. Si on utilise 'a `option array`, il suffit de mettre `None` dans les cases non utilisées du tableau de support.

Il est en fait possible d'utiliser une légère variante du type 'a `option` avec de meilleures propriétés pour cet usage. C'est ce que nous faisons dans notre implémentation :

2. Et pour les nombres flottants ? Il faut complexifier un peu le code pour les gérer de façon sûre, en raison des optimisations de représentation des tableaux de flottants.

3. Par contraste, avec les approches précédentes, on peut renvoyer une "valeur vide" dans ce cas. Les valeurs utilisées pour les cases vides sont valides au type 'a donc il n'y a pas de problème de sûreté.

```

type 'a slot =
| Empty
| Elem of { mutable v: 'a }

L'intérêt d'utiliser un 'a slot array plutôt qu'un 'a option array est qu'il n'y a pas
besoin d'allouer pour implémenter set :
(* version avec [arr : 'a option array] *)
let set a i v =
  a.arr.(i) <- Some v (* allocation *)

(* version avec [arr : 'a slot array] *)
let set a i v =
  match a.arr.(i) with
  | Empty -> invalid_arg "Dynarray.set"
  | Elem e -> e.v <- v (* modification en place *)

```

Il faut toujours allouer ce constructeur `Elem` quand on ajoute de nouveaux éléments au tableau, mais (contrairement à `'a option`) pas quand on s'en sert comme un tableau de taille fixe. (La modification en place a aussi un coût, supérieur parfois au coût de l'allocation, mais il y a de toute façon une écriture dans les deux versions.)

Nos utilisateurs potentiels étaient très inquiets de l'impact sur les performances de cette représentation *emboîtée* par rapport à `'a array`. En particulier, cette représentation est moins compacte en mémoire – elle utilise un mot de plus par élément et l'accès à un élément contient une indirection, ce qui pourrait coûter cher sur des cas d'usages intensifs brassant beaucoup de mémoire en raison d'une moins bonne localité mémoire. Ceci dit, le constructeur `Elem` est souvent alloué en même temps ou presque que la valeur qu'on ajoute au tableau, et les GC générationnels sont bons pour transformer la localité temporelle en localité spatiale.

Selon nos mesures, on observe un ralentissement finalement assez faible. Nous avons fait l'expérience de modifier un SAT-solver, qui faisait un usage critique de tableaux dynamiques, pour utiliser cette représentation moins efficace. Dans la plupart des cas les performances n'ont pas vraiment changé, nous avons trouvé des ralentissements de 10-15% pour certains fichiers de test, et 25% sur un seul cas de test.⁴

2.5 Choix de conception : invalidation des itérateurs

Le dernier choix de conception qui a fait couler beaucoup d'encre pour `Dynarray` est l'invalidation des itérateurs. Comment réagir si la taille du tableau est modifiée *pendant* qu'une fonction est en train d'itérer sur le tableau (par exemple `iter`, `map`, `exists`, mais aussi `to_list` ou `to_seq`) ? Une telle modification peut venir d'un usage concurrent de la structure, mais aussi d'une situation de réentrance – par exemple si l'itérateur appelle une fonction utilisateur sur chaque élément, qui ajoute ou retire des éléments au tableau.

Plusieurs choix s'offrent à nous, par exemple :

- Donner le droit à la fonction d'itération d'observer ou non chaque modification. Une implémentation conforme pourrait prendre en compte certains ajouts et suppressions mais aussi continuer à «voir» des éléments supprimés ou «manquer» des éléments ajoutés.
- Imposer une sémantique spécifique dans ce cas, par exemple demander à ce que l'itération observe toutes les modifications, ou au contraire soit équivalente à une itération sur le tableau tel qu'il était au départ.
- Donner le droit à la fonction d'itération d'échouer si elle observe une modification indépendante.
- Imposer que la fonction d'itération échoue si elle observe une modification indépendante.

4. <https://github.com/ocaml/ocaml/pull/11882#issuecomment-1405867624>

(La notion de «modification» ici est une modification de structure du tableau, c'est-à-dire de sa taille, en ajoutant ou retirant des éléments. Il y a aussi les modifications d'éléments (sans changer de structure), obtenues avec `Dynarray.set`. On peut imaginer de faire des choix différents ci-dessus pour les modifications de structure et les modifications d'éléments.)

Permettre plus de comportements a l'avantage de permettre des implémentations plus efficaces, mais augmente les chances que l'utilisateur ou utilisatrice introduise des bugs subtils liés à des comportements permis mais non observés pendant ses tests. À l'inverse, imposer un comportement précis simplifie la programmation mais peut être plus coûteux.

Une fois la discussion sur le boxing tassée, une très large part de la discussion de notre travail sur `Dynarray` a été consacrée à ces questions d'invalidation des itérateurs. Nous avons soumis plusieurs implémentations faisant des choix distincts, par exemple une implémentation maximisant les performances avec une spécification très permissive, ou une implémentation qui observe toutes les modifications (de structure et d'éléments) dans le cadre séquentiel sans échouer.

Le consensus auquel la discussion a abouti correspond à une position poussée par Simon Cruanes et Guillaume Munch-Maccagnoni :

- À ce jour il n'y a pas de cas d'usage clairement identifié pour l'usage d'itérateurs en même temps que des modifications de structure, et cette pratique cache habituellement une erreur de programmation.
- Il vaut donc mieux interdire les modifications de structure pendant l'itération, et échouer systématiquement.
- Un utilisateur ou une utilisatrice souhaitant prendre en compte des modifications de structure pendant l'itération peut implémenter ses propres itérateurs, et/ou devrait utiliser une structure de données conçue dès le départ pour un usage concurrent.

La spécification choisie est la suivante : effectuer une modification de structure pendant une itération est une erreur de programmation, peut lever une exception `Invalid_argument`, et l'implémentation fera des efforts raisonnables pour détecter ces cas et échouer.

Les itérateurs de notre implémentation échouent s'ils atteignent un élément qui a été retiré du tableau après le début de l'itération, et vérifient en fin d'itération que la taille du tableau n'a pas changé – donc ils échouent si des éléments ont été rajoutés. Le coût de ces vérifications est très faible, négligeable dans du code utilisateur typique. Cette approche peut manquer des modifications (elle peut échouer à échouer), par exemple si la fonction utilisateur ajoute temporairement des éléments avant d'en retirer le même nombre, ou en général effectue des modifications de structure qui préservent le nombre d'éléments. Voici notre implémentation de `Dynarray.iteri` :

```
let iteri k a =
  let {arr; length} = a in
  check_valid_length arr length;
  for i = 0 to length - 1 do
    k i (unsafe_get arr i length);
  done;
  check_same_length a length
```

La fonction `check_valid_length` vérifie l'invariant `length <= Array.length arr`, ce qui permet d'éliminer les tests d'accès au tableau support dans le corps de la boucle. La fonction `unsafe_get` fait un accès non sûr au tableau support, mais échoue sur des éléments vides (qui ont été retirés du tableau). Notons que le corps de la boucle utilise `arr` et `length`, le tableau support et la taille au début de la boucle, sans refaire de lecture de champs modifiables qui pourraient observer des modifications de structure concurrentes. Enfin, `check_same_length` vérifie en fin de boucle que la taille du tableau est restée celle de départ. C'est un test de confort ou de correction, qui sert à échouer plus souvent en cas de comportement invalide.

2.5.1 Aller plus loin ?

On pourrait imaginer une implémentation qui échouerait dans tous les cas de modification de structure, en stockant un «numéro de version» incrémenté à chaque modification de structure. Pour une implémentation correcte dans le cadre concurrent il faudrait un numéro de version atomique, et ajouter un incrément atomique à chaque `Dynarray.add_last` aurait un coût significatif. Rendre ce champ non-atomique diminue le coût en performance mais rend l'approche incomplète de nouveau. Cette piste d'amélioration, qui est utilisée dans le module `ArrayList` bibliothèque standard Java, a été suggérée par Guillaume Munch-Maccagnoni. Nous ne l'avons pas encore explorée.

Remarque 1. Au lieu d'écrire à chaque modification et de lire dans les itérateurs, on peut imaginer une version où les itérateurs modifient la structure pour indiquer qu'une itération est en cours (en mettant un drapeau atomique à `true`), et les modifications testent cette situation et échouent dans ce cas.

Cette approche serait plus efficace mais elle a le défaut majeur qu'elle fait des itérateurs des opérations qui modifient la structure de données. En particulier, dans le cadre d'un programme concurrent, un utilisateur pourrait vouloir faire deux itérations en parallèle sans synchronisation, tant que les itérations ne font que lire le tableau. (Avoir une possession unique pendant des phases d'écriture, mais une possession partagée pendant des phases de lecture seule.) Ce cas d'usage est tout à fait valide avec les implémentations habituelles, mais devient invalide si les itérateurs modifient l'état en place. Autrement dit, dans un cadre concurrent ajouter des modifications en place peut changer la spécification d'une fonction, et faire d'un itérateur une fonction modifiante est une très mauvaise idée.

(Le module `Hashtbl` de la bibliothèque standard utilise depuis 2015 des modifications en place pendant l'itération, pour des raisons de performance, et a ce problème.)

2.5.2 Itérateurs externes, générateurs.

Un cas particulier d'itérateur est

```
val to_seq : 'a dynarray -> 'a Seq.t
```

qui est un itérateur «externe» : une séquence de type `'a Seq.t` est énumérée à la demande. Dans un programme séquentiel, il est rare qu'un énumérateur strict comme

```
val to_list : 'a dynarray -> 'a list
```

puisse observer une invalidation – c'est possible en cas d'appels asynchrones (finaliseurs, etc.) causé par une allocation. C'est plus facile pendant l'exécution de

```
val iter : ('a -> unit) -> 'a dynarray -> unit
```

puisque la fonction fournie par l'utilisateur peut modifier le tableau dynamique. Dans le cas de `to_seq`, l'invalidation peut venir après que la fonction a renvoyé une valeur, quand le contexte d'appel forcera cette séquence.

Nous avons convergé vers une API proposant deux versions de `to_seq` : la version de base, nommée `to_seq`, échoue comme les autres itérateurs, si l'accès à un élément de la séquence a lieu après modification du tableau dynamique. Une variante `to_seq_reentrant` n'échoue pas dans ce cas : accéder au i -ème élément de la séquence générée renvoie le i -ème élément du tableau dynamique au moment de l'accès, ou `Empty` si le tableau s'arrête avant.

2.6 Mesures de performance ?

Mesurer précisément les performances d'une telle structure de données est fastidieux et très difficile à la fois. Pour être tout à fait honnêtes, nous ne l'avons pas bien fait. Nous rencontrons au moins les difficultés suivantes :

- Les microbenchmarks magnifient les différences de performance entre opérations très rapides, qui constituent une fraction très faible du temps total de calcul dans la plupart des applications finales.

- Au contraire, les microbenchmarks échouent typiquement à mesurer les effets de localité mémoire qui s’observent sur des programmes qui, à la fois, font un usage intense de gros tableaux dynamiques, et manipulent de gros volumes de données par ailleurs.

Pour donner une idée d’ordres de grandeur sur le premier point, voici les résultats d’un microbenchmark des opérations `get` et `set` sur une structure à quelques milliers d’éléments :

- `Dynarray` est environ 2x plus lent que `Array` (un test d’indice en plus à chaque accès).
- `Hashtbl` est environ 42x plus lent que `Array` (un calcul d’empreinte à chaque accès).
- `Map` est environ 78x plus lent que `Array` (un parcours d’arbre équilibré à chaque accès).

Les opérations déterminantes pour `Dynarray` nous semblent être `get` et `set`, d’une part, et `add_last` et `pop_last` d’autre part. Les autres opérations sont moins fréquentes et rarement dans les parties critiques pour les performances d’un programme.

Une fois ces pincettes prises, voici les ordres de grandeurs tirés de microbenchmarks effectués pendant la préparation de cette PR, en décembre-janvier 2023 :

- Emboîter les éléments n’apporte pas de surcoût mesurable sur `get` et `set`.
- Emboîter les éléments apporte un petit surcoût sur des séquences de `add_last` seules, qui peuvent apparaître dans des cas de construction itérative d’une collection finale : 1.6x plus lent sur des cas utilisant `ensure_capacity` pour limiter les redimensionnements, et 2x sinon.

(Dans le cas particulier de `append_array` on observe des différences plus marquées de 3x-4x, parce qu’utiliser la représentation déboîtée `'a array` permet d’implémenter `append_array` en utilisant `Array.blit` directement.)

- Les différences de performance liées aux différents choix de conception pour l’invalidation des itérateurs sont très faibles. Entre notre version optimisée pour les performances et notre version optimisée pour échouer le moins souvent, les différences étaient de l’ordre de 1.2-1.3x dans certains cas, 1.6x dans les cas les plus favorables à la version optimisée pour l’efficacité.

Globalement, nos tests suggèrent que les implémentations de la seconde PR de Gabriel Scherer ont des performances comparables à celle de la première PR par Simon Cruanes, malgré des évolutions notables sur les choix de spécification et d’implémentation.

2.7 Histoire de la Pull Request (PR) pour OCaml 5

Plusieurs bibliothèques de tableaux dynamiques existent en OCAML, mais une structure de données devient encore plus facile d’accès et plus utilisée quand elle intègre la bibliothèque standard. Le projet `extlib`, qui voulait proposer une extension communautaire largement utilisée à la bibliothèque standard minimale, avait déjà une implémentation de `Dynarray`, qui semble provenir de code écrit par Brian Hurt en 2003 ; ce code a été hérité par le projet `Batteries`, lui-même une inspiration de la bibliothèque `containers` de Simon Cruanes contenant `CCVector`.

Faire entrer du code dans la bibliothèque standard OCAML est cependant une tâche difficile : en cas d’hésitation sur la réponse à une question de conception, l’équipe qui la maintient décide souvent de bloquer les changements proposés plutôt que de trancher de façon arbitraire. Ajouter une ou deux fonctions est difficile, ajouter un nouveau module relève, en toute modestie, du tour de force.

2.7.1 Première PR par Simon Cruanes

En avril 2022, Simon Cruanes a décidé de tenter l’impossible et de faire entrer un module de tableaux extensibles dans la bibliothèque standard. Une réunion de discussion avec Florian Angeletti et Gabriel Scherer a permis de dégager une proposition minimale⁵. Des trois problèmes de conception que nous avons mentionné (concurrence, valeurs vides, invalidation des itérateurs), nous avons seulement identifié celui des valeurs vides, pour lequel Simon

5. MVP, Mise-en-oeuvre Vaguement Potable ?

Cruanes proposait une approche à la `Obj.magic ()`, éventuellement implémentée en C pour éviter toute inquiétude sur le cas des tableaux de flottants. La proposition contenait une API assez courte, inspirée de `CCVector` et aussi notablement des choix de conception de la bibliothèque de RUST. Simon Cruanes était désigné volontaire pour l'implémenter.

De façon indépendante, en mai 2022 les tests aléatoires de Jan Midtgaard et l'analyse détaillée de Jon Ludlam ont découvert une erreur de concurrence dans le code de `Buffer`, rapportée par David Allsopp en [#11279](#) et corrigée par Florian Angeletti en [#11742](#). On peut voir `Buffer` comme une version simplifiée de `Dynarray`, spécialisée au type `char` et sans accès à une position arbitraire. C'est le moment où les développeurs de la bibliothèque standard se sont rendu compte des problèmes d'interaction entre concurrence et accès non-sûr dans du code existant. Notons que la correction du problème n'a pas dégradé les performances – c'est même le contraire, l'attention portée aux micro-optimisations pendant sa correction a permis d'améliorer légèrement les performances de `Buffer`.

En septembre 2022, Simon Cruanes envoie sa proposition d'implémentation de tableaux dynamiques, [#11563](#). L'implémentation de départ utilise `Obj.magic` pour créer des valeurs vides, mais Simon Cruanes identifie le problème de la concurrence et change l'interface et l'implémentation pour mélanger des arguments explicites (dans `ensure_capacity`) et l'usage des éléments existants du tableau, abandonnant la propriété de vivacité minimale. Une longue discussion (92 messages) s'ensuit en septembre et octobre 2022, avec en particulier un débat sur la valeur du vide à choisir (valeur privée ou valeur invalide?), et une alerte de Guillaume Munch-Maccagnoni sur le problème d'invalidation des itérateurs. La valeur du vide divise les participants et la situation semble bloquée.

2.7.2 Deuxième PR par Gabriel Scherer

Gabriel Scherer propose en janvier 2023 une nouvelle implémentation de `Dynarray` par-dessus le travail de Simon Cruanes, modifiée pour «emboîter» les éléments dans un type option modifiable que nous avons décrit en Section 2.4.4. Le premier but est de convaincre les participants qu'une représentation emboîtante, bien que légèrement moins efficace, peut permettre d'avancer en remettant à plus tard le choix d'une approche non-sûre optimisée. Ceci est permis par des mesures de performances qui indiquent un coût relativement faible pour l'emboîtement.

Cette PR pointe aussi du doigt le problème d'invalidation des itérateurs. Elle n'indique pas de préférence entre les possibilités évoquées en Section 2.5, et propose deux alternatives, qui n'échouent pas en cas de modification concurrentes. La première version permet à l'implémentation d'observer ou non les modifications, maximisant les performances, l'autre impose une spécification simple et plutôt naturelle (chaque accès opère sur la version la plus récente du tableau observée par le fil d'exécution), légèrement moins efficace.

La version actuelle de la PR contient 46 fonctions, une interface de 641 lignes (surtout des commentaires) et une implémentation de 791 lignes. Un effort particulier est fait sur la qualité de la documentation du module, et sur des messages d'erreur les plus clairs et informatifs possibles.

La très longue discussion (337 messages) peut se résumer par les grands moments suivants :

- Janvier 2023 : on atteint un consensus sur le fait qu'une implémentation emboîtante est acceptable comme un premier pas.
- Janvier 2023 : Guillaume Munch-Maccagnoni et Simon Cruanes insistent pour obtenir une spécification où l'invalidation des itérateurs échoue et est considérée comme une erreur de programmation, au lieu que le code s'accommode de ces situations sans échouer. Une troisième version de l'implémentation est nécessaire.
- Mars 2023 : Daniel Bünzli relit l'interface et sa documentation, qui évolue nettement en réponse à ses commentaires.
- Avril 2023 : Clément Allain relit l'implémentation pour vérifier (informellement) sa correction. C'est de cette interaction qu'est née le présent article. C'est Armaël Guéneau qui a mis le doigt sur l'idée de raisonner avec deux spécifications superposées : une

spécification forte et précise dans le cas séquentiel, cherchant à établir la correction fonctionnelle ; une spécification plus faible dans le cas concurrent, suffisant à garantir la sûreté mémoire.

- Mai 2023 : relecture d'ensemble par Damien Doligez qui «approuve» la PR. (Pour les modifications à la bibliothèque standard, les règles de développement OCAML demandent que deux mainteneurs différents approuvent chaque changement.)
- Juin 2023 : relecture de la documentation et des messages d'erreur par Wiktor Kuchta.
- Juillet 2023 : relecture des questions de concurrence, réentrance et du choix des exceptions par Guillaume Munch-Maccagnoni.
- Mai et juillet : discussions sur `to_seq` (le cas des itérateurs externes, voir Section 2.5.2), consensus final sur une version avec une variante réentrante explicite.
- Septembre 2023 : relecture d'ensemble par Florian Angeletti qui «approuve» la PR, obtenant le quota de deux approbation de mainteneurs.

3 Raisonner sur Dynarray

L'examen de la PR décrite précédemment nous a conduit à adopter et vérifier (d'abord informellement) les critères de correction suivants :

1) Correction fonctionnelle. L'implémentation des fonctions de `Dynarray` respecte la spécification (informelle) donnée dans l'API. Autrement dit, chaque fonction se comporte comme attendu *si le programmeur l'utilise lui-même de façon correcte* – purement séquentielle ou concurrente mais synchronisée. Le comportement est indéfini si le programmeur en fait un usage concurrent non synchronisé.

Pour le vérifier, on raisonne dans un *cadre séquentiel* en s'appuyant sur un *invariant fort* affirmant notamment : à tout moment, si ce n'est transitoirement à l'intérieur des fonctions de `Dynarray`, la structure consiste en un enregistrement contenant une taille et un tableau dont la propre taille, appelée capacité, est supérieure ou égale à la première. Informellement :

```
forall (t : 'a dynarray). 0 <= t.length && t.length <= Array.length t.arr
```

2) Sûreté mémoire. L'implémentation de `Dynarray` est sûre. Autrement dit, *dans tous les cas*, usage correct ou non⁶, le programmeur ne peut observer un accès mémoire erroné menant à la terminaison prématurée du programme. Ce deuxième critère est imposé par les garanties dynamiques du langage OCAML, d'ordinaire assurées par typage statique. Il n'est pas immédiat car notre implémentation exploite des fonctions non sûres (`Array.unsafe_get` et `Array.unsafe_set`), c'est-à-dire pour lesquelles il ne suffit pas d'être bien typé pour être sûr. Il faut en plus vérifier que les indices sont compris dans les bornes des tableaux.

Pour le vérifier, on raisonne dans un *cadre concurrent* en s'appuyant sur un *invariant faible* : à tout moment, la structure consiste en un enregistrement contenant une taille *positive*. Informellement :

```
forall (t : 'a dynarray). 0 <= t.length
```

On demande aussi que le tableau support `t.arr` respecte son propre invariant faible : c'est un tableau OCaml bien formé dont les éléments sont bien typés.

En sus de cet examen informel, nous avons mené un travail de vérification formelle dans la logique de séparation IRIS (Jung, Krebbers, Jourdan, Bizjak, Birkedal, and Dreyer, 2018b), entièrement mécanisée dans le système COQ. Notre but était non seulement d'apporter des garanties supplémentaires mais aussi d'ajouter `Dynarray` à un corpus de structures

6. Nous considérons tous les contextes d'utilisation de `Dynarray` qui n'utilisent pas de fonctionnalités non-sûres. Mais on peut aussi étendre le raisonnement à tous les contextes qui sont sûrs contre toute implémentation entièrement sûre de `Dynarray`.

vérifiées utiles à de futurs développements. Nous décrivons ici le sous-ensemble traité et la méthode employée pour vérifier les deux critères. Les preuves mécanisées sont disponibles à https://github.com/clef-men/heap_lang/tree/jfla2024.

3.1 Fragment minimal

Le fragment minimal traité à ce jour se compose des fonctions suivantes :

```
val create : unit -> 'a dynarray
val make : int -> 'a -> 'a dynarray
val length : 'a dynarray -> int
val get : 'a dynarray -> int -> 'a
val set : 'a dynarray -> int -> 'a -> unit
val add_last : 'a dynarray -> 'a -> unit
val pop_last : 'a dynarray -> 'a
val fit_capacity : 'a dynarray -> unit
val reset : 'a dynarray -> unit
```

Ce fragment est suffisant pour parler d'usages concurrents problématiques, en particulier de conflits entre accès non sûrs au tableau support et rétrécissement de ce dernier par `fit_capacity` et `reset`.

Nous avons reproduit l'implémentation OCAML dans le langage HEAPLANG, l'instance canonique de la logique de séparation IRIS. Sa syntaxe et sémantique sont relativement proches de celle d'OCAML. Nous l'utilisons car il est standard dans la communauté IRIS et bien outillé.

Voici, par exemple, à gauche l'implémentation OCAML de `pop_last` et à droite l'implémentation HEAPLANG sans fard, telle qu'écrite en COQ.

<pre>let pop_last a = let {arr; length} = a in check_valid_length length arr; if length = 0 then raise Not_found; let last = length - 1 in match Array.unsafe_get arr last with Empty -> Error.missing_element last length Elem s -> Array.unsafe_set arr last Empty; a.length <- last; s.v</pre>	<pre>Definition dynarray_pop : val := λ: "t", let: "len" := dynarray_length "t" in let: "arr" := dynarray_array "t" in assume ("len" <= array_length "arr") ;; assume (#0 < "len") ;; let: "last" := "len" - #1 in match: array_unsafe_get "arr" "last" with None => diverge #() Some "ref" => array_unsafe_set "arr" "last" &None ;; dynarray_set_size "t" "last" ;; !"ref" end.</pre>
--	---

Les deux implémentations diffèrent en plusieurs points (en plus des préférences de nommage différentes des deux auteurs, `a` ou `t` pour les tableaux dynamiques) :

1) Exceptions. Dans la version OCAML, on lève une exception sur un tableau vide et lorsqu'on détecte un usage erroné. HEAPLANG n'étant pas muni d'exceptions, le programme diverge en bouclant dans ces cas – dans une logique de *correction partielle* comme IRIS, une boucle infinie est trivialement vérifiable. Néanmoins, [de Vilhena and Pottier \(2021\)](#) ont formalisé les effets algébriques et donc les exceptions en IRIS. Nous espérons les ajouter un jour.

2) Représentation mémoire. La représentation des éléments du tableau support est moins efficace en HEAPLANG : le type `'a slot` est en quelque sorte remplacé par le type

'a `ref option`. S'accorder à la représentation OCAML ne pose toutefois pas de difficulté. Nous avons choisi la simplicité.

3) Modèle mémoire faible. Le modèle mémoire de HEAPLANG est fortement consistant alors que celui d'OCAML est faiblement consistant (Dolan, Sivaramakrishnan, and Madhavapeddy, 2018). Le modèle mémoire OCAML a aussi été formalisé en IRIS dans le projet COSMO (Mével, Jourdan, and Pottier, 2020). Il nous apparaît maintenant qu'il serait préférable d'utiliser COSMO plutôt que HEAPLANG, et nous avons commencé à travailler dans cette direction.

4) Module Array. En lieu et place du module standard `Array`, la version HEAPLANG emploie des fonctions `array_*`. Ces fonctions et leurs spécifications ne sont pas axiomatisées. Nous les avons elles-même implémentées en HEAPLANG.

3.2 Correction fonctionnelle – correct tout seul

La vérification de la correction fonctionnelle de `Dynarray` en logique de séparation n'est pas novatrice. De façon classique, on définit une assertion logique `dynarray_model t vs` exprimant la possession unique de l'instance `t` contenant les valeurs `vs`. En COQ, cela ressemble à :

```
Definition dynarray_model t vs : iProp Σ :=
  ∃ l arr slots extra,
  ⌈t = #l⌉ *
  l.[length] ↦ #(length vs) *
  l.[array] ↦ arr * array_model arr (slots ++ replicate extra &&None) *
  [* list] slot; v ∈ slots; vs, slot_model slot v.
```

`iProp Σ` est le type des assertions IRIS. Cette définition décrit un enregistrement à deux champs. Le premier représente la taille de `vs`. Le second consiste en un tableau contenant les slots associés aux valeurs de `vs` suivis de valeurs `&&None` – ce qui permet d'arguer que la structure ne maintient pas ses anciens éléments en vie.

De façon classique toujours, on spécifie les fonctions de `Dynarray` à l'aide de triplets de Hoare. Typiquement, le prédicat `dynarray_model y` apparaît en précondition et postcondition en observant potentiellement un changement de valeurs contenues. Par exemple, les spécifications des fonctions `add_last` et `pop_last` s'écrivent :

```
Lemma dynarray_add_last_spec t vs v :
  {{{
    dynarray_model t vs
  }}}
  dynarray_add_last t v
  {{{
    RET #();
    dynarray_model t (vs ++ [v])
  }}}.

Lemma dynarray_pop_last_spec t vs v :
  {{{
    dynarray_model t (vs ++ [v])
  }}}
  dynarray_pop_last t
  {{{
    RET v;
    dynarray_model t vs
  }}}.
```

La spécification de `dynarray_add_last` se lit ainsi : étant donnée l'assertion `dynarray_model t vs` signifiant la possession unique de `t`, on peut procéder à l'appel et retrouver l'assertion où la liste de valeurs contenues a été augmentée de la valeur `v` donnée à la fonction. La spécification de `dynarray_pop_last` est symétrique.

3.3 Sûreté mémoire – sûr à plusieurs

En pratique, la vérification informelle de la sûreté mémoire n'apparaît pas difficile. Étant donné l'invariant faible déjà évoqué (taille positive, tableau support bien formé), il s'agit de

se convaincre que cet invariant potentiellement combiné à des tests dynamiques implique la validité de l'indice donné à `Array.unsafe_get` ou `Array.unsafe_set`.

Par exemple, dans l'implémentation de `pop_last` donnée plus haut, la bonne formation du tableau support, l'appel à `check_valid_length` et le test `if length = 0` (traduits par deux `assume` en `HEAPLANG`) suffisent à montrer que `last` est un indice valide pour `arr`. Le raisonnement reste local et simple, un humain peut aisément se convaincre de la sûreté.

Ceci étant dit, la question de la formalisation se pose. En particulier, intuitivement, l'invariant est plus faible mais parle de concurrence non synchronisée. Pour l'écrire, il nous faut donc une logique de séparation *concurrente* telle qu'IRIS. La tâche se complexifie encore dans le cas d'un modèle mémoire faible.

En IRIS, précisément, cette question a été posée dans les travaux sur RUSTBELT (Jung, Jourdan, Krebbers, and Dreyer, 2018a; Dang, Jourdan, Kaiser, and Dreyer, 2020; Matsushita, Denis, Jourdan, and Dreyer, 2022). Le problème dans sa forme générale est le suivant : dans un langage fortement et statiquement typé comme OCAML ou RUST, comment encapsuler un fragment – typiquement, une bibliothèque – non sûr.

En RUST, où cela se manifeste par des blocs `unsafe`, la réponse est : on encapsule du code non sûr derrière une interface sûre. C'est le cas, par exemple, de `std::vec`, qui expose la structure `Vec` comprenant des champs privés.

RUSTBELT formalise la chose à l'aide de la notion de *typage sémantique* : on interprète un type comme un prédicat en logique de séparation (relation logique). Pour justifier la sûreté d'un programme composé de parties sûres et de parties non sûres, on raisonne ainsi : 1) Sur le fragment sûr du langage, le typage syntaxique implique le typage sémantique. Les parties sans blocs `unsafe` acceptées par le typeur sont donc sémantiquement bien typées. 2) On peut composer des programmes sémantiquement bien typés pour obtenir un programme lui-même sémantiquement bien typé. 3) Le typage sémantique implique la sûreté (théorème d'adéquation d'IRIS).

Autrement dit, pour vérifier la sûreté d'un programme utilisant des mécanismes non sûrs, on montre – manuellement ou de façon automatisée – que ce programme est sémantiquement bien typé. La combinaison avec tout autre programme sémantiquement bien typé, en particulier syntaxiquement bien typé, est alors sûre. C'est l'approche que nous avons adoptée.

Nous avons choisi pour OCAML une notion de type sémantique plus simple que celle de RUST. Nos types sémantiques sont définis en COQ par la typeclasse `iType` suivante :

```
Class iType (PROP : bi) (τ : val → PROP) := {
  #[global] itype_persistent v :: Persistent (τ v) ;
}.
```

`PROP` peut ici être pris comme l'ensemble des propositions IRIS – c'est en quelque sorte une version plus bas niveau de `iProp Σ`. La partie importante est `itype_persistent`. Elle affirme la persistance du type sémantique τ au sens d'IRIS, c'est-à-dire – pour faire court – le fait qu'une proposition soit toujours vraie, même quand l'état du programme change.

On définit par exemple le type sémantique `opt_type` τ , associé au type syntaxique 'a `option`. Il s'agit d'exprimer qu'une valeur de ce type est soit de la forme `&&None`, soit de la forme `&&Some v` pour une valeur `v` de type τ .

```
Definition opt_type τ '{iType (iProp Σ) τ} t : iProp Σ :=
  「t = &&None」 ∨ ∃ v, 「t = &&Some v」 * τ v.
```

Le type `opt_type` τ ne traite pas tellement de ressources. Le type `reference_type` τ associé à 'a `ref`, en revanche, affirme l'existence d'une cellule mémoire partagée dont la possession est stockée dans un *invariant* IRIS.

```
Definition reference_type τ '{iType (iProp Σ) τ} t : iProp Σ :=
  ∃ (l : loc),
  「t = #l」 *
  inv nroot (
```



```

    ∃ w,
    l ↦ w * τ w
  ).

```

Le point remarquable dans cette définition est qu'on ne connaît pas la valeur contenue dans la cellule. On peut seulement accéder *atomiquement* à cette valeur et apprendre qu'elle est de type τ . On peut donc toujours lire et écrire une valeur de type τ dans la cellule.

Le type `dynarray_type` dirigeant notre vérification est à l'avenant :

Definition `dynarray_type` τ `{iType (iProp Σ) τ } t : iProp Σ :=`

```

  ∃ l,
  ⌈t = #l⌉ *
  inv nroot (
    ∃ len cap arr,
    ⌈0 <= len⌉ *
    l.[length] ↦ #len *
    l.[array] ↦ arr * array_type (slot_type  $\tau$ ) cap arr
  ).

```

C'est exactement la traduction formelle de l'invariant faible énoncé plus tôt : la structure consiste en un enregistrement contenant une taille *positive* et un tableau *bien formé* dont chaque élément constitue un objet valide pour le type `slot_type` τ .

Les spécifications *fortes* pour la correction fonctionnelle, `dynarray_model t vs`, sont alors remplacées par des spécifications *faibles*, `dynarray_type τ v` :

Lemma `dynarray_push_type` τ t v :

```

  {{{
    dynarray_type  $\tau$  t *  $\tau$  v
  }}}
  dynarray_push t v
  {{{ u,
    RET u; unit_type u
  }}}.

```

Lemma `dynarray_pop_type` τ t :

```

  {{{
    dynarray_type  $\tau$  t
  }}}
  dynarray_pop t
  {{{ v,
    RET v;  $\tau$  v
  }}}.

```

Il ne s'agit de rien d'autre que de déclarations de type (sémantique) présentées sous la forme de triplets de Hoare : étant données une valeur `t` de type `dynarray_type` τ et une valeur `v` de type τ , la fonction `dynarray_push` renvoie une valeur de type `unit_type`.

La difficulté non apparente ici est que la preuve de ces spécifications se fait non pas dans un cadre essentiellement séquentiel où l'on possède exclusivement les ressources, mais dans un cadre concurrent où il faut continuellement interagir avec un invariant IRIS. En particulier, l'implémentation de `dynarray_push` repose sur la fonction `array_blit` (notre implémentation de `Array.blit`) itérant sur une plage d'un tableau dont la spécification faible nous a conduit à généraliser sa spécifications forte de manière non triviale pour nous ramener à des accès atomiques au tableau.

Remarque 2. Il y a une différence de style entre les spécifications fortes et faibles. Une spécification forte `dynarray_model t vs` ne parle pas du type des éléments de `t` ; on pourrait le faire, mais ce n'est pas utile puisqu'on donne la valeur précise de tous les éléments, `vs`. Cette approche est plus expressive puisqu'elle permet de vérifier des programmes qui feraient des usage «mal typés» du tableau, mettant des éléments de type incompatible. (Par exemple si on voulait formaliser une implémentation utilisant `Obj.magic` pour les valeurs vides.) Au contraire, nos spécifications faibles parlent des types (sémantiques) des éléments. Ce n'est pas une préférence de notre part, c'est une nécessité pour réussir la formalisation. En effet, si on ne possède pas uniquement le tableau, on ne peut pas raisonner sur la valeur des éléments qui peut changer sous nos pieds. Il faut se mettre d'accord sur une spécification plus faible que leur valeur, que toutes les opérations préservent.

4 Discussion

4.1 Travaux futurs

Exceptions. Si le langage d’implémentation dans IRIS avait des exceptions, nous pourrions représenter plus fidèlement le comportement OCaml. L’approche la plus simple serait d’avoir des spécifications qui ne parlent pas des exceptions, donc des triplets de Hoare dont la postcondition est vraie seulement si aucune exception n’est lancée. On pourrait aussi avoir une logique qui parle des exceptions dans les spécifications, par exemple en s’inspirant de [de Vilhena and Pottier \(2021\)](#).

Itérateurs. Nous n’avons formalisé que les problématiques liées aux usages concurrents de la structure de données. Nous comptons enrichir le sous-ensemble vérifié de `Dynarray` pour y ajouter notamment des fonctions d’itération telles que `iter`, `map` et `fold_left`. Cela nous amènera à la formalisation de l’invalidation des itérateurs – avoir des exceptions serait utile pour cela.

Permissions fractionnaires. Le modèle de programmation le plus courant en OCaml 5 est que les structures de données impératives peuvent être utilisées dans un cadre séquentiel, mais aussi dans un cadre concurrent “bien synchronisé”, c’est-à-dire où les courses critiques sur la structure sont en lecture seule – les courses critiques où au moins un accès à une modification sont une erreur de programmation. Ce modèle de programmation s’applique à notre implémentation de `Dynarray`.

Par contre, les spécifications de correction fonctionnelle que nous avons formalisées demandent la possession unique du tableau dynamique, et ne permettent donc pas du tout les courses critiques, même en lecture seule. Il y a ici un écart entre les spécifications informelles et formelles. Pour capturer toute la spécification informelle, il faudrait utiliser des permissions fractionnaires dans la spécification formelle. Nous pensons que ce serait une extension facile de notre travail.

Cosmo. Une direction que nous avons déjà commencé à explorer consiste à adapter la vérification au modèle mémoire faible d’OCAML en utilisant COSMO ([Mével, Jourdan, and Pottier, 2020](#)).

Nous nous attendons à des spécifications fortes essentiellement inchangées. En revanche, nous nous attendons à une petite difficulté quant aux spécifications faibles : les tableaux non atomiques nécessitent d’introduire une notion de *vue mémoire*. Intuitivement, il s’agit d’écrire que toutes les valeurs passées des éléments du tableau support sont du bon type sémantique. Nous pensons que cette propriété peut s’exprimer assez facilement dans la logique sous-jacente BASECOSMO.

Automatiser la sûreté. La correction fonctionnelle séquentielle demande souvent des spécifications fines et des arguments de preuve parfois sophistiqués, qui doivent être apportés par l’utilisateur. Par contre, la sûreté mémoire repose sur des invariants plus simples qui devraient pouvoir être automatisés dans leur écrasante majorité.

Il serait intéressant d’explorer des formes d’automatisation de cette partie des preuves. L’utilisateur fournirait des invariants faibles pour les structures de données sous forme de types raffinés, dans notre cas $0 \leq t.length$. Cette approche pourrait tirer parti du langage de spécification [GOSPEL](#).

4.2 Travaux liés

Autres implémentations. De nombreux langages proposent la structure de tableau dynamique dans leurs bibliothèques de base – c’est d’autant plus courant que le style du

langage est impératif. C'est par exemple la structure de base en PYTHON, appelée `list`, mais PYTHON a une implémentation purement séquentielle (comme OCaml 4) où la question de la sûreté concurrente ne se pose pas. C++ propose `std::vector`, qui n'est pas synchronisé – des usages concurrents cassent la sûreté mémoire. Le système de types de RUST interdit les usages non synchronisés de `std::vec`. JAVA propose deux versions, `ArrayList` (non synchronisé) et `Vector` (synchronisé); `ArrayList` n'utilise pas d'accès non-sûrs au tableau support, pour éviter les problèmes de sûreté mémoire.

Spécifications faibles en logique de séparation. Rendre formelle la notion d'«invariant faible» utilisée pour écrire et relire le code de `Dynarray` nous a conduit naturellement à retrouver la notion de *type sémantique* du projet RUSTBELT (Jung, Jourdan, Krebbers, and Dreyer, 2018a; Dang, Jourdan, Kaiser, and Dreyer, 2020; Matsushita, Denis, Jourdan, and Dreyer, 2022). Ce n'est pas un hasard puisque la question est la même : vérifier du code `unsafe`.

Automatisation. Le projet REFINEDRUST propose de combiner vérification fonctionnelle et vérification de sûreté pour le langage RUST, avec automatisation. La [présentation](#) disponible en ligne discute justement du cas des tableaux dynamiques. Ce projet en cours aboutirait à un système proche de ce dont nous souhaiterions pour OCaml : par analogie, notre travail de vérification en IRIS correspond au projet RUSTBELT, la couche d'automatisation que nous avons suggérée correspondrait à REFINEDRUST.

Remerciements Merci à François Pottier, et aux relecteurs et/ou relectrices anonymes des JFLA'24, pour leur relecture attentive et leurs remarques.

Références

- Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. [Rustbelt meets relaxed memory](#). *Proc. ACM Program. Lang.*, 4(POPL), 2020.
- Paulo Emílio de Vilhena and François Pottier. [A separation logic for effect handlers](#). *Proc. ACM Program. Lang.*, 5(POPL), 2021.
- Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. [Bounding data races in space and time](#). In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 2018.
- Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. [Rustbelt: securing the foundations of the rust programming language](#). *Proc. ACM Program. Lang.*, 2(POPL), 2018a.
- Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. [Iris from the ground up: A modular foundation for higher-order concurrent separation logic](#). *J. Funct. Program.*, 28, 2018b.
- Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. [Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code](#). In Ranjit Jhala and Isil Dillig, editors, *PLDI '22 : 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. ACM, 2022.
- Glen Mével, Jacques-Henri Jourdan, and François Pottier. [Cosmo: a concurrent separation logic for multicore ocaml](#). *Proc. ACM Program. Lang.*, 4(ICFP), 2020.

COMPILATION

Chamelon : un minimiseur pour et en OCaml

Milla Valnet^{1,2,3}, Nathanaëlle Courant³, Guillaume Bury³, Pierre Chambart³ et Vincent Laviron³

¹École Normale Supérieure, Université PSL, Paris, 75005, France

²Sorbonne Université, CNRS, LIP6, F-75005 Paris, France

³OCamlPro, Paris, 75014, France

Lors du développement d'outils manipulant du code OCaml, il arrive que ceux-ci échouent sur un programme. Identifier et comprendre l'erreur passe généralement par réduire à la main la taille du programme en question, de sorte à obtenir un programme plus court provoquant la même erreur — tâche souvent longue, parfois complexe, rarement intéressante. Nos travaux consistent à automatiser cette tâche à l'aide d'un minimiseur, ou *delta-débugueur*. Pour ce faire, nous proposons une liste d'heuristiques unitaires, ie. appliquant au programme des réductions de faible ampleur, et un algorithme itératif pour les combiner. Ces propositions sont implémentées dans l'outil libre Chamelon. Bien que pensé pour assister le développement d'un compilateur OCaml, ce dernier s'adapte à toutes sortes de projets manipulant du code OCaml. Il permet d'analyser des projets composés de plusieurs fichiers et minimise efficacement des programmes réels/concrets, réduisant leur taille d'un à plusieurs ordres de grandeur. Il est actuellement utilisé pour assister le développement du compilateur optimisant flambda2.

1 Introduction

Les erreurs de programmes se produisent parfois sur des codes longs, très longs, de plusieurs centaines voire milliers de lignes. Identifier et isoler l'erreur est une tâche souvent longue et fastidieuse, qui consiste généralement à minimiser manuellement la taille du programme autant que possible, en supprimant des lignes, simplifiant des expressions, etc. L'objectif d'un minimiseur est alors d'automatiser ce travail.

L'idée de la minimisation n'est pas nouvelle. Parfois appelée réduction, ou encore delta-debugging en ce qu'elle travaille à la jonction, dans le « delta », entre programmes avec et sans erreur, elle est développée dès 1999 par Andreas Zeller [11] avec pour but d'isoler la cause d'une erreur de programmes en y appliquant itérativement des simplifications de parties inutiles du programme. Elle y est définie comme la méthodologie pour « réduire un problème tout en préservant une certaine propriété » — ici, l'erreur. L'outil ainsi construit ne supprime pas l'erreur, mais au contraire la pointe.

Cette méthode est déjà utilisée pour des langages comme C, avec C-reduce [1], SMT-lib, avec ddSMT [6], ou encore via de nombreuses implémentations des travaux originaux de Zeller [11]. Néanmoins, cette problématique demeure très étudiée. Zeller travaille avec Hildebrandt [10] pour identifier les entrées et interactions provoquant l'échec du programme,

avec comme cas d'étude des entrées utilisateurs sur le navigateur de Mozilla, puis démontre avec Cleve [3] que le delta-debugging fonctionne aussi bien pour identifier les erreurs dues au code lui-même qu'à ses paramètres. Dans une volonté de voir les tâches de débogage comme des cas particuliers de problèmes de minimisation, il utilise cette méthode avec Choi [2] pour les échecs d'ordonnancement de fils, ou encore avec Cleve [4] pour identifier quelles variables causent l'erreur, et à quel pas d'exécution. Enfin, Leitner et al. [7] combinent cette approche avec du découpage de programmes (*program slicing*) pour réduire la taille des cas d'échecs dans le cadre de la génération aléatoire de tests.

Pour améliorer l'état de l'art en delta-debugging, certains tentent d'utiliser de l'apprentissage machine [5], d'autres encore des algorithmes probabilistes [9], etc. En OCaml cependant, les outils existants se limitent à tenter de comprendre la cause d'une erreur de type [8].

Pendant, aucun tel outil généraliste n'existe pour le langage OCaml. L'objectif de ce projet est ainsi de proposer le premier minimiseur pour du code OCaml, Chamelon. Bien qu'initialement pensé pour assister le développement d'un compilateur OCaml, un tel outil peut se révéler utile pour le développement d'autres projets utilisant ou manipulant du code OCaml. Son fonctionnement ne repose ni sur de l'apprentissage machine, ni sur une approche probabiliste, et est exclusivement déterministe. Ce travail fournit donc les contributions suivantes :

- une liste d'heuristiques visant à minimiser le programme OCaml en entrée ;
- un algorithme simple et efficace permettant de les itérer ;
- une implémentation en OCaml d'un minimiseur appliquant ces heuristiques, supportant les projets multifichiers et les erreurs à runtime, et adaptables à moindre coût à différentes versions du compilateur.

Plan. La section 2 présente les heuristiques unitaires proposées pour minimiser le programme, tandis que la section 3 explicite la manière dont celles-ci sont combinées. La section 4 montre comment ce travail s'étend à un projet composé de plusieurs fichiers, mais aussi comment il peut être utilisé lorsque l'erreur est déclenchée non plus à la compilation, mais à l'exécution du programme. La section 5 présente les résultats expérimentaux.

2 Les heuristiques

Le concept de l'approche est de composer et combiner différentes heuristiques unitaires, en appliquant chacune d'entre elles autant que possible. Nous présentons ici les différentes heuristiques proposées pour minimiser un programme OCaml. Notons que, ciblant initialement des problèmes de compilation, notre approche vise bien plus à identifier des erreurs causées par une certaine structure de code bien plus que par une certaine sémantique ou une certaine exécution : cela guide donc nos choix d'heuristiques.

2.1 Supprimer des définitions

Supprimer les définitions en commençant par la fin

Les minimisations les moins raffinées s'intéressent à la simple suppression d'éléments du code. La première heuristique consiste ainsi à supprimer toute définition — de variables, de types, de modules, etc. — en partant de la fin. De la sorte, on cherche à supprimer le code situé après la cause de l'erreur, dont l'erreur ne dépend pas.

Remplacer les expressions par des valeurs triviales

Lorsque certaines définitions ne peuvent être simplement supprimées, on choisit de poursuivre la minimisation du code en simplifiant au maximum toutes les expressions présentes dans le code. Pour ce faire, on tente de les remplacer par des valeurs les plus

simples possibles. L'enjeu est alors de déterminer par quelle valeur triviale on souhaite remplacer notre expression tout en respectant la contrainte de type sur cette dernière.

Pour les types de base, on remplace simplement les expressions de type `int` par `0`, celles de type `float` par `0.0`, celles de type `char` par `'0'`, celles de type `string` par `""` et celles de type `unit` par `()`. Pour les autres types, nous avons opté pour une solution générale et nous basons ainsi sur les deux définitions suivantes :

```
let __dummy1__ () = assert false [@@inline never]
external __dummy2__ : unit -> 'a = "%opaque"
```

En effet, `__dummy1__ ()` et `__dummy2__ ()` sont des expressions de type `'a`, qui peuvent ainsi remplacer une expression de n'importe quelle type.

Remarquons que nous aurions pu définir `let __dummy1__ = assert false`. Cependant, cette ligne de code entraînerait un échec à son exécution : pour cette raison, un compilateur risque alors d'optimiser le code produit en propageant cet échec d'assertion, le modifiant ainsi significativement, ce qui n'est pas souhaitable en delta-debugging. En enveloppant l'assertion dans une fonction, on retarde ainsi son exécution, prévenant cette optimisation. En ajoutant `[@@inline never]`, on empêche également que cette assertion soit étendue (*inlinée*) aux sites d'appels de `__dummy1__`, où elle entraînerait une nouvelle propagation de l'échec d'assertion.¹

La définition de `__dummy2__` se repose quant à elle sur la primitive externe `opaque` : lors de la compilation, celle-ci est considérée comme une fonction renvoyant une valeur arbitraire — ici, comme une fonction de type `unit -> 'a` en raison de l'annotation. Cependant, à l'exécution, cette dernière se comporte comme la fonction identité. Pour cette raison, à l'exécution, la valeur de `__dummy2__ ()` sera `()`, causant une erreur de type.

Ainsi, dans les deux cas, l'utilisation de ces expressions *dummy*, triviales, entraîne une erreur à l'exécution du code produit. Lorsque l'objectif est d'assister la résolution d'échecs à la compilation, ce n'est pas une limitation. Cependant, à des fins de généralisation des cas d'utilisation de l'outil, cette problématique sera traitée dans la partie 4.

2.2 Simplifier des types

On cherche ensuite à simplifier les types construits définis par l'utilisateur.

Supprimer les constructeurs de types construits

Une première heuristique consiste à supprimer un constructeur `Cons` d'un type algébrique. Il s'agit alors de propager dans le code la suppression du constructeur : les expressions sous la forme `Cons(e1, ..., en)` sont remplacées par `__dummy1__ ()` ou `__dummy2__ ()`, et pour les motifs utilisant constructeur `Cons`, on supprime simplement le cas de filtrage concerné.

Supprimer les champs des types enregistrement ou des constructeurs

Lorsque supprimer le constructeur entier n'est pas possible, une heuristique peut à la place tenter d'en supprimer des champs. Ainsi, on supprime le *i*-ème champ de sa définition, et on parcourt à nouveau le code pour supprimer le *i*-ème champ dans chaque expression de la forme `Cons(e1, ..., en)`, et le *i*-ème sous-motif de chaque motif de la forme `Cons(p1, ..., pn)` — on veille alors à remplacer dans l'expression qui suit toutes les variables liées par `pi` par `__dummy1__ ()` ou `__dummy2__ ()`.

1. Notons qu'un compilateur pourrait utiliser l'information que `__dummy1__` ne renvoie jamais de valeur, ce qui influencerait sur le code produit — ce n'est cependant pas le cas du compilateur sur lequel nous travaillons.

2.3 Simplifier le code

Modifier les attributs

On cherche en premier lieu à retirer tous les attributs de fonctions, modules, etc. du programme de sorte à le rendre moins verbeux. Cependant, ajouter certains attributs peut également fournir des informations précieuses sur l'origine de l'échec. En conséquence, l'une des heuristiques tente d'ajouter `local[never|always]` et `inline[never|always]` aux fonctions du programme, de sorte à fixer les stratégies d'*inlining* du compilateur.

'Etendre les fonctions

'Etendre une fonction, ie. la remplacer par sa définition à un site d'appel, peut permettre des simplifications supplémentaires au dit site. Notons que cette transformation diffère en deux points de l'ajout de l'attribut `[@@inline always]`, qui indique au compilateur d'effectuer l'extension (ou *inlining* : tout d'abord, en effectuant la transformation « à la main », cela permet à Chamelon d'effectuer des minimisations supplémentaires à cet emplacement. Enfin, si la cause de l'échec du programme provient, par exemple, de l'utilisation d'attribut, ces deux heuristiques auront des effets différents.

Aplatir les modules

Aplatir les modules correspond à sortir les définitions de variables du bloc `module Name = struct ... end`. Ce faisant, il faut néanmoins prendre garde à éviter les conflits de noms entre des variables définies dans le module et des variables définies dans le programme. À cette fin, nous avons choisi de faire précéder le nom de la variable par le nom du module dont elle est initialement issue : il faut ensuite propager ce changement dans le programme.

2.4 Retirer les artefacts de simplification

Après avoir appliqué les heuristiques ci-dessus, des artefacts peuvent apparaître, ie. des situations qui n'apparaîtraient pas ou peu dans un code utilisateur.

Supprimer le code mort

Pour chaque variable, module, type, on parcourt le code de sorte à déterminer s'ils sont utilisés dans le programme. S'ils ne le sont pas, on supprime simplement leur définition.

Simplifier les filtrages par motif

Lorsque le filtrage ne contient qu'un seul motif constitué d'une variable, on remplace `match e1 with x -> e2` par `e2` dans lequel `x` a été substituée textuellement par `e1`.

Simplifier les séquences

On remplace les expressions sous la forme `() ; e` par `e`.

Séquentialiser les appels de fonction

À la suite de simplifications, on peut obtenir une application de fonction sous la forme `(__dummy__ ()) e1 ... en`. On séquentialise alors en évaluant séparément chaque argument, de sorte à obtenir des expressions non imbriquées. On utilise la primitive suivante² :

```
external __ignore__ : 'a -> unit = "%ignore"
```

2. Cette primitive existe déjà avec ce type dans la bibliothèque standard OCaml. La redéfinir permet cependant de ne pas en dépendre.

On transforme alors `(__dummy__ ()) e1 ... en en :`
`__ignore__ e1 ; ... ; __ignore__ en ; __dummy__ ()`

Simplifier les `rec` et les arguments inutilisés

À la suite du remplacement d'expressions et de définitions par les valeurs `dummy` détaillées plus haut, il arrive que des arguments d'une fonction ne soient plus utilisés dans son corps. Dans ce cas, on cherche à supprimer ces derniers.

Il faut alors prendre garde à propager leur suppression à tous les sites d'appel de la fonction en question. Lorsque le i -ème argument de la fonction `f` est supprimé, on remplace alors toutes les occurrences `f` par `(fun x1 ... xn -> f x1 ... xi-1 xi+1 ... xn)`.

De même, lorsque la fonction n'est plus récursive, on retire le mot clé `rec`.

3 L'itération

Une fois muni de ces heuristiques, il s'agit de les itérer astucieusement de sorte à minimiser efficacement un programme.

3.1 Interface avec les heuristiques

Une heuristique unitaire peut généralement s'appliquer en différents points d'un programme. Elle est implémentée de sorte que, étant donné un indice i , elle tente de minimiser le i -ème point qu'elle peut simplifier. Trois cas se présentent alors :

- Cette simplification ne supprime pas l'erreur : on a minimisé le programme !
- Cette simplification supprime l'erreur : on ne souhaite pas l'appliquer.
- L'indice est plus grand que celui du dernier point qu'on peut modifier.

On définit un type algébrique correspondant pour le type de retour d'un minimiseur :

```
type 'a minimized_step_result =
  | New_state of 'a
    (* New (smaller) states that produces an error *)
  | Change_removes_err
    (* This change removes the error, but other might be possible *)
  | No_more_changes
    (* The last possible position for changes has been reached *)
```

3.2 Itération linéaire

On dispose d'un état `state` qui représente notre programme et une heuristique `f`. Il s'agit désormais d'itérer l'application de cette heuristique sur le programme :

```
let minimize_basic (state : 'a)
  (f : 'a -> pos:int -> 'a minimized_step_result) : 'a * bool =
  let rec aux (state : 'a) (pos : int) (ever_changed : bool) =
    match f state ~pos with
    | New_state nstate -> aux nstate pos true
    | Change_removes_error -> aux state (pos + 1) ever_changed
    | No_more_changes -> (state, ever_changed)
  in
  aux state 0 false
```

Ici, la fonction `aux` prend en paramètre le programme, une position à laquelle tenter d'effectuer les changements, et un booléen symbolisant le fait que de tels changements aient eu lieu précédemment. Elle renvoie l'état du programme après application itérée de l'heuristique et ce booléen.

Le fonctionnement est le suivant : on tente de minimiser le programme au point donné. Si la minimisation est possible, on itère sur le nouvel état, en signifiant que des changements ont été effectués ; il n'y a nul besoin d'incrémenter la position, puisqu'après avoir simplifié le n -ième point, le point suivant est devenu n -ième sur le nouveau programme. Si la minimisation n'est pas possible, le programme et le booléen ne changent pas et on examine la position suivante. Enfin, si la position est trop grande, cela signifie que le programme a été parcouru, et on s'arrête donc en renvoyant le programme et le booléen courant.

On peut se représenter le fonctionnement de cet algorithme grâce à ce schéma :



Ainsi, l'idée est la suivante : on considère un programme dont les cases noires sont des points du programme déjà minimisés et les cases blanches sont des points non minimisés. On tente de minimiser la case 2, en gris, ie. le troisième point où peut s'effectuer une minimisation. Deux cas se présentent alors :

- La minimisation ne retire pas l'erreur (à gauche) : on l'applique, la case devient noire, et dans ce cas, on considère à nouveau le troisième point du programme où peut s'effectuer une minimisation.
- La minimisation retire l'erreur (à droite) : on ne souhaite pas l'appliquer, et dans ce cas, on considère la case 3, ie. le quatrième point du programme où peut s'effectuer une minimisation.

On remarque qu'un tel itérateur est modulaire en l'heuristique `f` passée en paramètre : il est donc aisé d'ajouter de nouvelles heuristiques.

3.3 Optimisation dichotomique

Une telle boucle s'optimise aisément par dichotomie, en ne tentant plus de minimiser des emplacements un par un, mais plutôt un ensemble d'emplacements d'une longueur 2^n . Cette méthode permet de gagner un facteur 10 d'efficacité sur des programmes réels de quelques milliers de lignes.

3.4 Ordre des heuristiques

L'ordre dans lequel combiner les différentes heuristiques pour optimiser le temps de minimisation a été déterminé expérimentalement, sur un petit échantillon de tests, en se contentant principalement de terminer par les heuristiques retirant les artefacts de simplification. Pour un ordonnancement plus robuste et efficace, une recherche plus poussée et des tests supplémentaires pourraient se révéler utiles.

4 Extensions

Jusqu'alors, nous nous sommes concentrés sur la minimisation d'un programme unique dont la compilation échoue. Cependant, des extensions ont été réalisées de sorte à en élargir le cadre d'utilisation.

4.1 Multifichier

Dans des cas d'utilisation réels, le travail s'effectue sur un projet composé d'un ensemble de fichiers, dépendant les uns des autres. Ainsi, nous avons cherché à adapter Chamelon de sorte à le rendre capable d'opérer sur des projets composés de plusieurs fichiers. On ajoute ainsi quelques minimisations spécifiques au cas multifichier :

- Tout d'abord, on cherche à supprimer le maximum de ces fichiers, dans l'ordre des dépendances ;
- ensuite, on tente de fusionner des fichiers pour aboutir au nombre minimal de fichiers à minimiser ;
- une fois ces deux minimisations effectuées, on minimise indépendamment chaque fichier restant avec les méthodes montrées précédemment.

Lors de ce dernier point, une précaution toute particulière doit être prise : en effet, lorsqu'un objet est modifié par une heuristique, il faut prendre bien garde à en propager les modifications à toutes ses dépendances : par exemple, si un argument de la fonction `f` est supprimé, il faut supprimer cet argument à tous les points d'appels de `f`, dans chacune des dépendances du programme.

Pour utiliser Chamelon en multifichier, il faut lui fournir la liste des fichiers sur lesquels effectuer la minimisation, dans l'ordre des dépendances — qui peut être obtenu à l'aide de l'outil `ocamldep`.

4.2 Runtime

Les travaux présentés jusqu'à présent se concentrent sur des erreurs à la compilation. Cependant, il arrive que l'erreur soit provoquée non pas à la compilation, mais à l'exécution du programme, via notamment une erreur de segmentation. En conséquence, on adapte les heuristiques proposées.

En effet, l'utilisation des variables `__dummy1__` et `__dummy2__` provoque des erreurs à l'exécution, rendant le minimiseur inutilisable lorsque le programme doit être exécuté.

L'enjeu est donc d'écrire un algorithme qui, étant donné un type d'entrée, génère une expression de même type, la plus simple possible. Il se construit de la manière suivante :

```
let rec generate_dummy_expr env typ =
  match typ with
  | Tint -> Texp_constant (Const_int 0)
  | Tvar vtyp -> (* 'a *)
    let id = find_value_of_type vtyp env in Texp_ident id
  | Tarrow (arg_label, t1, t2) -> (* t1 -> t2 *)
    let param = Ident.create "x" in
    let nenv = add_value param t1 env in
    Texp_function({arg = param ; body = generate_dummy_expr env t2})
  | Tconstr (tname, t_list, _) -> (* t1 * ... * tn tname *) ->
    let type_descr = find_type tname env in
    let non_rec_cons = List.filter non_recursive type_descr.constrs in
    let cons = arity_min non_rec_cons in
    let arg_types = instantiate cons.args t_list in
    Texp_construct(cons, List.map (generate_dummy_expr env) arg_types)
```

Notons que nous présentons ici une version simplifiée de l'AST à des fins pédagogiques. Son fonctionnement est le suivant : lorsque le type est `int`, `char`, `string` ou `unit`, on génère une expression simple du type souhaité, ie. `0`, `'0'`, `""` ou `()`. Lorsque le type est une variable de type (par exemple, à l'intérieur d'une fonction polymorphe), on cherche

dans l'environnement une variable de ce type et on la renvoie. Lorsque ce type est un type flèche $t_1 \rightarrow t_2$, on ajoute dans l'environnement une variable de type t_1 , on construit une expression e de type t_2 dans cet environnement, et on renvoie `fun x -> e`.

Enfin, lorsque ce type est un type construit (t_1, \dots, t_n) $tname$, on cherche la définition de ce type dans l'environnement. On construit ensuite une expression avec le constructeur `Cons` d'arité minimale du type (t_1, \dots, t_n) $tname$ ne contenant pas d'appel récursif au type $tname$. Pour ce faire, on considère le type des arguments de ce constructeur et on prend garde d'en instancier les variables de type a_1, \dots, a_n par t_1, \dots, t_n . Enfin, on génère une expression e_i du type de chaque argument et on renvoie `Cons(e1, ..., en)`.

Remarquons qu'un tel algorithme peut échouer (s'il n'existe pas de constructeur non récursif ou une variable d'un certain type) : dans ce cas, l'expression n'est pas simplifiée. Dans le cas contraire, on simplifie sans recours à `__dummy1__` ou `__dummy2__` : ainsi, minimiser un programme n'introduit pas d'erreur supplémentaire à l'exécution. Selon le contexte, l'utilisateur peut ainsi choisir entre les heuristiques initiales, plus efficaces mais introduisant de telles erreurs, et cette heuristique.

4.3 Bibliothèque de compatibilité

L'implémentation se base sur l'utilisation des compiler-libs d'OCaml, afin de lire le `.cmt` produit à la compilation d'un programme, et de manipuler l'arbre de syntaxe abstrait typé qu'elle en obtient (via `Tast_mapper`). Cependant, cette bibliothèque est différente selon le compilateur ou sa version.

En conséquence, une bibliothèque de compatibilité a été implémentée : ainsi, changer de version de compilateur s'effectue à moindre coût, requérant simplement quelques informations sur la nouvelle forme de l'AST.

5 Implémentation et évaluation expérimentale

Les différentes méthodes et heuristiques explicitées plus haut ont toutes été implémentées dans l'outil Chamelon, disponible en source libre sur github³.

5.1 Contexte de développement

À l'origine, Chamelon avait pour objectif d'assister le développement du compilateur optimisant `flambda2`⁴. En effet, lorsque celui-ci échouait sur un programme ou ensemble de programmes pourtant corrects au regard du compilateur standard, identifier la cause de l'erreur dans `flambda2` n'était pas toujours aisé.

Cependant, Chamelon est construit de manière modulaire. S'il applique les minimiseurs décrits plus hauts en s'assurant qu'une certaine condition demeure valide, il est parfaitement modulaire en la dite condition. Il suffit alors de fournir à l'outil la commande que l'on souhaite exécuter sur le programme, et la condition/l'erreur que l'on souhaite maintenir en résultat. Il faut néanmoins noter que les transformations effectuées par Chamelon s'intéressent plus à la structure du programme et ne maintiennent pas sa sémantique. En conséquence, l'utiliser sur des outils dépendant fortement de données sémantiques comme la valeur des variables produirait vraisemblablement des résultats limités.

5.2 Résultats expérimentaux

L'outil ainsi développé est actuellement utilisé par l'équipe `flambda` à OCamlPro pour assister la production de `flambda2`. Il y a obtenu des résultats probants sur des cas réels d'échec de celui-ci, diminuant significativement la taille du problème en sortie.

3. disponible au lien <https://github.com/ocaml-flambda/flambda-backend/tree/main/chamelon>

4. disponible au lien <https://github.com/ocaml-flambda/flambda-backend>

Parmi les résultats expérimentaux, le minimiseur a été en mesure de minimiser un programme de 650 lignes dont la compilation échouait en un programme minimisé de seulement 6 lignes provoquant la même erreur, permettant d'identifier un problème lié à l'optimisation des filtrages par motifs :

```

1 let offset ~byte_order byte_nr =
2   match byte_order with | `Little_endian -> 0 | `Big_endian -> byte_nr
3 let pack_unsigned_16 ~byte_order =
4   __ignore__ ((offset ) ~byte_order 0);
5   __ignore__ ((__dummy2__ ()) ((offset ) ~byte_order 1));
6   __dummy2__ ()

```

Nous avons également testé le minimiseur sur des programmes de taille plus conséquente. Par exemple, étant donné un des fichiers du typeur Caml de 3842 lignes, sur lequel le compilateur échouait, le minimiseur a été en mesure de le réduire à un programme de 22 lignes seulement, le tout en une trentaine de minutes sur un ordinateur portable moyen :

```

1 open Types
2 module String = Misc.Stdlib.String
3 type module_entry = | Mod_persistent
4 and module_data = { mda_declaration: Subst.Lazy.module_declaration }
5 and t = { local_constraints: type_declaration Path.Map.t }
6 let persistent_env : module_data Persistent_env.t ref = __dummy2__ ()
7   [@@local never][@@inline never]
8 let find_pers_mod name =
9   Persistent_env.find (!persistent_env) (__dummy2__ ()) (__dummy2__ ())
10  [@@local never][@@inline never]
11 type _ load =
12   | Don't_load: unit load
13   | Load: module_data load
14 let lookup_ident_module (type a) (load : a load) =
15   match __dummy2__ () with
16   | Mod_persistent ->
17     (match load with
18     | Don't_load -> (__dummy2__ (), __dummy2__ ())
19     | Load ->
20       (match find_pers_mod (__dummy2__ ()) with
21       | mda -> ((__dummy2__ ()), (mda : a))
22       | exception Not_found -> __dummy2__ ())[@@local never][@@inline never]

```

Remarquons que la réduction de la taille du programme n'est pas la seule action intéressante du minimiseur. En effet, sur cet exemple, on peut noter qu'à la ligne 18, `(__dummy2__(), __dummy2__())` n'a pas été simplifié en `__dummy2__()` : cela signifie que cette modification retirait l'erreur, information qui peut donc être exploitée pour la comprendre. Il s'agit de ne pas seulement tirer avantage de la minimisation, mais aussi de la *minimalité* du programme au regard des heuristiques appliquées.

Souvent, la sortie peut encore être minimisée à la main. Cependant, le minimiseur réalise une partie conséquente du travail de manière automatique. Enfin, dans le cas d'erreurs dans un cadre multifichier, le minimiseur est bien en mesure de fusionner ou supprimer des fichiers, de sorte à aboutir à une copie minimisée du projet qui déclençait l'erreur.

Conclusion

À l'avenir, une extension intéressante serait de rendre le minimiseur Chamelon compatible avec un projet *dune*, et non plus seulement avec une liste de fichiers. Enfin, notamment à travers son utilisation dans des exemples réels, il serait intéressant d'améliorer les heuristiques existantes ou d'en trouver de nouvelles, de sorte à rendre le minimiseur plus robuste, plus efficace et plus rapide.

Néanmoins, ce travail propose le premier minimiseur, ou delta-débugueur pour et en OCaml, au service de sa communauté !

Références

- [1] *C-reduce project*. URL : <https://github.com/csmith-project/creduce>.
- [2] Jong-Deok CHOI et Andreas ZELLER. “Isolating failure-inducing thread schedules”. In : *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*. 2002, p. 210-220.
- [3] Holger CLEVE et Andreas ZELLER. “Finding Failure Causes through Automated Testing”. In : *Proceedings of the Fourth International Workshop on Automated Debugging, AADEBUG 2000, Munich, Germany, August 28-30th, 2000*. Sous la dir. de Mireille DUCASSÉ. 2000. URL : <https://arxiv.org/abs/cs/0012009>.
- [4] Holger CLEVE et Andreas ZELLER. “Locating causes of program failures”. In : *Proceedings of the 27th international conference on Software engineering*. 2005, p. 342-351.
- [5] Kihong HEO et al. “Effective Program Debloating via Reinforcement Learning”. In : *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada : Association for Computing Machinery, 2018, p. 380-394. ISBN : 9781450356930. DOI : [10.1145/3243734.3243838](https://doi.org/10.1145/3243734.3243838). URL : <https://doi.org/10.1145/3243734.3243838>.
- [6] Gereon KREMER, Aina NIEMETZ et Mathias PREINER. “ddSMT 2.0 : Better Delta Debugging for the SMT-LIBv2 Language and Friends”. In : *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part II*. Sous la dir. d’Alexandra SILVA et K. Rustan M. LEINO. T. 12760. Lecture Notes in Computer Science. Springer, 2021, p. 231-242. DOI : [10.1007/978-3-030-81688-9_11](https://doi.org/10.1007/978-3-030-81688-9_11). URL : https://doi.org/10.1007/978-3-030-81688-9_11.
- [7] Andreas LEITNER et al. “Efficient Unit Test Case Minimization”. In : *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*. ASE ’07. Atlanta, Georgia, USA : Association for Computing Machinery, 2007, p. 417-420. ISBN : 9781595938824. DOI : [10.1145/1321631.1321698](https://doi.org/10.1145/1321631.1321698). URL : <https://doi.org/10.1145/1321631.1321698>.
- [8] Joanna SHARRAD et Olaf CHITIL. “Refining the Delta Debugging of Type Errors”. In : *Proceedings of the 33rd Symposium on Implementation and Application of Functional Languages*. IFL ’21. Nijmegen, Netherlands : Association for Computing Machinery, 2022, p. 10-19. ISBN : 9781450386449. DOI : [10.1145/3544885.3544888](https://doi.org/10.1145/3544885.3544888). URL : <https://doi.org/10.1145/3544885.3544888>.
- [9] Guancheng WANG et al. “Probabilistic Delta Debugging”. In : *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece : Association for Computing Machinery, 2021, p. 881-892. ISBN : 9781450385626. DOI : [10.1145/3468264.3468625](https://doi.org/10.1145/3468264.3468625). URL : <https://doi.org/10.1145/3468264.3468625>.
- [10] A. ZELLER et R. HILDEBRANDT. “Simplifying and isolating failure-inducing input”. In : *IEEE Transactions on Software Engineering* 28.2 (2002), p. 183-200. DOI : [10.1109/32.988498](https://doi.org/10.1109/32.988498).
- [11] Andreas ZELLER. “Yesterday, My Program Worked. Today, It Does Not. Why ?” In : *SIGSOFT Softw. Eng. Notes* 24.6 (oct. 1999), p. 253-267. ISSN : 0163-5948. DOI : [10.1145/318774.318946](https://doi.org/10.1145/318774.318946). URL : <https://doi.org/10.1145/318774.318946>.

Correct, Fast LR(1) Unparsing

François Pottier

Inria Paris

We describe an extension of the LR(1) parser generator Menhir with new features that aim to facilitate *unparsing*, that is, transforming abstract syntax trees back into text. Our method supports non-LR(1) grammars decorated with precedence declarations and guarantees correct unparsing, by which we mean that parentheses or other disambiguation symbols are inserted where necessary. Furthermore, it allows the user to control other aspects of the unparsing process, such as layout. Our contributions include a novel view of unparsing as a composition of several successive transformations; the novel concept of *disjunctive concrete syntax trees* (DCSTs); a fast algorithm that converts DCSTs to ordinary concrete syntax trees (CSTs), thereby deciding where disambiguation symbols must be inserted; and the automated generation of safe APIs for the construction of DCSTs and deconstruction of CSTs.

In the beginning were the words, and the words were trees,
and the trees were words.

Kats, Visser and Wachsmuth [KVVW10]

Broadly speaking, *parsing* transforms text into a tree-structured representation; *unparsing* is the opposite transformation. Whereas parsing is a challenging problem, which has been heavily studied in the last sixty years [GJ08], unparsing has received little attention, if any at all. At first, this may seem natural. Whereas parsing aims to *discover* tree structure, unparsing *destroys* this structure—what could possibly be easier?

One must recall, however, that parsing usually includes a transformation of parse trees into abstract syntax trees (ASTs). This transformation typically *destroys* information, such as the placement of parentheses or other syntactic delimiters. Unparsing must *reconstruct* this information. Thus, unparsing is not so easy after all, and *it is possible to get it wrong*. An incorrect unparsers might insert too few parentheses and emit a sequence of tokens that does not have the intended meaning.

Parsing and unparsing in multiple steps Parsing is traditionally decomposed into multiple steps, recalled in in Figure 1:

1. A stream of characters is *lexed*, i.e., turned into a stream of tokens.
2. The stream of tokens is *grown*, i.e., turned into a concrete syntax tree (CST).
3. The concrete syntax tree is *interpreted*, i.e., turned into an abstract syntax tree (AST).

Step 2 alone, or the combination of steps 2 and 3, are often referred to as “parsing”. We reserve “parsing” to refer to the composition of steps 1, 2, and 3. Concrete syntax trees are also known as “parse trees”.

In the OCaml world, step 1 is usually performed with the help of a lexer generator, such as `ocamllex`. Steps 2 and 3 are performed with the help of a parser generator, such as

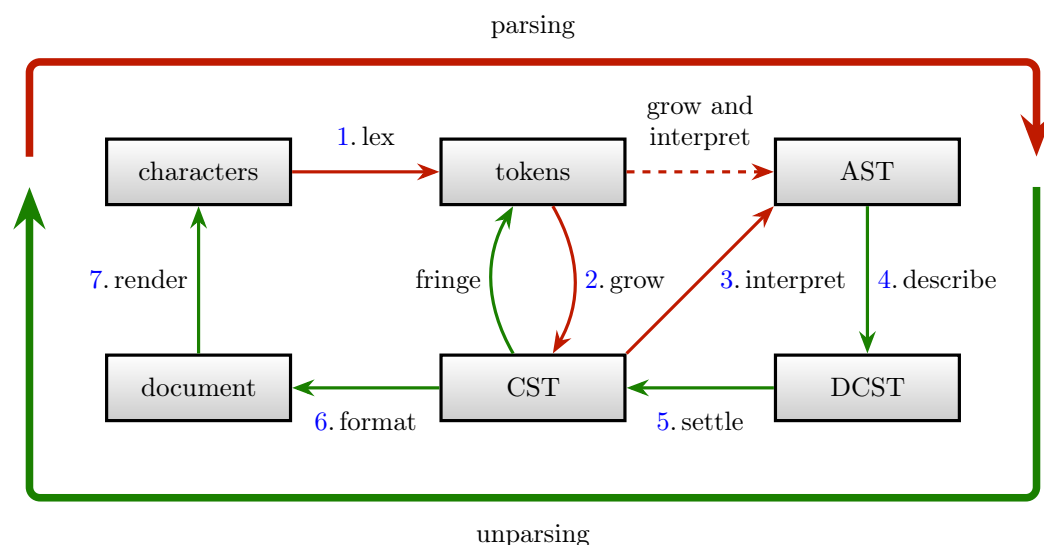


Figure 1. From text to abstract syntax trees and back

Menhir [PRG23]. The latter two steps are usually combined: the grammar is decorated with *semantic actions* that indicate how to transform CSTs into ASTs, so the generated parser transforms a stream of tokens directly into an AST, without building a CST.

We propose to decompose unparsing into several steps, also shown in Figure 1:

4. An AST is first *described*, i.e., turned into a *disjunctive concrete syntax tree* (DCST). This step must be programmed by the user (§1, §2). A DCST may contain *disjunction* nodes. They allow the user to indicate that a subtree admits multiple descriptions, e.g., without or with surrounding parentheses. In the next step, a choice between these descriptions must be made.
5. This DCST is *settled*, i.e., turned into a CST c . This step is performed by an algorithm that we provide (§3). This algorithm settles every choice: at every disjunction node, it decides which subtree must be used. This algorithm provides a correctness guarantee: it chooses a CST c such that the round-trip property $grow(fringe(c)) = c$ holds. It fails if it is unable to satisfy this property.
6. The CST is *formatted*, i.e., turned into a data structure that may include layout information. The *documents* offered by the PPrint library [Pot23] are an example of such a data structure. We provide a default implementation of this step in the form of a visitor class for CSTs (§4). By overriding selected methods, the user can modify the default behavior and express a custom layout policy.
7. The document is *rendered*, i.e., turned into a string. This step is performed by the PPrint library or by whichever layout library the user chooses to rely upon.

It is up to the user to ensure the correctness of the steps that she implements. The correctness of *describe* is expressed by the equation $interpret \circ settle \circ describe(t) = t$, where t is an AST. The correctness of *format* and *render* is expressed by $grow \circ lex \circ render \circ format(c) = c$, where c is a CST. Ensuring that this property holds requires some care; for instance, it may require sometimes inserting whitespace between two consecutive tokens.⁵

Contribution We describe an extension of Menhir with new facilities for unparsing:

- a DCST construction API (a set of constructor functions), generated, for use in step 4;
- the settlement algorithm, written once and for all, which performs step 5;
- a CST deconstruction API (a visitor class), generated, for use in step 6.

By decomposing unparsing into several steps, we give the user the freedom to control or customize certain steps. The tasks that we leave to the user include programming step 4 and customizing step 6 insofar as desired. We do not attempt to automate these tasks. Automating step 4 would amount to automatically reversing the semantic actions. Removing the need or the ability to customize step 6 would require giving the user some other way of expressing a layout policy. We have not explored these avenues, but others have (§5).

A correctness guarantee Because we let the user implement or customize steps 4 and 6, we cannot guarantee that these steps are correct. Thus, we cannot guarantee an end-to-end round-trip property of the form $parse(unparse(t)) = t$, where t is an AST. Still, we *can* and do guarantee a more restricted property, namely the correctness of step 5, which converts a DCST to a CST. We guarantee that this step always produces a viable CST, where a CST c is *viable* if the round-trip property $grow(fringe(c)) = c$ holds.

The function *fringe* maps a CST to its fringe, a sequence of tokens. The partial function *grow*, which models a deterministic parser, transforms a sequence of tokens into a CST. Thus, the equation $grow(fringe(c)) = c$ means that it is safe to display the CST c in the form of the sequence of tokens $fringe(c)$: there is no danger that this sequence might be rejected or misread as a different CST.

The reverse round-trip property $fringe(grow(s)) = s$ states that if the parser accepts the sentence s and constructs a CST c then the fringe of this CST is s . This means that *grow* is a correct parser. We expect this property to hold, but it is not the focus of this paper.

The reader may wonder: are there non-viable CSTs? and could one fuse steps 4 and 5 and let the user implement a direct transformation of ASTs to CSTs? Our answer is two-fold:

- If the grammar is in the class LR(1) [Knu65, GJ08] then every concrete syntax tree is viable. In this special case, we *could* trust the user to implement a transformation of ASTs into CSTs, as there is no danger that the user might build a non-viable CST.
- If the grammar is not in this class then the deterministic parser that is constructed by Menhir, after resolving LR(1) conflicts, is incomplete: there exist well-formed CSTs that the parser will never produce. In other words, the function *grow* is not surjective. As a result, there exist non-viable CSTs: if a CST c lies outside the range of *grow*, then $grow(fringe(c)) = c$ cannot hold. Furthermore, it is difficult to tell which CSTs are viable and which are not viable: this requires detailed knowledge of the LR(1) automaton and of the manner in which conflicts have been resolved, perhaps under the influence of user-provided *precedence declarations*. Therefore, we *do not* trust the user to build CSTs: she might, by mistake, construct non-viable CSTs.

Our approach, then, is to ask the user to build DCSTs. Thanks to disjunction nodes, a single DCST represents a family of many possible CSTs. Each member of this family may or may not be viable. It is then up to our settlement algorithm to choose a viable CST, if there exists one, among this family. The algorithm has access to a description of the LR(1) automaton and can therefore determine which CSTs are viable.

In other words, one could say that we let the user indicate where parentheses *may* be inserted, and we decide where to *actually* insert parentheses so as to ensure viability.

We do not guarantee optimality, that is, minimal use of parentheses. Instead, we offer a biased disjunction: our settlement algorithm favors the left branch. In most real-world scenarios, this lets the user achieve optimality simply by building disjunction nodes whose meaning is: “first, try displaying the subtree d without parentheses; if this is not permitted, display d surrounded with parentheses”. In such a disjunction node, the subtree d is *shared* between the two disjuncts. Thus, although “T” stands for “tree”, DCSTs are often DAGs.

If the grammar is in the class LR(1), then the user typically constructs DCSTs that do not contain any disjunction nodes (§1). These DCSTs are also CSTs, and, because in this case every CST is viable, they are viable CSTs. Then, the settlement algorithm behaves as the identity function.

Settlement algorithms We discuss two settlement algorithms, both of which we have implemented. One algorithm is complete: if the DCST d , viewed as a family of CSTs, contains at least one viable member, then $settle(d)$ must succeed. This algorithm, which exploits memoization, has linear time and space complexity in the size of the DAG d , where the constant factor depends on the sizes of the alphabet and automaton. Unfortunately, in practice, it is slow. The other algorithm is simpler, much faster in practice, and has linear time complexity in the tree size of d , which we define below. However, it is incomplete: under certain conditions, it can fail even though the family d contains a viable CST.

Definitions We fix a context-free grammar [GJ08]. We write a, b for terminal symbols and A for a nonterminal symbol. We write s for a sentence (a string of tokens) and α for a sentential form (a string of terminal or nonterminal symbols). We write κ for a *token*. We assume that there is a mapping sym of tokens to terminal symbols.

A *concrete syntax tree* (CST) is a tree whose nodes are *terminal nodes*, labeled with a token κ , or *nonterminal nodes*, labeled with a production $A \rightarrow \alpha$. A terminal node has no children. A nonterminal node has children whose number and head symbols must match α . Thus, $c ::= T \kappa \mid N (A \rightarrow \alpha, \vec{c})$. The *head symbol* of a CST c , written $head(c)$, is defined by the equations $head(T \kappa) = sym(\kappa)$ and $head(N (A \rightarrow \alpha, \vec{c})) = A$.

A *disjunctive concrete syntax tree* (DCST) may have binary disjunction nodes: thus, $d ::= T \kappa \mid N (A \rightarrow \alpha, \vec{d}) \mid d \vee d$. The two children of a disjunction node $d_1 \vee d_2$ must have the same head symbol, which is considered the head symbol of the disjunction node: $head(d_1 \vee d_2) = head(d_1) = head(d_2)$. A DCST d represents a family of CSTs; we write $c \in d$ when the CST c is a member of this family.

A DCST can be represented as a DAG in memory. We write $\#d$ for the *DAG size* of d , that is, for the number of vertices involved in its representation as a DAG. We write $|d|$ for the *tree size* of d . This function is defined by $|T \kappa| = 1$, $|N (A \rightarrow \alpha, \vec{d})| = 1 + \Sigma|\vec{d}|$, and $|d_1 \vee d_2| = \max(|d_1|, |d_2|)$. Thus, $|d|$ is the maximum size of a CST c such that $c \in d$.

1 Working with a Conflict-Free Grammar

A simple grammar, expressed in Menhir’s syntax, appears in Figure 3. This grammar describes arithmetic expressions that include integer literals, additions, multiplications, and parenthesized subexpressions. This grammar is stratified: three syntactic categories, also known as nonterminal symbols, are distinguished. A *factor* is either an integer constant or a parenthesized expression; a *term* is either a factor or the multiplication of a term and a factor; an *expression* is either a term or the addition of an expression and a term. Menhir accepts this grammar without reporting any conflict: this guarantees that this grammar is in the class LR(1), therefore is unambiguous.

To see intuitively why this grammar is unambiguous, we suggest this exercise: construct a CST whose fringe forms the token sequence INT(1); ADD; INT(2); MUL; INT(3), which corresponds to the characters 1+2*3. The reader will find that there exists only one such tree, namely the one shown in Figure 4a.

In the grammar of Figure 3, the productions carry semantic actions that construct ASTs. ASTs form an algebraic data type whose definition appears in Figure 2. ASTs do not keep track of parentheses: the semantic action for the production `factor: LPAR; expr; RPAR` (line 12) does not construct an AST node.

An unusual feature of the grammar in Figure 3 is that each production is assigned a unique name via a `@name` attribute. These names are used in the construction of data constructor names and method names in the generated APIs for DCSTs and CSTs. The reader may wish to peek ahead at Figures 5 and 13.

The code in Figures 2 and 3 is written by the programmer. Throughout the paper, hand-written code is shown on a green background, whereas code generated by Menhir appears on a yellow background.

```

1 type binop = BAdd | BMul      (* Binary operators *)
2 type expr =                   (* Expressions *)
3   | EConst of int
4   | EBinOp of expr * binop * expr
5 type main = expr

```

Figure 2. Arithmetic expressions: the module AST

```

1 %token<int> INT              (* Tokens *)
2 %token      ADD  "+"
3 %token      MUL  "*"
4 %token      LPAR "("
5 %token      RPAR ")"
6 %token      EOL
7 %start<AST.main> main      (* Start symbol *)
8 %{ open AST %}
9 %%                          (* Productions and semantic actions *)
10 factor:
11 | i = INT                  { EConst i }           [@name const]
12 | LPAR; e = expr; RPAR    { e }                 [@name paren]
13
14 term:
15 | f = factor               { f }                 [@name factor]
16 | t = term; MUL; f = factor { EBinOp (t, BMul, f) } [@name mul]
17
18 expr:
19 | t = term                 { t }                 [@name term]
20 | e = expr; ADD; t = term  { EBinOp (e, BAdd, t) } [@name add]
21
22 main:
23 | e = expr; EOL           { e }                 [@name eol]

```

Figure 3. Stratified grammar: the parser

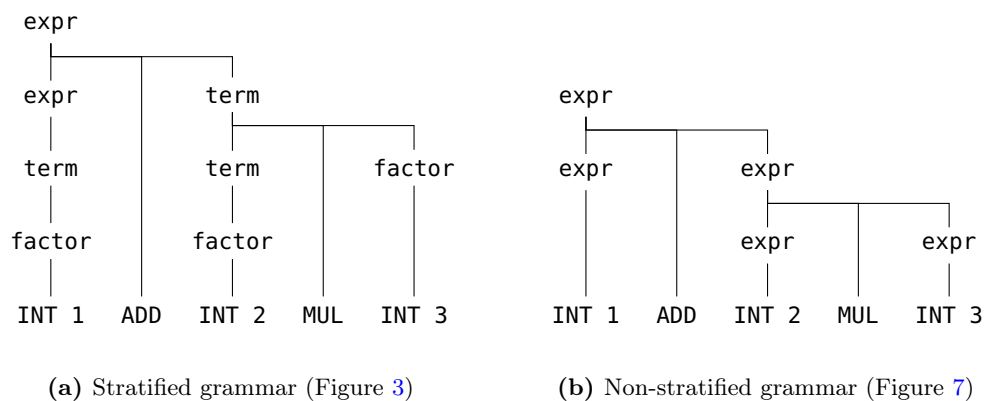


Figure 4. Concrete syntax trees for 1+2*3

```

0 module DCST : sig
1   type factor
2   type term
3   type expr
4   type main
5   val factor_choice: factor -> factor -> factor (* Constructors for [factor] *)
6   val const: int -> factor
7   val paren: expr -> factor
8   val term_choice: term -> term -> term          (* Constructors for [term] *)
9   val factor: factor -> term
10  val mul: term -> factor -> term
11  val expr_choice: expr -> expr -> expr          (* Constructors for [expr] *)
12  val term: term -> expr
13  val add: expr -> term -> expr
14  val main_choice: main -> main -> main         (* Constructors for [main] *)
15  val eol: expr -> main
16 end

```

Figure 5. Stratified grammar: signature of the module DCST

```

1 let rec factor : AST.expr -> DCST.factor = function
2   | EConst i          -> DCST.const i
3   | e                 -> DCST.paren (expr e)
4
5 and term           : AST.expr -> DCST.term = function
6   | EBinOp (e1, BMul, e2) -> DCST.mul (term e1) (factor e2)
7   | e                   -> DCST.factor (factor e)
8
9 and expr          : AST.expr -> DCST.expr = function
10  | EBinOp (e1, BAdd, e2) -> DCST.add (expr e1) (term e2)
11  | e                     -> DCST.term (term e)
12
13 and main         : AST.main -> DCST.main = function
14  | e                 -> DCST.eol (expr e)

```

Figure 6. Stratified grammar: translating ASTs to DCSTs

Transforming ASTs to CSTs As explained earlier, we propose to break down “unparsing” into several steps. The first two steps, namely steps 4 and 5, aim to transform an AST t into a CST c . Given an AST t , the problem is to construct a CST c that corresponds to t . Writing $interpret_A$ for the transformation of CSTs to semantic values that is defined by the semantic actions, the problem can be stated as follows:

Given a semantic value t for a nonterminal symbol A ,
produce a CST c whose head symbol is A , such that $interpret_A(c) = t$.

When the grammar is in the class LR(1), there is no danger of picking a non-viable CST c , that is, a CST whose fringe is rejected by the parser or understood by the parser as a different CST c' .¹

¹Suppose the grammar is in the class LR(1). Then, there exists a correct and complete parser, which we model as a partial function $grow$ of sequences of tokens to CSTs. Let us prove that every CST c satisfies $grow(fringe(c)) = c$. Let c be a CST, and let s stand for its fringe. Then, s must be a grammatically valid sentence. Because the parser is complete, it must accept this sentence: $grow(s)$ must be defined. Let us refer to this CST as c' . Because the parser is correct, we must have $fringe(c') = s$, that is, $fringe(c') = fringe(c)$. Because the grammar is unambiguous, two distinct CSTs cannot have the same fringe: so, $c' = c$ must hold. In other words, $grow(fringe(c)) = c$ holds.

A simple and safe way of implementing the transformation of ASTs to CSTs is as a family of mutually recursive functions whose structure mirrors the structure of the grammar. For each nonterminal symbol A , one writes a function, also named A , whose specification is the one given above: this function transforms an AST t into a CST c whose head symbol is A while respecting the constraint $\text{interpret}_A(c) = t$.

By following this pattern, for the stratified grammar of Figure 3, it is straightforward to implement a correct and optimal transformation of ASTs to CSTs. The code is shown in Figure 6. Technically, this code transforms ASTs into DCSTs that do not contain any disjunction nodes. Thus, it implements step 4 of Figure 1. Step 5 in this case is trivial—it is essentially the identity function—and is performed by Menhir’s settlement algorithm (§3).

A DCST Construction API The code in Figure 6 relies on a strongly-typed API for constructing DCSTs, offered by the module `DCST`, whose signature is shown in Figure 5. This module is generated by Menhir based on the grammar in Figure 3.

The module `DCST` offers a family of abstract types of DCSTs. For each nonterminal symbol A , there is an abstract type, also named A , of DCSTs whose head symbol is A . Furthermore, for these abstract types, the module `DCST` offers a family of constructor functions. For each production, there is a constructor function, whose name is determined by the production’s `@name` attribute. This constructor function takes parameters whose number and types match the right-hand side of the production. This explains why the constructor `const`, which corresponds to the production `factor: INT`, takes one argument of type `int`. If a parameter corresponds to a terminal symbol that does not carry a semantic value, such as `LPAR` or `RPAR`, then this parameter is omitted. This explains why the constructor `paren`, which corresponds to the production `factor: LPAR; expr; RPAR`, takes just one argument of type `expr`. Furthermore and finally, for each nonterminal symbol A , there is a constructor function that constructs a disjunction node. These functions are named `factor_choice`, `term_choice`, `expr_choice`, etc. The code in Figure 6 does not use these functions; their use is illustrated in the next section (§2).

The module `DCST` allows constructing DCSTs but offers no way of deconstructing or inspecting a DCST. The only way of doing anything useful with a DCST is to convert it to a CST. This is done by the settlement algorithm (§3).

Transforming ASTs to CSTs—More Comments Let us momentarily come back to the transformation in Figure 6. The code is straightforward. At each level A of the grammar, if the AST node at hand can be directly represented by a DCST constructor for the nonterminal symbol A , then this constructor is used; otherwise a recursive call to the next lower level is performed. A recursive call from the lowest level (`factor`) back to the highest level (`expr`) is made possible by the constructor `DCST.paren` (line 3), which corresponds to the production `factor: LPAR; expr; RPAR`.

The functions in Figure 6 always construct well-formed DCSTs. This property is enforced by the strongly-typed API of the module `DCST`. If the programmer mistakenly attempted to construct an ill-formed tree, this would be detected by the OCaml type-checker. For instance, attempting to call the function `expr` or `factor` in a position where one must call `term` would give rise to a static type error. This is a pleasant and valuable guarantee.

2 Working with Conflicts and Precedence Declarations

In practice, people commonly work with grammars that lie *outside* the class LR(1). Indeed, manually refactoring a grammar so as to fit in the class LR(1) is usually unpleasant, sometimes difficult, and can lead to a blow-up in the size of the grammar. Instead, following an approach first suggested by Aho, Johnson and Ullman [AJU75] and first implemented in `yacc` [Joh75], people often prefer to keep a non-LR(1) grammar and annotate it with

precedence declarations. When building an LR(1) automaton for such a grammar, the parser generator encounters *conflicts*, that is, situations where several actions are possible, such as “shift”, “reduce production p_1 ”, and “reduce production p_2 ”. The parser generator exploits precedence declarations to statically *resolve conflicts*: at parser construction time, in each situation where several actions are possible, at most one action is chosen. Thus, a deterministic LR(1) automaton is obtained.²

This automaton is a correct parser, but, because some legitimate actions have been removed, it is *incomplete*: there exist CSTs that it cannot construct. In other words, there exist non-viable CSTs. The trees shown in Figure 9 are examples: the parser will never construct them. As explained earlier, we do not trust the user to write a correct transformation of ASTs into *viable* CSTs. Indeed, because the user usually does not have good knowledge and understanding of the LR(1) automaton, it seems extremely difficult for her to ascertain that only viable CSTs are ever constructed. Instead, our approach is to ask the user to implement a transformation of ASTs into DCSTs and to provide an algorithm that transforms DCSTs into viable CSTs.

An Ambiguous Grammar and its Automaton This approach is illustrated by the grammar in Figure 7. This grammar does not distinguish several subcategories of expressions, such as `factor`, `term`, and `expr`: there is a single nonterminal symbol `expr`. This grammar is ambiguous. The precedence declarations on lines 7 and 8 indicate how to resolve the conflicts that appear when an LR(1) automaton for this grammar is constructed.

The LR(1) automaton that corresponds to this decorated grammar is shown in Figure 8. The vertices of this graph are the states of the automaton. Each state is identified by a number (in blue). Each state (except state 0, which is the initial state) also carries an *incoming symbol* (in red). This terminal or nonterminal symbol is in fact the label of every edge that enters this state. Edges labeled with a terminal symbol are shown as plain edges; edges labeled with a nonterminal symbol are dashed. Thus, starting in state 0, following the path labeled with `expr`; `ADD`; `expr` leads to state 8. State 8 is the only state where the automaton’s behavior depends on the lookahead symbol, that is, on the first unconsumed token of the input stream. If this symbol is `MUL` then the automaton consumes this symbol and follows the transition from state 8 to state 5. This is known as “shifting”. If this symbol is `ADD`, `RPAR`, or `EOL`, then the automaton does not consume this symbol, and reduces the production `expr`: `expr`; `ADD`; `expr`.³ This means that the automaton moves back by three steps, thereby going back to state 0 or to state 1, depending on its history, which is stored in its stack. There, it follows a transition labeled with the left-hand side of this production, namely `expr`, thereby reaching state 10 or state 3. This is known as taking a “goto” transition.

For the purposes of this paper, it does not matter how precedence declarations are used to resolve conflicts, because our settlement algorithm (§3) relies on a description of the deterministic LR(1) automaton and does not care how this automaton is constructed. Nevertheless, let us briefly explain the effect of the precedence declarations in Figure 7.

The declaration `%left ADD` suggests that “addition is left-associative”. Technically, this implies that in state 8, when the lookahead symbol is `ADD`, reduction must be preferred over shifting. Thus, because of the precedence declaration, a transition labeled `ADD` out of state 8, which could have existed, is suppressed. As a result, after reading `1 + 2`, if the next input symbols are `+ 3`, then the parser constructs an AST for the subexpression `1 + 2` and considers this subexpression as the first operand of the addition `_ + 3`.

Similarly, the declaration `%left MUL` implies that in state 6, when the lookahead symbol is `MUL`, reduction must be preferred over shifting. This causes the removal of a transition

²For example, at the time of writing, the precedence declarations found in the OCaml compiler’s parser involve 36 distinct levels and resolve shift/reduce conflicts in 142 states (out of 2068 states).

³For readability, in Figure 8, we write “reduce” but do not indicate which production must be reduced. This automaton is so simple that, in each state, at most one production can be reduced. We let the reader reconstruct which one.

```

1 %token<int> INT          (* Tokens *)
2 %token      ADD "+"
3 %token      MUL "*"
4 %token      LPAR "("
5 %token      RPAR ")"
6 %token      EOL
7 %left      ADD          (* Priority levels: weakest to strongest *)
8 %left      MUL
9 %start<AST.main> main   (* Start symbol *)
10 %{ open AST %}
11 %%                    (* Productions and semantic actions *)
12 %inline op:
13 | ADD          { BAdd }          [@name add]
14 | MUL          { BMul }          [@name mul]
15
16 expr:
17 | LPAR; e = expr; RPAR      { e }          [@name paren]
18 | i = INT           { EConst i }        [@name const]
19 | e1 = expr; op = op; e2 = expr { EBinOp (e1, op, e2) }
20
21 main:
22 | e = expr; EOL          { e }          [@name eol]
    
```

Figure 7. Non-stratified grammar: the grammar

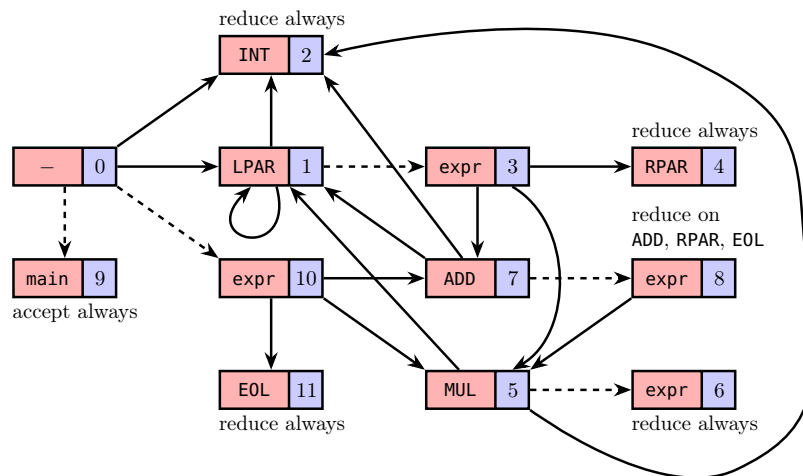


Figure 8. Non-stratified grammar: the LR(1) automaton

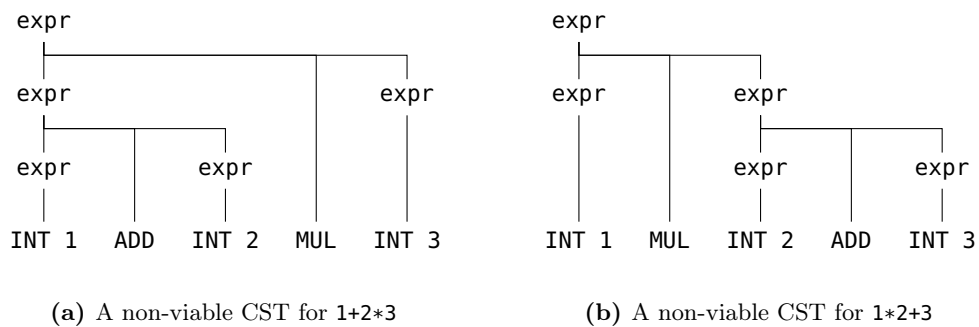


Figure 9. Non-stratified grammar: two non-viable concrete syntax trees


```

0 module DCST : sig
1   type expr
2   type main
3   val expr_choice: expr -> expr -> expr      (* Constructors for [expr] *)
4   val paren: expr -> expr
5   val const: (int) -> expr
6   val add: expr -> expr -> expr
7   val mul: expr -> expr -> expr
8   val main_choice: main -> main -> main     (* Constructors for [main] *)
9   val eol: expr -> main
10 end

```

Figure 10. Non-stratified grammar: signature of the module DCST

```

1 let possibly_paren (e : DCST.expr) : DCST.expr =
2   DCST.expr_choice e (DCST.paren e)
3
4 let rec expr (e : AST.expr) : DCST.expr =
5   possibly_paren @@
6   match e with
7   | EConst i          -> DCST.const i
8   | EBinOp (e1, BAdd, e2) -> DCST.add (expr e1) (expr e2)
9   | EBinOp (e1, BMul, e2) -> DCST.mul (expr e1) (expr e2)
10
11 and main          : AST.main -> DCST.main = function
12 | e                -> DCST.eol (expr e)

```

Figure 11. Non-stratified grammar: translating ASTs to DCSTs

labeled MUL out of state 6.

Finally, the order in which the two %left declarations appear suggests that “MUL takes precedence over ADD”. Technically, this implies that in state 8, when the lookahead symbol is MUL, shifting must be preferred over reduction, and that in state 6, when the lookahead symbol is ADD, reduction must be preferred over shifting. This causes the removal of a reduction action in state 8 and the removal of a transition labeled ADD out of state 6. Thus, for example, the text `1+2*3` gives rise to the concrete syntax tree depicted in Figure 4b.

Translating ASTs to DCSTs As in the previous section (§1), for the grammar of Figure 7, Menhir generates a module DCST, whose interface is shown in Figure 10. By relying on this API, the user is expected to implement a transformation of ASTs to DCSTs. Such a transformation appears in Figure 11. The main point of interest resides in the use of `possibly_paren` at line 5. Using the constructor function `DCST.expr_choice`, this auxiliary function constructs a disjunction node whose children are `e` and `DCST.paren(e)`. Thus, it expresses the fact that there is a choice between displaying the subtree `e` without parentheses and displaying it surrounded with parentheses.

This code constructs DCSTs that are analogous to the one shown in Figure 12a. There, a binary disjunction node is identified by the label `expr?`. This DCST is a DAG: some internal nodes are reachable via multiple paths. Because this DCST involves two disjunction nodes, it can be viewed as a compact description of a family of four different CSTs. Some members of this family are viable CSTs; some are not. For instance, the concrete syntax tree shown in Figure 12b is a viable member of this family: it is obtained by choosing the left child of the outermost disjunction node and the right child of the innermost disjunction node. On the other hand, the concrete syntax tree in Figure 9b, which is obtained by choosing the

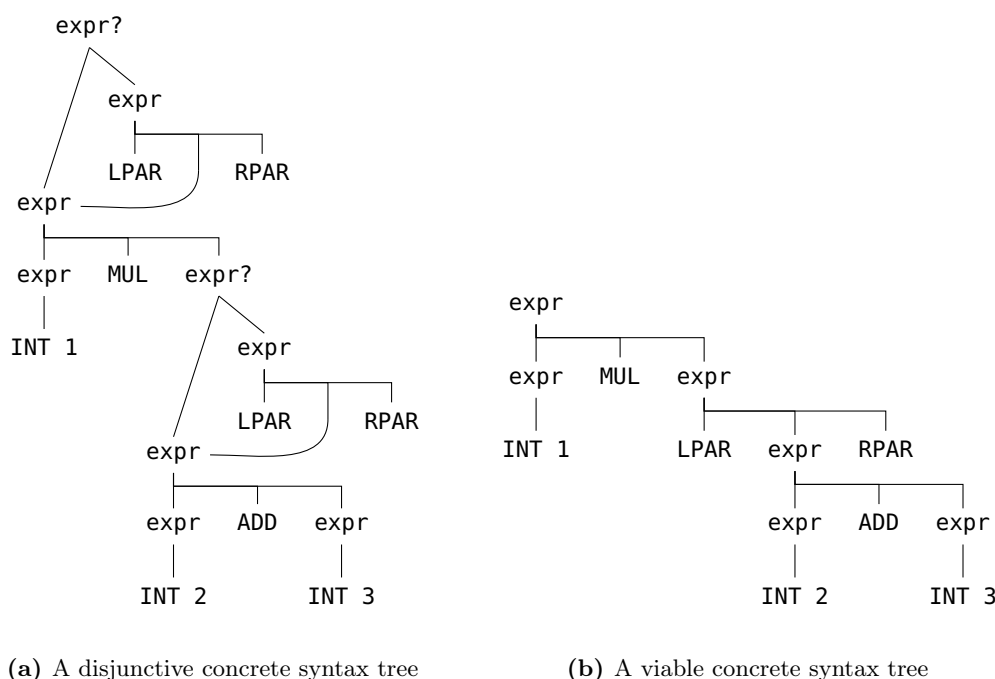


Figure 12. Non-stratified grammar: a DCST and one of the CSTs that it describes

left branch of both disjunction nodes, is a non-viable member of this family.

Let us say that a DCST is viable if at least one of the CSTs that it describes is viable. In our approach, it is up to the user to construct a viable DCST. The code in Figure 11 does this. Indeed, by using `possibly_paren` at every AST node, this code allows parentheses to be inserted at every node, and it is intuitively clear that if parentheses are inserted everywhere then a viable CST is obtained. If for some reason the user constructs a DCST that is not viable (for example, by forgetting to use `possibly_paren` at line 5) then the settlement algorithm (§3) fails. So, such a mistake is detected; however, it is detected only at runtime.

3 Settling Choices

The distinction between DCSTs and CSTs, and the settlement algorithm, which offers a bridge between these concepts, are the key technical contributions of this paper.

API The API of the settlement algorithm appears in Figure 14. The type of the function `Settle.main` reflects the purpose of this algorithm: it transforms a DCST whose head symbol is the start symbol `main` into some viable CST whose head symbol is also `main`. If it cannot do so, it reports a failure by returning `None`.

The API of the module `CST` is shown and discussed in the next section (§4). Let us point out that the type `CST.main` is abstract, so the user has no way of constructing CSTs besides invoking `Settle.main`. As a result, every value of type `CST.main` must be a *viable* CST.

Viability Relative to a Context Our discussion so far has relied on a simple abstract model of parsing, and our definition of viability has been based on this model. We have modeled a deterministic parser as a partial function *grow*: if *s* is a sequence of tokens then *grow(s)* is either undefined or a CST. Then, we have written that a CST *c* is viable if and only if the round-trip property $grow(\text{fringe}(c)) = c$ holds. This models how a parser behaves

```

0 module CST : sig
1   type expr
2   type main
3   class virtual [ 'r ] reduce : object      (* This class is used while formatting (§4) *)
4     method virtual zero : 'r              (* Document construction methods *)
5     method virtual cat : 'r -> 'r -> 'r
6     method virtual text : string -> 'r
7     method virtual visit_INT : int -> 'r (* Visitor methods for terminal symbols *)
8     method visit_ADD : 'r
9     method visit_MUL : 'r
10    method visit_LPAR : 'r
11    method visit_RPAR : 'r
12    method virtual visit_EOL : 'r
13    method visit_expr : expr -> 'r        (* Visitor methods for nonterminal symbols *)
14    method case_paren : expr -> 'r
15    method case_const : int -> 'r
16    method case_add : expr -> expr -> 'r
17    method case_mul : expr -> expr -> 'r
18    method visit_main : main -> 'r
19    method case_eol : expr -> 'r
20  end
21 end

```

Figure 13. Non-stratified grammar: signature of the module CST

```

0 module Settle : sig
1   val main: DCST.main -> CST.main option
2 end

```

Figure 14. Non-stratified grammar: signature of the module Settle

when it is applied to a complete input. However, a less abstract and more compositional view is now needed: we must discuss how an LR(1) parser behaves when it is applied to an input fragment.

An LR(1) automaton maintains a *stack*, which records what input fragment has been consumed and reflects how this input fragment has been interpreted. A stack is a sequence of *cells*, where a cell is a pair of a state and a CST: $\sigma ::= \epsilon \mid \sigma \cdot (q, c)$. In every cell (q, c) , the head symbol of the CST c is also the incoming symbol (§2) of the state q . We refer to it as *the cell symbol* of this cell. At every time, the stack σ forms a path in the automaton's state diagram. The stack σ determines the *current state* $cur(\sigma)$ of the automaton: it is the state found in the top stack cell. If the stack is empty, it is the initial state.

An LR(1) automaton also maintains a *remaining input*, a nonempty sequence of tokens. A pseudo-token $\#$ is used as a terminating sentinel. If κ is the first element of the remaining input then $sym(\kappa)$ is known as the *lookahead symbol*.

An LR(1) automaton makes progress via two kinds of actions. A *shift* action consumes a token κ and pushes the cell (q', c) onto the stack, where q' is the target of a transition labeled $sym(\kappa)$ out of the current state, and where c is the terminal node $T \kappa$. A *reduce* action for a production $A \rightarrow \alpha$ first pops several cells off the stack, whose sequence of cell symbols is α , and whose sequence of CSTs is \vec{c} . It then pushes the cell (q', c) onto the stack, where q' is the target of a transition labeled A out of the current state, and where c is the nonterminal node $N(A \rightarrow \alpha, \vec{c})$. At every step, the choice of the next action is determined solely by the current state and by the lookahead symbol.

The behavior of an LR(1) automaton, when faced with a certain input *segment*, depends

both on the input that the parser has read before reaching this segment, which is summarized by the automaton’s current state, and on the lookahead symbol, that is, on the first input symbol that lies beyond this segment. Therefore, when looking at a CST c that is intended to correspond to a segment of the input, it does not make sense to ask whether this CST is “viable” without any qualification. Instead, one should ask whether c is viable *with respect to a certain state q and a certain lookahead symbol b* . This notion can be defined as follows:

Definition 1 (Relative Viability). A CST c is *viable* with respect to (q, b) iff for every stack σ such that $cur(\sigma) = q$ and for every nonempty sentence $\kappa \cdot s$ such that $sym(\kappa) = b$, the automaton, beginning with the stack σ and the remaining input $fringe(c) \cdot \kappa \cdot s$, reaches (in zero or more steps) the stack $\sigma \cdot (q', c)$ and the remaining input $\kappa \cdot s$.

In this definition, because the stack $\sigma \cdot (q', c)$ must correspond to a path in the automaton’s state diagram, the state q' must be the target of a transition labeled $head(c)$ out of q . Therefore, for c to be viable with respect to (q, b) , it is necessary that there exist a transition labeled $head(c)$ out of q .

In short, c is viable with respect to (q, b) if and only if the automaton, beginning in state q and faced with the fringe of c followed with the lookahead symbol b , consumes precisely this fringe and constructs precisely the tree c , which it pushes onto its stack.

Our earlier notion of “viability” without qualification corresponds to viability with respect to the automaton’s initial state q_0 and sentinel symbol $\#$.

Suppose c is a nonterminal node $N(A \rightarrow \alpha, \vec{c})$. If c is viable with respect to (q, b) then the following two facts must be true: (1) out of the state q , there exists a path, whose edge labels form the string α , to some state q' ; (2) in state q' , with lookahead symbol b , the automaton is willing to reduce the production $A \rightarrow \alpha$. When both conditions are satisfied, we say that $A \rightarrow \alpha$ is *apparently viable* with respect to (q, b) . This is a necessary condition for c to be viable with respect to (q, b) .

Compositional Specification of a Settlement Algorithm A compositional settlement algorithm solves *problems* of the form (d, q, b) where d is a DCST, q is a state, and b is a terminal symbol. A *solution* to such a problem is a CST c such that $c \in d$ and c is viable with respect to (q, b) . A settlement algorithm can also return \perp , which represents a failure. Because a solution can exist only if the state q has an outgoing transition labeled $head(d)$, we restrict our attention to problems that satisfy this condition. A *complete* algorithm fails only when no solution exists; an *incomplete* algorithm can fail even if a solution exists.

An Eager Settlement Algorithm A linear-time, incomplete settlement algorithm *settle* can be described as follows. Assume the problem is (d, q, b) , where q has an outgoing transition labeled $head(d)$. Then, compute $settle(d, q, b)$ as follows:

1. If d is a terminal node $T \kappa$, return this terminal node.
2. If d is a nonterminal node $N(A \rightarrow \alpha, \vec{d}')$,
 - a) Verify that $A \rightarrow \alpha$ is apparently viable with respect to (q, b) . If not, fail.
 - b) Settle each of the subtrees \vec{d}' , *from right to left*, via recursive calls to *settle*, yielding a sequence of CSTs \vec{c}' . Then, return the nonterminal node $N(A \rightarrow \alpha, \vec{c}')$.
3. If d is a disjunction node $d_1 \vee d_2$, where d_1 is a nonterminal node $N(A \rightarrow \alpha, \vec{d}')$,
 - a) If $A \rightarrow \alpha$ is apparently viable with respect to (q, b) then return $settle(d_1, q, b)$;
 - b) otherwise return $settle(d_2, q, b)$.

More explanations about step **2b** are needed. The check in step **2a** guarantees that there exists a path out of the state q whose edge labels form the string α . This is also the string of the head symbols of the subtrees \vec{d}' . The source states of the edges along this path are the states that must be used in the recursive calls to *settle*. A key remark is that these states are known before any of the recursive calls takes place. So, there is no requirement that the recursive calls be performed from left to right. We perform them from

right to left, which is crucial, because the result of settling a subtree is needed to compute the lookahead symbol that must be used while settling the next subtree towards the left. Formally, if the sequence \vec{d}' is decomposed as $\vec{d}'_1 \cdot d' \cdot \vec{d}'_2$, and if settling \vec{d}'_2 has produced \vec{c}'_2 , then the lookahead symbol that must be passed as an argument to the next recursive call, $settle(d', -, -)$, is the first element of the nonempty sequence $sym(fringe(\vec{c}'_2)) \cdot b$.

In step 3, we assume that the left disjunct d_1 is a nonterminal node. This causes no loss of generality. Indeed, a disjunction node whose disjuncts are terminal nodes is pointless; and a disjunction node whose left-hand child is itself a disjunction can be re-associated.

This algorithm is correct: assuming that q has an outgoing transition labeled $head(d)$, if $settle(d, q, b)$ returns c then $c \in d$ holds and c is viable with respect to (q, b) .

This algorithm runs in time $O(|d|)$, that is, in linear time in the tree size of the DCST d . In steps 3a and 3b, it is crucial that $settle$ is recursively applied to d_1 or d_2 , but not both. Thus, even if d_1 and d_2 share a subtree, this subtree is examined only once by the algorithm. The tests in steps 2a and 3a can be carried out in constant time, provided certain tables are precomputed. In step 2b, in order to efficiently compute the lookahead symbol that must be passed to each recursive call, it is convenient to adopt the convention that $settle(d, q, b)$ returns not only a CST c , but also the first symbol of the sequence $sym(fringe(c)) \cdot b$.

This algorithm is unfortunately incomplete. The source of incompleteness lies in step 3a. The apparent viability condition in step 3a is necessary for the recursive call $settle(d_1, q, b)$ to succeed, but does not guarantee that this call will succeed. If this call fails, then $settle(d_1 \vee \vec{d}'_2, q, b)$ fails as well, even though the subproblem (d_2, q, b) may have a solution. In other words, the algorithm is incomplete because it commits to the left-hand disjunct as soon as the apparent viability test succeeds, even though apparent viability does not imply viability. Completeness can be obtained by adding a backtracking mechanism: in step 3a, if $settle(d_1, q, b)$ fails, then return $settle(d_2, q, b)$. With this modification, the algorithm is complete, but has exponential time complexity, because d_1 and d_2 can (and usually do) share subtrees (recall Figure 12a).

A Memoizing Settlement Algorithm A complete settlement algorithm whose worst-case time complexity is polynomial can be given. This algorithm decomposes the original problem into subproblems of the form (d, q, a, b) . A solution of such a subproblem is a CST c such that $c \in d$ and c is viable with respect to (q, b) and the first symbol of $fringe(c) \cdot b$ is a . Memoization ensures that each subproblem is considered at most once. Compared with the eager algorithm, this algorithm requires a more complex internal representation of DCSTs: every DCST node must carry a unique identifier, and its “nullable” flag and “first” set must be precomputed. An apparent viability test is not needed, but can be used to speed up this algorithm.

Because this algorithm spends constant time and constant space on each subproblem, its time and space complexity is the number of subproblems that can ever be considered. Thus, in the worst case, it is $O(\#d \cdot |Q| \cdot |\Sigma|^2)$, where $\#d$ is the DAG size of the DCST d , $|Q|$ is the number of states of the LR(1) automaton, and $|\Sigma|$ is the number of terminal symbols. In practice, we expect that for each node d' in the DAG d , the number of subproblems (d', q, a, b) that are ever considered is independent of the size of the alphabet and of the size of the automaton. We have experimentally confirmed this expectation: for the grammar of Figure 7, we find that in practice this number is at most 3. Therefore, the time and space complexity of this algorithm is effectively linear in $\#d$.

Comparing Settlement Algorithms We have implemented both algorithms and performed a limited evaluation of their performance, based on the grammar of Figure 7 and on randomly-generated arithmetic expressions. As expected, both algorithms exhibit linear time complexity. The eager algorithm appears to be much faster in practice: taking into account both DCST construction time and settling time, the eager algorithm outperforms the memoizing algorithm by a factor of 15. We believe that the high cost of the memoizing

algorithm is due, to a large extent, to memoization itself: even if the cost of solving subproblems is discounted, the cost of storing and fetching data in and out of a large hash table seems very high.

Choosing an Algorithm In summary, the eager algorithm is fastest, but incomplete; the memoizing algorithm is complete, but significantly slower; and the eager algorithm with backtracking is complete and usually fast, but can be exponentially slow in pathological cases. At the time of writing, Menhir offers the eager algorithm only. This could change in the future, based on user feedback.

To what extent could the incompleteness of the eager algorithm be a problem in practice? We do not know yet. For the grammar of Figure 7, it is not a problem: indeed, for this simple grammar, apparent viability implies viability. However, this implication is not obvious. Furthermore, we have constructed more complex grammars where this is not the case. More experience is required to tell whether users are likely to face difficulties caused by incompleteness.

The eager algorithm is not optimal: it does not necessarily produce the smallest viable CST c such that $c \in d$. Instead, in every disjunction node, it favors the left disjunct. The code in Figure 11 relies on this behavior: at line 2, the left disjunct proposes not to insert parentheses, while the right disjunct inserts parentheses. Thus, in this simple example, optimality is achieved anyway. Regarding the memoizing algorithm, we believe that there are two variants of it: a left-biased variant, which we have implemented, and an optimal variant, which we have not implemented, and which we imagine will be slightly more expensive.

4 Formatting and Rendering

Once a (viable) CST has been constructed, there remains to transform it into a printable form, such as string or a `PPrint` document. This corresponds to steps 6 and 7 in Figure 1.

To allow this, we could expose an API that allows deconstructing CSTs, and let the user implement this transformation entirely on her own. We could for instance present the types `CST.expr` and `CST.main` as *private* algebraic data types, that is, algebraic data types whose values can be inspected via `match` constructs, but cannot be constructed. However, this approach would have two drawbacks. First, it would leave to the user the rather dreary task of implementing step 6, whereas one might expect this task to be performed by generated code. Second, it would force us to expose a grammar-specific representation of CSTs in memory, thereby preventing us from internally using a different, grammar-independent representation, or forcing us to transform one representation into the other.

Visitors to the Rescue Our solution to these problems is to present `CST.expr` and `CST.main` as *abstract types* and to generate code for step 6 in such a form that the behavior of this code can be customized. Indeed, not everything about step 6 is mechanical. For one thing, we wish to let the user pick the return type of this transformation: strings, `PPrint` documents, or some other type of her choosing. Furthermore, should the user decide to produce `PPrint` documents, we wish to let her customize the layout of these documents: she must decide where and how to use the document-construction combinators offered by the `PPrint` library, which offer fine-grained control over indentation, line breaks, and so on. To allow customization, we generate a *visitor class* whose methods reduce a CST to a printable representation. For the grammar of Figure 7, this API is shown in Figure 13.

The module `CST`, whose code is generated by Menhir, advertises `expr` and `main` as abstract types. Then, it advertises a visitor class, `reduce`. This class is parameterized by a type `'r`, which is the result type of every method: it is the type of the printable form to which every tree must be reduced. The user is expected to provide implementations for the three virtual methods `zero`, `cat`, and `text`. In short, `zero` is the empty printable thing; `cat` concatenates

```

1 class print = object
2   inherit [string] CST.reduce
3   method zero = ""
4   method cat = (^)
5   method text s = s
6   method visit_INT i = Printf.sprintf "%d" i
7   method visit_EOL = "\n"
8 end
9
10 let main (m : CST.main) : string =
11   (new print)#visit_main m

```

Figure 15. Non-stratified grammar: translating CSTs to strings

```

1 open PPrint
2 let lparen = lparen ^^ ifflat empty space
3 let rparen = ifflat empty hardline ^^ rparen
4 class print = object (self)
5   inherit [document] CST.reduce as super
6   method zero = empty
7   method cat = (^^)
8   method text = string
9   method visit_INT i = utf8format "%d" i
10  method! visit_ADD = space ^^ plus ^^ break 1
11  method! visit_MUL = space ^^ star ^^ break 1
12  method visit_EOL = hardline
13  method! visit_expr e = group (super#visit_expr e)
14  method! case_paren e = nest 2 (lparen ^^ self#visit_expr e) ^^ rparen
15 end
16
17 let main (m : CST.main) : document =
18   (new print)#visit_main m

```

Figure 16. Non-stratified grammar: translating CSTs to PPrint documents

```

# Printing as a string:
65*((22+38)*69+(24+58))+(84*70+(20+63*83*97+49*(70+0))*(93+89)*(12*15+85+21))
# Printing via PPrint in 20 columns:
65 *
( (22 + 38) * 69 +
  (24 + 58)
) +
( 84 * 70 +
  ( 20 +
    63 * 83 * 97 +
    49 * (70 + 0)
  ) *
  (93 + 89) *
  ( 12 * 15 + 85 +
    21
  )
)
)

```

Figure 17. Sample output produced by the code in Figures 15 and 16

two printable things; and `text` converts a string to a printable thing. Furthermore, the following methods exist:

- For each terminal symbol, there is a visitor method: these include `visit_INT`, `visit_ADD`, and so on. If the terminal symbol carries no semantic value and if a “token alias” (a concrete string) has been provided by the user for this symbol (Figure 7) then a default implementation of this method is generated by Menhir; this default implementation relies on the method `text`. Otherwise, this method is declared virtual.
- For each nonterminal symbol, there is a visitor method: these are `visit_expr` and `visit_main`. These methods expect a CST as an argument, perform a case analysis of the root node, and apply a suitable `case` method to the children of the root node.
- For each production, there is a visitor method: these include `case_paren`, `case_const`, and so on. These methods expect zero, one, or more arguments. First, each argument is reduced to a printable thing via recursive calls to suitable `visit` methods; then these printable things are concatenated using `zero` and `cat`.

Examples of Use Figure 15 shows how a user may rely on the class `CST.reduce` to convert CSTs to strings. The user defines a new class, `print`, which inherits `reduce` where the type parameter `r` is instantiated with `string`. Then, the user supplies trivial implementations of the methods `zero`, `cat`, and `text`: the empty string, string concatenation, and the identity function. Finally, the user provides implementations of the virtual methods `visit_INT` and `visit_EOL`.⁴ Then, the method call `(new print)#visit_main` converts a CST to a string.⁵

Figure 16 shows how a user may rely on the class `CST.reduce` to convert CSTs to `PPrint` documents. The idea is the same. This time, the type parameter `r` is instantiated with the type `PPrint.document`. The document construction combinators offered by `PPrint` are used in a few key places. For example, by overriding the method `visit_expr`, the user indicates that a `group` combinator, which offers a choice between flat and non-flat output, should be used at every subexpression. By overriding the method `case_paren`, the user customizes the manner in which parenthesized subexpressions are indented. An example of the output produced by this code appears in Figure 17.

5 Related Work

van den Brand and Visser [vdBV96] describe a tool that generates “formatters” (that is, unparsers and pretty-printers) for ASF+SDF grammars. An ASF+SDF grammar is a context-free grammar, augmented with priority and associativity declarations. It does not have semantic actions, but a production can be marked with the attribute `bracket`, which means that no AST node must be produced. Thus, an unparser must decide where to insert parentheses. van den Brand and Visser spell out the round-trip property that a correct unparser must obey. They provide a brief description of a (linear-time, bottom-up) unparser of arithmetic expressions, but do not describe how they deal with the general case. They separate unparsing and layout in the same way as we do (§3, §4).

Ramsey [Ram98] offers a detailed presentation of a (linear-time, bottom-up) unparser. The design and correctness proof of his algorithm are guided by thinking about the behavior of a shift-reduce parser. This is also the case in this paper: in fact, at runtime, our settlement algorithm relies directly on the parser’s tables. Ramsey supports a single syntactic category of

⁴Because Menhir does not allow a token alias to contain a newline character, the token alias “\n” cannot be associated with the symbol `EOL`.

⁵This example happens to “just work” because two consecutive tokens can be safely concatenated: for this simple language of arithmetic expressions, *in a grammatically valid sentence*, there is no possibility for the concatenation of two tokens to be confused by the lexer with some other token. In a more complex scenario, greater care would be required. The type of printable things would be more elaborate than `string`, and the `cat` method would sometimes insert a space between two things.

“expressions” with infix, prefix, and postfix operators whose precedence and associativity are specified by the user. We deal with the more general case of an arbitrary LR(1) automaton, yet our eager settlement algorithm is arguably simpler than Ramsey’s unparsing. Our eager algorithm is incomplete in general, but we believe that it is complete in the restricted setting considered by Ramsey. We sketch (and have implemented) a slower, complete algorithm.

Bouwers, Bravenboer, and Visser [BBV08] regret that the style of precedence declarations supported by most LR(1) parser generators, including `yacc`, `bison`, and `Menhir`, does not have a clear semantics. To clarify this semantics, and to help compare two parsers that are based on different methodologies, they propose “a core formalism for defining precedence rules”, based on “tree patterns” of bounded depth. They describe a “precedence rule recovery tool”, which tests whether a parser, viewed as a black box, can or cannot construct certain tree patterns. Thanks to this tool, they discover discrepancies between several parsers for C99 and PHP5. The question that their tool answers, “can this parser possibly construct a tree of this shape?”, is closely related to the question that our settlement algorithm addresses. That said, we believe that the answer to such a question can depend on the current state of the LR(1) automaton: that is, it can be the case that in a certain state q the parser will construct a tree of a certain shape whereas in some other state q' it will refuse to construct a tree of this shape. Another source of worry is Bravenboer *et al.*’s remark that tree patterns of depth 1 are often sufficient, yet depths 2 and 3 are sometimes needed. Thus, it is not clear whether (or under what conditions) Bravenboer *et al.*’s tree pattern formalism is capable of accurately describing the behavior of an LR(1) parser. This open question could be related to the open question of clarifying under what conditions “apparent viability” (§3) implies viability. Indeed, apparent viability is an inspection of the tree at depth 1.

Danielsson [Dan13] discusses the construction of pretty-printers in a dependently-typed programming language. He uses dependent types to express and enforce the end-to-end round-trip property $parse(unparse(t)) = t$. In contrast, we guarantee the more limited property $grow(fringe(c)) = c$, and we rely on (generated) abstract types to ensure that the user can construct only grammatically valid DCSTs and CSTs (see Figures 5 and 6).

Several authors have investigated the idea of producing parsers and pretty-printers out of a single “invertible syntax description” [RO10, ANO12, MW13, ZZK⁺16, ZKZ⁺20]. This approach promises to reduce redundancy and guarantee that a round-trip property holds by construction. Our starting point in this paper is an existing parser generator that allows arbitrary semantic actions, so we cannot follow this approach.

6 Conclusion

We have described several new features offered by `Menhir` to support fast and correct unparsing. `Menhir` now generates type-safe, grammar-specific DCST construction and CST deconstruction APIs. A settlement algorithm, which transforms DCSTs into CSTs, is implemented once and for all in a library. Because it relies directly on a description of the LR(1) automaton, it supports arbitrary precedence declarations and all LR(1) construction methods. The concept of a DCST, as well as our settlement algorithms, are new.

The most unpleasant aspect of our approach is perhaps the need for the user to manually implement the translation of ASTs to (viable) DCSTs. In the future, it would be desirable to automate this transformation; this should be possible, provided the expressive power of semantic actions is restricted. Another limitation is the incompleteness of our fast settlement algorithm. Whether it is a problem in practice remains to be determined: we do not yet have practical experience with this new tool. In the future, we would like to allow decorating trees with user-specified data, such as comments or source code locations, so as to allow “high-fidelity” program transformations [dJV11, AvdS20].

References

- [AJU75] Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. [Deterministic parsing of ambiguous grammars](#). *Communications of the ACM*, 18(8):441–452, 1975.
- [ANO12] Reynald Affeldt, David Nowak, and Yutaka Oiwa. [Formal network packet processing with minimal fuss: invertible syntax descriptions at work](#). In *Programming Languages Meets Program Verification (PLPV)*, pages 27–36, 2012.
- [AvdS20] Rodin T. A. Aarssen and Tijs van der Storm. [High-fidelity metaprogramming with separator syntax trees](#). In *Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 27–37, January 2020.
- [BBV08] Eric Bouwers, Martin Bravenboer, and Eelco Visser. [Grammar engineering support for precedence rule recovery and compatibility checking](#). *Electronic Notes in Theoretical Computer Science*, 203(2):85–101, 2008.
- [Dan13] Nils Anders Danielsson. [Correct-by-construction pretty-printing](#). In *Workshop on dependently-typed programming*, pages 1–12, September 2013.
- [dJV11] Maartje de Jonge and Eelco Visser. [An algorithm for layout preservation in refactoring transformations](#). In *Software Language Engineering*, volume 6940 of *Lecture Notes in Computer Science*, pages 40–59. Springer, July 2011.
- [GJ08] Dick Grune and Ceriel J. H. Jacobs. *Parsing techniques: a practical guide, second edition*. Monographs in computer science. Springer, 2008.
- [Joh75] Stephen C. Johnson. [Yacc: Yet another compiler-compiler](#). Computing Science Technical Report 32, Bell Laboratories, 1975.
- [Knu65] Donald E. Knuth. [On the translation of languages from left to right](#). *Information & Control*, 8(6):607–639, December 1965.
- [KVV10] Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. [Pure and declarative syntax definition: Paradise lost and regained](#). In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 918–932, 2010.
- [MW13] Kazutaka Matsuda and Meng Wang. [FliPpr: A prettier invertible printing system](#). In *European Symposium on Programming (ESOP)*, volume 7792 of *Lecture Notes in Computer Science*, pages 101–120. Springer, March 2013.
- [Pot23] François Pottier. The PPrint pretty-printing library, 2007–2023. <https://github.com/fpottier/pprint/>.
- [PRG23] François Pottier and Yann Régis-Gianas. The Menhir parser generator, 2005–2023. <https://gitlab.inria.fr/fpottier/menhir/>.
- [Ram98] Norman Ramsey. [Unparsing expressions with prefix and postfix operators](#). *Software: Practice and Experience*, 28(12):1327–1356, 1998.
- [RO10] Tillmann Rendel and Klaus Ostermann. [Invertible syntax descriptions: unifying parsing and pretty printing](#). In *Symposium on Haskell*, pages 1–12, September 2010.
- [vdBV96] Mark van den Brand and Eelco Visser. [Generation of formatters for context-free languages](#). *ACM Transactions on Software Engineering and Methodology*, 5(1):1–41, 1996.

- [ZKZ⁺20] Zirun Zhu, Hsiang-Shang Ko, Yongzhe Zhang, Pedro Martins, João Saraiva, and Zhenjiang Hu. [Unifying parsing and reflective printing for fully disambiguated grammars](#). *New Generation Computing*, 38(3):423–476, 2020.
- [ZZK⁺16] Zirun Zhu, Yongzhe Zhang, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. [Parsing and reflective printing, bidirectionally](#). In *Software Language Engineering*, pages 2–14, November 2016.

Source-to-Source Optimizations Validated using Separation Logic

Guillaume Bertholon^{1,2}, Arthur Charguéraud^{2,1}, and Thomas
Kœhler^{2,1}

¹Université de Strasbourg, CNRS, ICube, France

²Inria, France

We present a demo of OptiTrust, an interactive framework for optimizing general-purpose programs via series of programmer-guided, source-to-source transformations. Optimization steps are described in transformation scripts, expressed as OCaml programs. At every step, the programmer may interactively visualize the effect of the transformation as the difference between two pieces of human-readable code. The framework currently applies to C code. That said, our internal AST is based on the imperative lambda-calculus, thus we expect it to be straightforward to extend OptiTrust for optimizing OCaml code.

A central question is how to verify that nontrivial transformations preserve the semantics of the code. To that end, we require the programmer to provide Separation Logic annotations, and use them to compute resource usage throughout the code. As we show, this information suffices to justify the correctness of numerous essential source-to-source transformations, such as swapping of instructions, swapping of loops, hoisting of instructions out of loops, etc.

Producing high-performance code is critical in many domains, in particular numerical simulation, image processing and machine learning. Yet, achieving high-performance on modern hardware is a very challenging task.

A common industrial practice is to optimize code *by hand* in languages such as C/C++ [TV14, MLP⁺17] or Fortran [VD18]. On the one hand, the process of manually rewriting code is highly time-consuming. In particular, several optimization paths must be empirically explored because code performance is hard to assess without benchmarking. On the other hand, the output of manual rewriting is very unsatisfying: the optimized code is much longer than the original, hard to read, hard to maintain, hard to adapt to other hardware, and—worst of all—is highly likely to contain bugs.

Another common industrial practice consists of using DSLs, such as Halide [RKBA⁺13] or TVM [CMJ⁺18]. A DSL consists of a programming language restricted to a particular application domain, accompanied with a compiler providing specific optimizations for that language. The main limitation of that approach is that it falls completely short when the programmer requires features that are just outside of the scope of DSL.

OptiTrust is an optimization framework that aims to support optimization of general-purpose code. OptiTrust operates by applying a series of source-to-source transformations, guided by the programmer. The programmer interactively develops a transformation script. At every transformation step, the programmer may visualize the corresponding *diff* between pieces of human-readable code.

Prior work on OptiTrust [CBRB22] has demonstrated its ability to reproduce a state-of-the-art optimized implementation of a numerical Particle-In-Cell simulation, as well as an

```

void mm(float* C, float* A, float* B, int m, int n, int p) {
  __reads("A ~ Matrix2(m, p), B ~ Matrix2(p, n)");
  __modifies("C ~ Matrix2(m, n)");
  for (int i = 0; i < m; i++) {
    __modifies("Group(range(0, n, 1), fun j -> &C[i][j] ~ Cell)");
    __seq_reads("A ~ Matrix2(m, p), B ~ Matrix2(p, n)");
    for (int j = 0; j < n; j++) {
      __modifies("&C[i][j] ~ Cell");
      __seq_reads("A ~ Matrix2(m, p), B ~ Matrix2(p, n)");
      float sum = 0.0f;
      for (int k = 0; k < p; k++) {
        __GHOST_BEGIN(focusA, matrix2_ro_focus, "A, i, k");
        __GHOST_BEGIN(focusB, matrix2_ro_focus, "B, k, j");
        sum += A[i][k] * B[k][j];
        __GHOST_END(focusA);
        __GHOST_END(focusB);
      }
      C[i][j] = sum;
    }
  }
}

void mm1024(float* C, float* A, float* B) {
  __reads("A ~ Matrix2(1024, 1024), B ~ Matrix2(1024, 1024)");
  __modifies("C ~ Matrix2(1024, 1024)");
  mm(C, A, B, 1024, 1024, 1024);
}

```

Figure 1. Unoptimized code for matrix multiplication: A and B denote the input, and C the output; and mm1024 specializes input sizes to 1024.

optimized matrix multiplication kernel generated by the specialized compiler TVM [CMJ⁺18]. However, code transformations in that prior work were *unchecked*, in the sense that the user could request transformations that do not preserve the semantics of the code.

The present work aims at equipping OptiTrust with means of validating the correctness of the transformations described in the user’s scripts. This validation process leverages Separation Logic shape information. To compute this information at every program point, we require the user to provide logical annotations on functions and loops (as well as on function calls with nontrivial instantiation of *auxiliary variables*). These annotations may be viewed either as a weak form of Separation Logic (describing shapes of data structures but not specifying the values stored inside), or as a strong form of typing (more expressive than Rust). As we show, this information suffices to verify transformations such as reordering of instructions or loops, or loop parallelization. Beyond the validation of transformations, another central aspect of our work is to show how the loop annotations can be automatically maintained through a series of transformations affecting the loops. Through our demo, we will highlight the key components of OptiTrust:

1. An internal, simplified abstract syntax tree (AST) of an imperative lambda-calculus, from which readable C code can be recovered throughout transformations. The parsed source code is encoded into this OptiTrust AST, pushing all mutable variables behind a pointer. Then, the AST can be pretty-printed back to C code, using annotations to disambiguate between patterns that are encoded in the same way. A similar idea of a simplified AST has been employed in the Cetus project [DBM⁺09].
2. An annotation language for lightweight separation logic predicates [Cha20]. These annotations include *function contracts*, *loop contracts*, and *ghost code*. Function contracts specify the ownership and the shape of the data accessible from pointers. Ghost code are annotations that allow to reorganize the view on a data structure. Loops contracts are optional. They can be used in particular to check if loops are parallelizable.
3. A system to compute available separation logic resources at every program point and check specifications. This can be compared to the Rust type system, yet with a more explicit management of aliasing through fractional permissions, or with RefinedC [SLK⁺21] with less automation but more flexibility thanks to expressive ghost code annotations.

<pre> void mm1024(float* C, float* A, float* B) { __modifies("C ~> Matrix2(1024, 1024)"); __reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)"); for (int i = 0; i < 1024; i++) { __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)"); __modifies("Group(range(0, 1024, 1), fun j -> &C[i][j] ~> Cell)"); for (int j = 0; j < 1024; j++) { __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)"); __modifies("&C[i][j] ~> Cell"); float sum = 0.f; for (int k = 0; k < 1024; k++) { __GHOST_BEGIN(focusA, matrix2_ro_focus, "M := A, i := i, j := k"); __GHOST_BEGIN(focusB, matrix2_ro_focus, "M := B, i := k, j := j"); sum += A[i][k] * B[k][j]; __GHOST_END(focusB); __GHOST_END(focusA); } C[i][j] = sum; } } } </pre>	<pre> void mm1024(float* C, float* A, float* B) { __modifies("C ~> Matrix2(1024, 1024)"); __reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)"); for (int i = 0; i < 1024; i++) { __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)"); __modifies("Group(range(0, 1024, 1), fun j -> &C[i][j] ~> Cell)"); __GHOST_BEGIN(tile_divides, "tile_count := 32, tile_size := 32, n := 1024, to_item := " "fun j -> &C[i][j] ~> Cell, bound_check := checked"); for (int bj = 0; bj < 32; bj++) { __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)"); __modifies("Group(range(0, 32, 1), fun j -> &C[i][bj * 32 + j] ~> Cell)"); for (int j = 0; j < 32; j++) { __seq_reads("A ~> Matrix2(1024, 1024), B ~> Matrix2(1024, 1024)"); __modifies("&C[i][bj * 32 + j] ~> Cell"); float sum = 0.f; for (int k = 0; k < 1024; k++) { __GHOST_BEGIN(focusA, matrix2_ro_focus, "M := A, i := i, j := k"); __GHOST_BEGIN(focusB, matrix2_ro_focus, "M := B, i := k, j := bj * 32 + j"); sum += A[i][k] * B[k][bj * 32 + j]; __GHOST_END(focusB); __GHOST_END(focusA); } C[i][bj * 32 + j] = sum; } } __GHOST_END(tile_divides); } } </pre>
--	---

Figure 2. Transformation: `Loop.tile (int 32)-index:"bj"-bound:TileDivides [cFor "j"]`. Replace the loop on `j` of 1024 iterations, with two nested loops on `bj` and `j` of 32 iterations each. The inserted operation `tile_divides` is a ghost instruction.

4. A system for targeting program points, somewhat similar to XPath [CD⁺99] for XML, but specialized for an AST. This system allows to describe, in a concise and robust manner, one or several program points to transform.
5. A library of general-purpose transformations that can leverage resources computed at each program point to justify their correctness, including: core data layout transformations [SSH10], instruction-level transformations [AK02], control flow transformations [Wol95]. Transformation also preserve or adapt annotations on functions and loops, and insert ghost operations if needed.
6. A scripting language, embedded in OCaml, for describing transformations. Transformation scripts are a classic technique for programmer-guided optimization frameworks [BHRS08, BZHB16, NS16, ZCG18, CCH08, RKH⁺11, BC20, YQ08, YWC14]. Transformation scripts provide fine-grained control, and they allow to chain large numbers of transformation steps.
7. An interactive interface allowing to visualize code *diffs* associated with the transformation at a given line of the script, via a key shortcut in the code editor.

Our running example will consist of an optimization script for a matrix multiplication function. Figure 1 shows the input code, including its annotations. In particular, each function and each loop has a contract (`seq_reads(H)` indicates that every loop iteration reads the full resource `H`, whereas `reads(Hi)` indicates that only the `i`-th iteration reads the resource `Hi`.) We will explain how to annotate the C code with our lightweight Separation Logic predicates, and explain how OptiTrust type-checks the code to compute resources at every program point. We will then present the transformation script, and explain how transformations are implemented. Figure 2 shows the example of a *tiling transformation*. The transformation does not improve performance by itself but allows for efficient parallelization schemes in a subsequent transformation (like using SIMD instructions for the inner loop).

We hope to engage with the JFLA community, in particular on the benefits of exploiting lightweight Separation Logic, as well as on potential applications of source-to-source transformations on C and OCaml programs.

References

- [AK02] J. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. 2002.
- [BC20] João Bispo and João MP Cardoso. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX*, 12:100565, 2020.
- [BHRS08] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI'08 ACM Conf. on Programming language design and implementation*, 2008.
- [BZHB16] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. Opening Polyhedral Compiler's Black Box. In *IEEE/ACM International Symp. on Code Generation and Optimization*, March 2016.
- [CBRB22] Arthur Charguéraud, Begatim Bytyqi, Damien Rouhling, and Yann A Barsamian. OptiTrust: an Interactive Framework for Source-to-Source Transformations. working paper, September 2022.
- [CCH08] Chun Chen, Jacqueline Chame, and Mary W. Hall. CHiLL: A framework for composing high-level loop transformations. Technical Report 08-897, University of Southern California, Jun 2008.
- [CD⁺99] James Clark, Steve DeRose, et al. XML path language (xpath), 1999.
- [Cha20] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP), aug 2020.
- [CMJ⁺18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *OSDI*, 2018.
- [DBM⁺09] Chirag Dave, Hansang Bae, Seung-Jai Min, Seyong Lee, Rudolf Eigenmann, and Samuel Midkiff. Cetus: A source-to-source compiler infrastructure for multicores. *Computer*, 42(12):36–42, 2009.
- [MLP⁺17] Suejb Memeti, Lu Li, Sabri Pllana, Joanna Kołodziej, and Christoph Kessler. Benchmarking opencl, openacc, openmp, and cuda: programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pages 1–6, 2017.
- [NS16] Kedar S. Namjoshi and Nimit Singhania. Loopy: Programmable and formally verified loop transformations. In *Static Analysis - 23rd International Symposium, SAS*, volume 9837 of *LNCS*, 2016.
- [RKBA⁺13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Conf. on Programming Language Design and Implementation*, 2013.
- [RKH⁺11] Gabe Rudy, Malik Murtaza Khan, Mary Hall, Chun Chen, and Jacqueline Chame. A programming language interface to describe transformations and code generation. In *Languages and Compilers for Parallel Computing*, 2011.

- [SLK⁺21] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: Automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conf. on Programming Language Design and Implementation*, pages 158–174, 2021.
- [SSH10] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. *PACT '10*, pages 513–522, 2010.
- [TV14] Ashkan Tousimojarad and Wim Vanderbauwhede. Comparison of three popular parallel programming models on the intel xeon phi. In *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II 20*, pages 314–325. Springer, 2014.
- [VD18] Wim Vanderbauwhede and Gavin Davidson. Domain-specific acceleration and auto-parallelization of legacy scientific code in fortran 77 using source-to-source compilation. *Computers & Fluids*, 173:1–5, 2018.
- [Wol95] M. Wolfe. *High performance compilers for parallel computing*. 1995.
- [YQ08] Qing Yi and Apan Qasem. Exploring the optimization space of dense linear algebra kernels. In *LCPC*, 2008.
- [YWC14] Qing Yi, Qian Wang, and Huimin Cui. Specializing compiler optimizations through programmable composition for dense matrix computations. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, page 596–608, USA, 2014.
- [ZCG18] Oleksandr Zinenko, Lorenzo Chelini, and Tobias Grosser. Declarative Transformations in the Polyhedral Model. Research Report RR-9243, December 2018.

Chamois: agile development of CompCert extensions for optimization and security

David Monniaux¹ and Sylvain Boulmé¹

¹Univ. Grenoble Alpes, CNRS, Grenoble INP

CompCert is the only formally-verified C compiler, and one of the very few formally verified compilers altogether. It is intended for use for safety-critical applications. This paper describes the improvements that we and associates have brought to CompCert: new VLIW target, new optimizations, and security features.

1 Introduction

Most compilers have no formal proof of correctness: their reliability is established by testing and by their track record for a certain kind of code on a certain platform. In industries such as avionics, it is required that features of the source code can be traced to the target code and the converse. One traditional approach then is to disable most optimizations, so that the source and assembly programs match closely, but this can cost a lot in performance. [Bed+11]

One solution is to use a *formally verified compiler*: a machine-checked proof of correctness states that if the compiler succeeds in compiling, then the semantics of the source and target codes match. There are few formally verified compilers: the only ones that we know of for general-purposes languages, are CompCert [Ler09b; Ler09a] for C and CakeML [Kum+14] for a dialect of ML. In this tool paper, we describe our extensions to CompCert.

CompCert's moderate optimization capabilities significantly improve upon disabling optimizations in a conventional compiler. However they are, as of 2023, still inferior to those of mainstream compilers such as gcc and LLVM. Furthermore, prior to 2023, CompCert was still missing support for desirable security features provided by mainstream compilers. The goal of Chamois¹ is to narrow the gap between CompCert and mainstream compilers.

2 Optimizations

We have improved CompCert in a variety of ways, which we shall discuss briefly.

2.1 Instruction selection

We added a full back-end for the Kalray K VX family of VLIW cores. We added support for certain instructions of certain processors, such as bitfield setting, clearing and extraction on ARM. In addition, we changed the operation sequences emitted for certain operations (such as signed division by two).

¹<https://gricad-gitlab.univ-grenoble-alpes.fr/certicompile/Chamois-CompCert>

2.2 Instruction scheduling

Many processors used in safety-critical or cheaper applications are *in-order*: they execute instructions in the order they occur in the program. If the operands of an instruction are not yet available when it executes, it and all instructions behind it in the pipeline *stall* until they become available. It is important that the compiler reorders instructions according to a *schedule* that minimizes stalling by taking into account instruction *latencies* (the clock cycles between when the instruction reads its operands and when it writes its results) as well as what combinations of instructions the processor can issue at the same clock cycle.

We added two scheduling passes. The first [Six21; Six+22] operates over the RTL intermediate representation, a form of “three-address code” that operates over an unbounded number of pseudo-registers, with memory accessed through “load” and “store” operations. It first divides the code into linear *superblocks*, each with one single entry point, a main exit and possibly some side exits. Superblocks are carved by heuristics that predict, at each branch point, the most common successor, or, often, conservatively opt not to identify one and terminate the superblock there. Furthermore, it is possible to use profiling information and user-provided annotations (`__builtin_expect()`, as in `gcc` and `LLVM`).

Each superblock is expressed in the BTL intermediate representation, a variant of RTL where control, instead of stepping through individual instructions, takes big steps across structured blocks, possibly containing conditionals, with one single entry point and possibly multiple exit points. Scheduling reorders instructions so that live outgoing values match between the original and reordered codes, and that no new trapping condition is introduced. For instance, the superblock `c=a/b; if (b>100) goto X;`, where `c` is not live at `X`, may be replaced by `if (b>100) goto X; c=a/b;`, but the converse replacement is valid only on machines where division by zero does not trap.²

Two simple *alias analyses* are used to allow swapping loads and stores over non-overlapping locations: one based on CompCert’s existing value analysis, and another dealing with non-overlapping offsets from the same base address. In addition, a mechanism (currently being improved) avoids introducing too many live pseudo-registers, which may result in spilling.

A second scheduling pass [Six21; SBM20] is run after register allocation (K VX and AArch64 only). This pass can thus take into account loads and stores that have been added because of values spilled to the stack frame, as well as loads and stores induced by register being saved during procedure calls. A *peephole optimizer* [Gou21] replaces accesses to consecutive memory locations by “load/store pair/quadruplets of registers” instructions. On VLIW processors (K VX) this pass also forms *instruction bundles*.

In both passes, following Tristan [Tri09] and Tristan and Leroy [TL08], an untrusted transformation is followed by a formally verified checker that symbolically executes, in a term algebra, both the original and transformed programs, and checks equivalence. It uses hash-consing, nontrivial to model in a functional context [BJM14; Bou21].

2.3 Code restructuring

We perform *loop peeling* (unrolling of the first iteration of a loop³ and *loop rotation*⁴ as particular cases of “morphisms” [Gou+23]: each node in the transformed program corresponds to a node in the original program, and instructions are to be preserved between corresponding nodes. We also provide an opposite transformation, which merges bisimilar nodes.

²On certain processors, such as x86, division by zero results in a system trap which, by default, results in the program being terminated by the operating system. We modified CompCert to account for division by zero not trapping on other architectures, so that division operations can be rescheduled over branches or moved out of loops.

On most processors, loading from an incorrect memory location results in a “segmentation violation” trap, again terminating the program. We added support to CompCert for the “non trapping” flag of load operations on the Kalray K VX processor.

³This amounts to replacing `while (c) {b}` by `if(c) {{b} while(c) {b}}`, but on unstructured code.

⁴Replacing `while(c) {b}` by `if(c) {do {b} while(c)}`

2.4 Global common subexpression elimination

Common subexpression elimination identifies that some expressions at different positions in the program are identical and thus that one can avoid recomputing their values and instead reuse previously computed values. CompCert featured a *local* common subexpression elimination, which would not propagate information across control-flow joins.

Our new pass [MS22] identifies when the result of evaluating an operation, or a load from memory, is known to always lie in a certain pseudo-register at a given control location. Then, if that control location computes the same operation, it can be replaced by a “move” from that pseudo-register. In the simplest case, any “store” to memory cancels all known relationships involving “loads”, but a simple alias analysis, dealing with accesses relative from the same base addressing, allows cancelling only those involving locations that cannot be proved not to overlap with the store. This pass was extended to remove redundant branches when their condition is known to always hold. Combined with loop peeling, this pass performs a form of *loop-invariant code motion*.

2.5 Lazy code motion and strength reduction

A second form a loop-invariant code motion moves loop-invariant expressions out of loops without need for unrolling (unless the expression may trap). This pass [Gou+23; Gou23] transforms BTL code, then runs a formally verified checker, which again performs symbolic execution, but initializes pseudo-registers according to invariants provided by the transformation pass. The checker also verifies these invariants hold inductively.

Lazy code motion was extended to *strength reduction*: replacing costly operations within loops by cheaper versions. Typically, an expression used within a loop contains a multiplication of an index by a constant (say, for computing the address of data in an array), with the index being incremented by a constant at every iteration, in which case it can be replaced by a computation of an initial value out of the loop and incrementing by a constant. That is, if i is initialized with n and then incremented at every loop iteration, $t + 4i$ can be computed by initializing it with $t + 4n$ and then incrementing it by 4 at every iteration. The formally verified checker was modified to rewrite strength-reduced expressions, so that $t + 4(i + 1)$ is considered equivalent to $(t + 4i) + 4$.

2.6 Tail recursion optimization

CompCert performs *tail call elimination*: tail calls to functions, which would normally retain the current stack frame, are replaced by the destruction of the current stack frame followed by a jump to the head of the function. If applied to a recursive function, this optimization creates a loop that destructs and recreates the stack frame at every iteration. We improve upon this by bypassing stack frame destruction and creation [Mon24].

3 Security features

We have added features that block certain software attacks and proved they do not disturb normal executions. In the future, we will also consider hardware attacks, and proving that countermeasures block certain attacks.

3.1 Branch target identification

On some platforms (x86, AArch64), Chamois can optionally add instructions that specify locations that are legal targets for a jump or function call. This reduces opportunities for “return address programming” and other techniques used to exploit software vulnerabilities.

3.2 Return address authentication

AArch64 provides *pointer authentication*. [App21; App19; Aza19; Tec17] Special instructions add some authentication bits to pointers, computed using keys that the operating system sets up in special CPU registers. Before the pointer is used, these bits are checked and removed; an invalid pointer is produced if they are incorrect. This prevents easy exploitation of vulnerabilities such as buffer overflows: the intruder cannot predict the keys and thus the value of the authentication bits to supply. If vulnerable software reads an intruder-supplied pointer from memory and de-authenticates it before using it to access memory, then the access traps.

Applying this approach to pointers in general needs some language-based mechanism for tagging which pointers are authenticated (with which key) and which are not; we do not have that mechanism, which would entail modifying CompCert’s front-end and most likely adding a new “tagged pointer” type. We currently just authenticate return addresses [Mon24].

3.3 Canaries

A well-known attack method is to overflow a buffer allocated on the stack to overwrite the return address of a function. A *canary*⁵ is a piece of data wedged between the local variables and the return address. Buffer overflows most often overwrite consecutive pieces of memory and thus overwrite the canary. Before restoring the return address and exiting the function, the value of the canary is checked, and the program aborts if it does not match. The canary value is randomized at program start and may not be predicted by the attack. We implemented this security feature, which had been standard for years in gcc and LLVM, and proved that it does not perturb legal executions [Mon24]. We discovered using these canaries that some benchmark programs that we were using exhibited undefined behavior (e.g. due to assuming a 32-bit platform).

4 Conclusion

Various optimizations and security features have been implemented, and more are on the way. Many of these optimizations are composed of an untrusted transformation followed by a formally verified checker. It is necessary to test, on representative and also “trick” examples, if it happens that the checker does not accept the results [Mon+23]. The main difficulty so far has been to identify what optimizations would be interesting and what causes performance discrepancies. The other difficulty is that some possible improvements would entail deep changes in the semantics of intermediate representations. It is often difficult to predict, from the outside, the degree of changes necessary.

Acknowledgements

Xavier Leroy kindly improved the abstract domain used in the value analysis. This results in more precise alias information, which helps several of our optimizations.

References

- [App19] Apple. *Pointer Authentication*. Documentation for Apple’s fork of LLVM. 2019. URL: <https://github.com/apple/llvm-project/blob/apple/main/clang/docs/PointerAuthentication.rst>.

⁵The name comes from canaries formerly used in mines to warn miners of carbon monoxide poisoning by dying before the miners.

- [App21] Apple. *ARMv8.3 Pointer Authentication in xnu*. 2021. URL: <https://opensource.apple.com/source/xnu/xnu-7195.50.7.100.1/doc/pac.md>.
- [Aza19] Brandon Azad. *Examining Pointer Authentication on the iPhone XS*. Google Project Zero. 2019. URL: <https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html>.
- [Bed+11] Ricardo Bedin França et al. “Towards Optimizing Certified Compilation in Flight Control Software”. In: *Workshop on Predictability and Performance in Embedded Systems (PPES 2011)*. Vol. 18. OpenAccess Series in Informatics. Grenoble, France: Dagstuhl Publishing, 2011, pp. 59–68. URL: <http://hal.archives-ouvertes.fr/inria-00551370/>.
- [BJM14] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. “Implementing and Reasoning About Hash-consed Data Structures in Coq”. In: *Journal of Automated Reasoning* (June 2014), pp. 1–34. ISSN: 0168-7433. DOI: 10.1007/s10817-014-9306-0. HAL: hal-00816672. URL: <https://hal.archives-ouvertes.fr/hal-00816672>.
- [Bou21] Sylvain Boulmé. “Formally Verified Defensive Programming (efficient Coq-verified computations from untrusted ML oracles)”. See also <http://www.verimag.imag.fr/boulme/hdr.html>. Habilitation à diriger des recherches. Université Grenoble-Alpes, Sept. 2021. URL: <https://hal.science/tel-03356701>.
- [Gou+23] Leo Gourdin et al. “Formally Verifying Optimizations with Block Simulations”. In: *OOPSLA*. To appear. Oct. 2023.
- [Gou21] Léo Gourdin. “formally verified postpass scheduling with peephole optimization for AArch64”. In: *AFADL*. 2021. URL: https://www.lirmm.fr/afadl2021/papers/afadl2021_paper_9.pdf.
- [Gou23] Léo Gourdin. “Lazy Code Transformations in a Formally Verified Compiler”. In: *Proceedings of the 18th ACM International Workshop on Implementation, Compilation, Optimization of OO Languages, Programs and Systems, ICPOOLPS 2023, Seattle, WA, USA, 17 July 2023*. Ed. by Eric Jul and Dimi Racordon. ACM, 2023, pp. 3–14. DOI: 10.1145/3605158.3605848.
- [Kum+14] Ramana Kumar et al. “CakeML: A Verified Implementation of ML”. In: *Principles of Programming Languages (POPL)*. ACM Press, Jan. 2014, pp. 179–191. DOI: 10.1145/2535838.2535841. URL: <https://cakeml.org/popl14.pdf>.
- [Ler09a] Xavier Leroy. “A Formally Verified Compiler Back-end”. In: *J. Autom. Reason.* 43.4 (2009), pp. 363–446. DOI: 10.1007/s10817-009-9155-4.
- [Ler09b] Xavier Leroy. “Formal verification of a realistic compiler”. In: *Commun. ACM* 52.7 (2009), pp. 107–115. DOI: 10.1145/1538788.1538814.
- [Mon+23] David Monniaux et al. “Testing a Formally Verified Compiler”. In: *Tests and Proofs - 17th International Conference, TAP 2023, Leicester, UK, July 18-19, 2023, Proceedings*. Ed. by Virgile Prevosto and Cristina Secleanu. Vol. 14066. Lecture Notes in Computer Science. Springer, 2023, pp. 40–48. DOI: 10.1007/978-3-031-38828-6_3.
- [Mon24] David Monniaux. “Memory Simulations, Security and Optimization in a Verified Compiler”. In: *Certified Programs and Proofs 2024*. Ed. by Brigitte Pientka and Sandrine Blazy. Brigitte Pientka and Sandrine Blazy and Amin Timany and Dmitriy Traytel. London, United Kingdom: Association for computing machinery (ACM), Jan. 2024. DOI: 10.1145/3636501.3636952. HAL: hal-04336347.
- [MS22] David Monniaux and Cyril Six. “Formally Verified Loop-Invariant Code Motion and Assorted Optimizations”. In: *ACM Trans. Embed. Comput. Syst.* 22.1 (Dec. 2022). ISSN: 1539-9087. DOI: 10.1145/3529507.

- [SBM20] Cyril Six, Sylvain Boulmé, and David Monniaux. “Certified and efficient instruction scheduling: application to interlocked VLIW processors”. In: *Proc. ACM Program. Lang.* 4.OOPSLA (2020), 129:1–129:29. DOI: 10.1145/3428197. HAL: hal-02185883.
- [Six+22] Cyril Six et al. “Formally verified superblock scheduling”. In: *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*. Ed. by Andrei Popescu and Steve Zdancewic. ACM, 2022, pp. 40–54. DOI: 10.1145/3497775.3503679. URL: <https://doi.org/10.1145/3497775.3503679>.
- [Six21] Cyril Six. “Compilation optimisante et formellement prouvée pour un processeur VLIW”. PhD thesis. Grenoble Alpes University, France, 2021. URL: <https://tel.archives-ouvertes.fr/tel-03326923>.
- [Tec17] Qualcomm Technologies. *Pointer authentication on ARMv8.3. Design and Analysis of the New Software Security Instructions*. 2017. URL: <https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf>.
- [TL08] Jean-Baptiste Tristan and Xavier Leroy. “Formal verification of translation validators: a case study on instruction scheduling optimizations”. In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2008, San Francisco, California, USA, January 7-12, 2008*. Ed. by George C. Necula and Philip Wadler. ACM, 2008, pp. 17–27. DOI: 10.1145/1328438.1328444. URL: <https://doi.org/10.1145/1328438.1328444>.
- [Tri09] Jean-Baptiste Tristan. “Formal verification of translation validators”. PhD thesis. Paris Diderot University, France, 2009. URL: <https://tel.archives-ouvertes.fr/tel-00437582>.

L'interprète, le JIT et la licorne

Frédéric Recoules et Sébastien Bardin

Université Paris Saclay, CEA, List
Saclay, France
frederic.recoules@cea.fr sebastien.bardin@cea.fr

L'exécution symbolique, une méthode populaire d'analyse de programmes, a du mal à passer à l'échelle sur de gros codes. Outre les traditionnels problèmes d'explosion de chemins et de résolution des prédicats logiques, d'aucuns se plaignent du coût d'interprétation du langage cible et se tournent par conséquent vers la compilation, qu'elle soit statique ou à la volée, pour améliorer les performances. Sceptiques mais non moins curieux, nous saisissons ce prétexte pour rajouter un peu de compilation à la volée dans le moteur d'exécution symbolique de la plateforme BINSEC, écrite en OCaml. Permettez-nous ainsi d'introduire JITPSI, une petite bibliothèque qui tire son inspiration de MetaOCaml et ocaml-jit. JITPSI concourt à compiler *harmonieusement* en x86-64, directement depuis OCaml, une séquence d'appels de fonctions. Nous utilisons JITPSI pour transformer à la volée l'interpréteur d'arbres syntaxiques abstraits de BINSEC en sous-programmes filetés (*threaded code*). Notre intention première est ici d'améliorer la résolution du défi de rétro-ingénierie licorne issu du *France CyberSecurity Challenge 2022* proposé par l'ANSSI ; soit une accélération d'environ 40%.

1 Introduction

L'exécution symbolique [CS13] est une analyse précise mais coûteuse. L'analyse évalue le programme avec des entrées symboliques et construit le long de chaque chemin un prédicat logique exprimant l'ensemble des valeurs d'entrée qui mènent à ce chemin. La résolution de ces prédicats se fait traditionnellement à l'aide de solveurs SMT (*Satisfiability Modulo Theory*), ce qui permet de vérifier si un point du programme est atteignable et d'obtenir un exemple de valeurs concrètes le cas échéant. Ce procédé prend toutefois un temps considérable. Pour ne rien arranger, l'opération est répétée un grand nombre de fois. Le nombre de chemins croît en effet exponentiellement avec le nombre de conditions rencontrées. Le raisonnement logique représente ainsi généralement la quasi-totalité du temps d'analyse, au point que toute considération de performance en dehors de l'amélioration de ces deux facteurs est d'ordinaire considérée comme de l'optimisation prématurée.

La combinaison avec le *fuzzing* [MZH20] est venue rebattre les cartes. Il s'est développé une forme particulière d'exécution symbolique [SGS⁺16] qui se concentre sur une seule trace d'exécution et vise à inverser un petit nombre de conditions, souvent rendues plus simples à l'aide de sous-approximations (concrétisations). Dans ce contexte où il n'y a plus d'explosion de chemins ni de raisonnements interminables, le coût d'interprétation du langage peut largement devenir prédominant. De nouvelles approches basées sur la spécialisation du moteur symbolique pour un programme donné ont ainsi vu le jour, que ce soit à base de compilation statique [PF20, WJG⁺23] ou d'instrumentation dynamique [YLX⁺18, PF21].

Problème. L'interprétation du langage peut devenir le goulot d'étranglement de l'analyse, même en dehors de la combinaison avec le *fuzzing*. Nous avons pu expérimenter cette « inversion des coûts » dans la plateforme d'analyse de code binaire BINSEC [DB15] lors du *France CyberSecurity Challenge 2022*. Tenter de résoudre un défi de rétro-ingénierie consiste à rechercher des valeurs d'entrée qui déclencheront « la » bonne sortie du programme. Bien que souvent mise à mal par des contre-mesures, l'exécution symbolique se prête plutôt bien à l'exercice. Ici, la subtilité du défi licorne [Ver22] réside dans son utilisation d'Unicorn [ND15] (machine virtuelle x86-64) pour émuler du code ARM caché. Or, si le prédicat de chemin de ce dernier est, *in fine*, trivial à résoudre pour un solveur SMT (une milliseconde), construire ce prédicat nécessite de propager symboliquement les entrées du programme le long d'une trace d'exécution d'environ un milliard d'instructions (majoritairement issues de la virtualisation). Dans sa version 0.6.0, BINSEC prenait jusqu'à trois heures pour en arriver à bout.

Objectif. Bien que BINSEC ait fait de gros progrès depuis (la version 0.8.0 est 15 fois plus rapide sur cette trace), nous souhaitons réduire encore davantage le coût d'interprétation de notre langage intermédiaire à l'aide de compilation à la volée. Nous nous intéressons particulièrement à l'évaluation des blocs de base, éléments simples qu'il nous est facile d'optimiser agressivement. Nous voulons spécialiser, au sens de la projection de Futamura [Fut99], la boucle d'évaluation récursive des instructions pour chacun des blocs rencontrés. En remplaçant le filtrage par motif (*pattern matching*) par des appels directs, nous espérons réduire le nombre d'indirections afin d'atteindre le même niveau de performance que si nous avions écrit ce bloc de base à la main sous la forme de sous-programmes filetés (*Subroutine Threaded Code* [Bel73, SYZZ⁺14]). Dans les cas favorables, cette forme permet également de transformer des variables temporaires du programme interprété en variables locales OCaml, économisant ainsi des opérations d'écriture et de lecture dans l'environnement symbolique.

Pour ce faire, nous avons besoin d'un moyen de dérouler la boucle d'évaluation récursive de BINSEC et d'assembler la séquence d'appels aux fonctions de manipulation de l'environnement symbolique sous forme d'une fonction OCaml native.

Défis. Malgré l'apparente simplicité de la tâche, les solutions existantes ne sont pas adaptées. Nous écartons d'emblée l'utilisation de *llvm* [LA04] en raison des spécificités du langage OCaml, notamment, de sa convention d'appel atypique et de sa gestion du glaneur de cellules (*garbage collector* [DL93]). *MetaOCaml* [Kis18] est un projet qui avait tout du candidat idéal, mais deux points négatifs viennent ternir le tableau :

1. *MetaOCaml* génère *littéralement* du code OCaml à compiler plus tard, ce qui rend le programme dépendant à l'exécution de la disponibilité des interfaces (*cmi*) – son utilisation dans un *foncteur*, dont le résultat n'est connu qu'au cours de l'exécution, est par ailleurs une difficulté que nous ne sommes humblement pas arrivés à surmonter ;
2. la génération du code machine est quelque peu inefficace, en particulier pour les petites fonctions – *MetaOCaml* passe en effet par les étapes classiques du compilateur OCaml, invoque le processus assembleur (*as*) puis charge le résultat en mémoire (*Dynlink*).

Le projet *ocaml-jit* résout quant à lui ce second inconvénient. Son objectif est d'accélérer l'interpréteur interactif de code OCaml (*utop*), ce qu'il fait en générant le code machine x86-64 directement en mémoire avant de le lier à l'application sans jamais avoir recours à un processus externe. Il nous manque cependant toujours une interface programmatique permettant d'assembler une nouvelle fonction à partir des valeurs accessibles durant l'exécution sans passer par du code OCaml textuel.

Proposition. Nous remédions à ce manque grâce à un petit langage dédié [FP11] que nous exposons dans une bibliothèque à travers les primitives *lambda*, *const*, *apply* et *return*.

En se basant sur les couches basses du compilateur OCaml, la bibliothèque JITPSI (*Just In Time Partial Specialization for Interpreter*) permet de générer du code x86-64 qui

s'intègre avec harmonie à son environnement, respectant sa convention d'appel et son modèle mémoire.

La conception de cette bibliothèque répond aux deux critères suivants :

- *La simplicité.* Que ce soit à l'utilisation, avec les garanties de la sûreté du typage, ou lors de son développement, en choisissant le niveau d'abstraction le plus adapté dans le compilateur OCaml ([Lambda](#)) ;
- *L'efficacité.* En s'inspirant de la génération de code d'ocaml-jit pour assembler la fonction et allouer sa fermeture (*closure*) sans quitter le monde d'OCaml.

Contributions. En résumé, cet article fait état des contributions suivantes :

- la conception de la bibliothèque JITPSI qui permet d'assembler programmatiquement le résultat de la composition de plusieurs appels de fonctions et de produire un code natif aussi performant, *si ce n'est plus*, que s'il avait été écrit en OCaml à la main ;
- la preuve de concept de son application dans le moteur d'exécution symbolique de BINSEC avec un effet positif et notable sur (*au moins*) l'exemple licorne, défi de rétro-ingénierie issu du *France CyberSecurity Challenge 2022*.

Les codes de JITPSI et de BINSEC sont accessibles sur GitHub, respectivement à <https://github.com/recoules/jitpsi> et <https://github.com/binsec/binsec/tree/jitpsi>.

Discussion. Nous abordons le thème de la compilation à la volée avec le prisme restreint de l'exécution symbolique. Cet article est avant tout l'occasion de partager le fruit de diverses expérimentations qui nous tenaient à cœur, bien plus que ce qu'elles ne se justifiaient. Pourtant, nous pensons que la compilation à la volée peut avoir de nombreux cas d'usages et nous espérons relancer l'intérêt de la communauté pour cette thématique mais aussi récolter de précieux retours sur l'utilité de JITPSI et les pistes d'améliorations pour répondre à de nouveaux besoins.

2 Exemple et motivations

Illustrons tout d'abord le rôle de JITPSI à travers la spécialisation d'un interpréteur jouet. La représentation intermédiaire de ce petit interpréteur de code x86-64 est donnée en Figure 1a. Un programme est représenté sous forme d'un graphe de micro-instructions permettant de manipuler un état mémoire composé de différents registres et d'un drapeau de condition. Le langage permet de calculer des expressions simples à base de registres et de constantes et de tester si une expression est égale à zéro. Le langage inclut également l'instruction `Builtin`. Elle permet d'appeler une fonction OCaml en charge de mettre à jour l'état du programme. Elle pourra servir à réaliser des tâches complexes ou difficilement exprimables dans ce langage, par exemple la division, ou comme nous allons le voir, à optimiser l'exécution d'une suite d'opérations connue. Le code de la Figure 1b permet d'évaluer le résultat concret d'un appel de fonction.

Pour la suite, nous allons nous intéresser à la fonction `fibonacci` dont le code assembleur est donné en Figure 3a, et dont la sémantique dans notre langage apparaît dans la Figure 3b.

V0. Afin d'exécuter cette fonction dans notre petit interpréteur, nous pouvons traduire littéralement cette sémantique en termes OCaml comme défini dans la Figure 2a. En moyenne, cette première mouture calcule le 92^{ème} élément de la suite de `fibonacci` en 66 mille cycles d'horloge (unité sans importance dans l'absolu).

V1 et V2. Commençons notre quête de performance. Au vu de la structure du code, la majeure partie du temps se déroule dans le corps de la boucle (instructions `i4` à `i9`). Ces instructions génèrent toujours la même séquence d'invocation des primitives de notre langage (`R.find`, `R.add`, etc.). En revanche, notre fonction d'évaluation se voit obligée de

```

module V = Stdint.Uint64
type register = Rax | Rdx | Rdi | Tmp0
type operator = Plus | Minus
type expression =
  | Cst of V.t
  | Reg of register
  | Op of operator * expression * expression
module R = Map.Make
(struct
  type t = register
  let compare = compare
end)
type instruction =
  | SetR of register * expression * instruction
  | SetZ of expression * instruction
  | IfZ of instruction * instruction
  | Ret of register
  | Builtin of
    (V.t R.t -> bool -> instruction -> V.t) *
    instruction

type code = Fun of instruction

let rec eval regs = function
  | Cst v -> v
  | Reg r -> R.find r regs
  | Op (Plus, x, y) ->
    V.add (eval regs x) (eval regs y)
  | Op (Minus, x, y) ->
    V.sub (eval regs x) (eval regs y)
let rec step regs z = function
  | SetR (r, e, i) ->
    step (R.add r (eval regs e) regs) z i
  | SetZ (e, i) ->
    step regs V.(compare (eval regs e) zero = 0) i
  | IfZ (i, i') ->
    if z then step regs z i else step regs z i'
  | Ret r -> R.find r regs
  | Builtin (f, i) -> f regs z i
let run rdi (Fun i) =
  step (R.singleton Rdi rdi) false i

```

(a) Représentation intermédiaire

(b) Fonctions d'évaluation

Figure 1. Interpréteur simplifié de code x86-64 écrit en OCaml

```

let rec fibonacci = Fun i0
and i0 = SetR (Rax, Cst V.zero, i1)
and i1 = SetZ (Reg Rdi, i2)
and i2 = IfZ (i6, i3)
and i3 = SetR (Rdx, Cst V.one, i4)
and i4 = SetR (Tmp0, Reg Rdx, i5)
and i5 = SetR (Rdx, Reg Rax, i6)
and i6 = SetR (Rax, Reg Tmp0, i7)
and o1 = Op (Plus, Reg Rax, Reg Rdx)
and i7 = SetR (Rax, o1, i8)
and o2 = Op (Minus, Reg Rdi, Cst V.one)
and i8 = SetR (Rdi, o2, i9)
and i9 = SetZ (Reg Rdi, ia)
and ia = IfZ (ib, i4)
and ib = Ret Rax

```

(a) Définition V0

```

let rec fibonacci = Fun i0
and i0 = SetR (Rax, Cst V.zero, i1)
and i1 = SetZ (Reg Rdi, i2)
and i2 = IfZ (i6, i3)
and i3 = SetR (Rdx, Cst V.one, i4)
and loop r _ i =
  let v0 = V.zero in
  let v1 = V.one in
  let v2 = R.find Rdx r in
  let v3 = R.find Rax r in
  let v4 = R.find Rdi r in
  let v5 = V.add v2 v3 in
  let v6 = V.sub v4 v1 in
  let r0 = R.add Rdx v3 r in
  let r1 = R.add Rax v5 r0 in
  let r2 = R.add Rdi v6 r1 in
  let z0 = V.compare v6 v0 = 0 in
  step r2 z0 i
and i4 = Builtin (loop, i5)
and i5 = IfZ (i6, i4)
and i6 = Ret Rax

```

(c) Définition V2

```

let rec fibonacci = Fun i0
and i0 = SetR (Rax, Cst V.zero, i1)
and i1 = SetZ (Reg Rdi, i2)
and i2 = IfZ (i6, i3)
and i3 = SetR (Rdx, Cst V.one, i4)
and loop r _ i =
  let v0 = R.find Rdx r in
  let r0 = R.add Tmp0 v0 r in
  let v1 = R.find Rax r0 in
  let r1 = R.add Rdx v1 r0 in
  let v2 = R.find Tmp0 r1 in
  let r2 = R.add Rax v2 r1 in
  let v3 = R.find Rax r2 in
  let v4 = R.find Rdx r2 in
  let v5 = V.add v3 v4 in
  let r3 = R.add Rax v5 r2 in
  let v6 = R.find Rdi r3 in
  let v7 = V.one in
  let v8 = V.sub v6 v7 in
  let r4 = R.add Rdi v8 r3 in
  let v9 = R.find Rdi r4 in
  let va = V.zero in
  let z0 = V.compare v9 va = 0 in
  step r4 z0 i
and i4 = Builtin (loop, i5)
and i5 = IfZ (i6, i4)
and i6 = Ret Rax

```

(b) Définition V1

```

let rec fibonacci = Fun i0
and i0 = SetR (Rax, Cst V.zero, i1)
and i1 = SetZ (Reg Rdi, i2)
and i2 = IfZ (i6, i3)
and i3 = SetR (Rdx, Cst V.one, i4)
and loop = compile V0.[ i4; i5; i6; i7; i8; i9 ]
and i4 = Builtin (loop, i5)
and i5 = IfZ (i6, i4)
and i6 = Ret Rax

```

(d) Définition V3

Figure 2. Représentation intermédiaire de la fonction fibonacci en OCaml

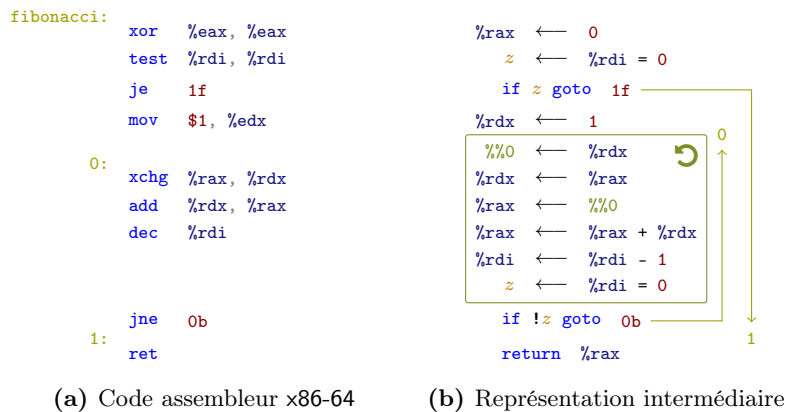


Figure 3. Calcul de la suite de Fibonacci

filtrer les différents motifs pour savoir quelle primitive appeler. Or, il est bien connu que les indirections sont les grandes ennemies des interpréteurs [RLV⁺96]. La seconde version présentée dans la Figure 2b supprime le filtrage de motifs et déroule manuellement le corps de la boucle. Pour aller plus loin, nous pouvons également observer que certaines invocations de primitives se révèlent superflues au regard du bloc dans son ensemble. Aussi, il est par exemple inutile de créer la valeur `v3` en allant lire la variable `%rax` car nous venons d'y écrire la valeur `v2` (lecture sur écriture). Or, maintenant que nous n'allons plus lire l'environnement pour récupérer `v3`, l'écriture de `v2` dans `%rax` devient inutile. En effet, une nouvelle écriture dans `%rax` a lieu plus tard et seule cette dernière sera retenue une fois le bloc exécuté (écriture sur écriture). En déroulant le raisonnement, nous aboutissons à la version de la Figure 2c. Cette version est plus compacte, utilise moins de primitives et a complètement fait disparaître la variable temporaire `%%0`. Tout cela est rendu possible grâce à une capacité que notre boucle d'évaluation n'a pas : nous pouvons garder autant de résultats intermédiaires que nous le voulons sans les écrire dans l'environnement du programme. Les performances sont également au rendez-vous. En moyenne, cette version produit son résultat en 39 mille cycles, soit une accélération d'environ 70%.

V3. *Au sein de cet interpréteur*, il nous paraît difficile de faire beaucoup mieux que la version précédente. En revanche, il reste un point sur lequel nous pouvons encore gagner : l'automatisation. En effet, il n'est pas envisageable d'écrire et d'optimiser la représentation intermédiaire de notre langage à la main. C'est donc ici que JITPSI rentre en jeu.

Sur le modèle de nos fonctions d'évaluation, JITPSI nous permet d'écrire un générateur de fonctions d'évaluation. Le code de la Figure 4 dépeint l'usage de JITPSI couplé, comme nous l'avons fait à la main, à de la propagation d'expressions. Notre fonction `compile` prend en argument une liste d'instructions `il` et renvoie la fonction « `Builtin` » équivalente à l'évaluation récursive de ces instructions par la fonction `step` (Figure 1b).

Dans les grandes lignes, nous allons suivre les mêmes étapes que le code de la version **V2** (Figure 2c). La primitive `lambda3` (ligne 11) permet de créer une nouvelle fonction d'arité 3. Cette primitive prend en argument une fonction OCaml en charge de définir notre valeur de retour à partir de ces trois arguments. Nous commençons par récupérer la valeur des registres à l'aide de `R.find`, ce qui se traduit par un appel à la primitive `apply2` (ligne 13). Les deux premiers arguments, `R.find` et `r` sont des constantes, au sens de JITPSI, et sont récupérés depuis l'environnement OCaml à l'aide de la primitive `const`. Nous calculons ensuite les valeurs intermédiaires en déroulant la liste d'instructions `il` (ligne 14-19). La fonction `fold` incorpore ainsi les appels aux fonctions `V.add` et `V.sub` à la ligne 9. L'état mémoire est finalement mis à jour avec la nouvelle valeur des registres (`R.add`) et du drapeau `z` si nécessaire avant de terminer par l'appel final (`tailcall`) à la fonction `step` (ligne 23).

```

1  type 'a param = ('a, value) t
2  let is_zero v = V.compare v V.zero = 0
3  let j_add = const V.add and j_sub = const V.sub and j_is_zero = const is_zero
4  and j_assign = const R.add and j_lookup = const R.find
5
6  let rec fold e env = match e with
7  | Cst v -> const v
8  | Reg r -> R.find r env
9  | Op (op, x, y) -> apply2 (match op with Plus -> j_add | Minus -> j_sub) (fold x env) (fold y env)
10 let compile (il : instruction list) : V.t R.t -> bool -> instruction -> V.t =
11 return (lambda3 (fun (regs : V.t R.t param) (z : bool param) (i : instruction param) ->
12   let env = List.fold_left
13     (fun env r -> R.add r (apply2 j_lookup (const r) regs) env) R.empty [ Rax; Rdx; Rdi ] in
14     let env, zf = List.fold_left
15       (fun (env, z) -> function
16         | SetR (r, e, _) -> (R.add r (fold e env) env, z)
17         | SetZ (e, _) -> (env, Some (fold e env))
18         | IfZ _ | Ret _ | Builtin _ -> failwith "unexpected instruction kind")
19       (env, None) il in
20     let env = R.remove Tmp0 env in
21     let regs = R.fold (fun r v regs -> apply3 j_assign (const r) v regs) env regs in
22     let z = match zf with None -> z | Some v -> apply j_is_zero v in
23     apply3 (const step) regs z i))

```

Figure 4. Compilation d'un bloc d'instructions

La dernière version présentée dans la Figure 2d recourt à cette fonction `compile` pour optimiser automatiquement le morceau de code original (V0). Conformément à nos attentes, cette nouvelle version s'exécute, tout comme la précédente, en moyenne en 39 mille cycles.

Avant d'en finir avec la fonction `fibonacci`, il nous faut encore aborder un petit désagrément. Comme tout procédé de compilation à la volée, notre fonction `compile` a un coût et il ne faut pas le négliger. Ainsi, pour produire automatiquement notre version V3, il aura fallu en moyenne 1 million 770 mille cycles. Ce temps de *préparation* est bien plus important que le temps nécessaire à faire tourner la version originale V0. Aussi, rapportée à un tour de boucle, la compilation ne devient ici intéressante qu'à partir de la 5950^{ème} itération. La légèreté de JITPSI permet néanmoins de limiter cet impact négatif comparé à des approches utilisant un assembleur externe (MetaOCaml peut être 20 à 50 fois plus lent, voir Section 5).

Maintenant que nous en avons fini avec la fonction `fibonacci`, intéressons-nous un petit peu aux détails de la bibliothèque JITPSI.

3 Conception d'un générateur de code léger en OCaml

La bibliothèque JITPSI se veut simple et efficace en reposant sagement sur les couches basses du compilateur OCaml. En interne, JITPSI manipule les valeurs OCaml pour ce qu'elles sont à l'exécution, des entiers, des pointeurs et des fermetures. Le code de JITPSI s'organise en deux parties principales : l'interface utilisateur et l'émetteur de code. La Figure 5 montre comment JITPSI s'intègre dans le schéma du compilateur OCaml.

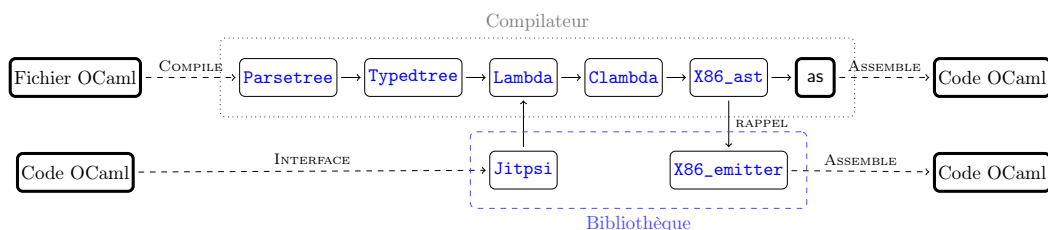


Figure 5. Organisation de JITPSI

```

type value and lambda and (+'a, 'b) t
val const : 'a -> ('a, value) t
val apply : ('a -> 'b, value) t -> ('a, value) t -> ('b, value) t
val lambda : (('a, value) t -> ('b, 'c) t) -> ('a -> 'b, lambda) t
val return : ('a, lambda) t -> 'a

```

Figure 6. Interface typée de JITPSI

L'interface. L'interface de JITPSI est donnée en Figure 6. Elle propose des primitives qui garantissent la sûreté du typage à l'aide de types fantômes. Le typage se fait ainsi à la compilation initiale du programme, économisant une partie du travail à l'exécution tout en prévenant les erreurs d'inattention.

En interne, l'interface est en charge de construire des termes `Lambda` bien formés. Le type `t` représente une expression, soit d'une valeur (`value`), soit d'une fonction (`lambda`). La primitive `const` a le rôle de la fonction identité. La primitive `apply` permet d'appeler une fonction avec un argument. La création d'une nouvelle fonction passe par la primitive `lambda`. Elle prend en argument une fonction OCaml en charge de définir la valeur de retour à partir de son argument. Cette valeur de retour peut prendre la forme d'une autre expression `lambda`, ce qui permettra de construire une fonction de plus grande arité (par exemple `lambda2 body` est équivalent à `lambda (fun a -> lambda (body a))`). Finalement, une expression `lambda` pourra être assemblée en mémoire à l'aide de la primitive `return`.

Afin de faire le lien avec l'environnement OCaml, JITPSI initialise les expressions constantes à partir d'éléments fictifs (*placeholder*) et ce depuis l'extérieur de l'environnement lexical de l'expression `lambda`. Ce procédé aura pour effet de créer des entrées dans sa fermeture.

Émission de code. Comme illustré en Figure 5, le module `X86_emitter` reçoit de la part du compilateur OCaml (*callback*) un programme assembleur de type `X86_ast`. JITPSI interprète ce programme dont la structure est décomposée en trois segments :

1. *Le corps de la fonction.* Les mnémoniques sont directement traduites en opcodes sur le même modèle qu'`ocaml-jit`. Notons qu'ici, la simplicité des constructions utilisées limite naturellement la variété des instructions à quelques mnémoniques (`call`, `mov`, `lea`, `add`, `sub`, `cmp`, `jmp`, `cc`, `jmp` et `ret`) ;
2. *L'allocation de la fermeture.* Les mnémoniques sont ici évaluées sommairement plutôt que traduites. JITPSI suit ainsi simplement la recette concoctée par le compilateur pour initialiser la fermeture. Durant cette phase, JITPSI simule les accès aux éléments fictifs et les remplace par les valeurs OCaml issues de l'environnement hôte ;
3. *Les métadonnées.* Les informations nécessaires au glaneur de cellules (`frametable`) sont extraites afin d'être déclarées auprès de l'environnement d'exécution d'OCaml.

Cette méthode permet de renvoyer une fermeture complète qui s'intègre parfaitement au monde OCaml et s'utilise comme le serait n'importe quelle fonction nativement compilée.

Les petits plus. Le code généré par JITPSI s'accompagne des deux avantages suivants :

- une fonction qui n'est plus utilisée peut être collectée par le glaneur de cellule et la mémoire allouée pour son code et ses métadonnées libérées ;
- de nature dynamique, JITPSI peut déterminer l'adresse réelle et le nombre d'arguments des fonctions qu'il manipule, connaissance qui offre plus de latitude au compilateur pour générer des appels directs en lieu et place des primitives de type `caml_apply`.

Limites actuelles et extensions envisagées. Sous sa forme actuelle, JITPSI souffre des limitations suivantes.

Puissance du langage. JITPSI permet pour le moment de composer des appels de fonction OCaml présentes dans l'environnement, mais ne propose pas d'autres opérations de manière

native. Ainsi, par exemple, les branchements conditionnels ne sont pas disponibles en toute généralité. On peut cependant gérer des cas simples en ajoutant une fonction de choix *if-then-else* (`val ite : bool -> 'a -> 'a -> 'a`), mais, dans ce cas, les arguments seront tous les deux évalués. L'extension du moteur pour une gestion plus générale du flot de contrôle est une des priorités d'extension ;

Autres architectures. L'émission de code est un processus qui dépend fortement de l'architecture cible, ici le `x86-64`. Supporter une nouvelle architecture demande de traduire de nouvelles mnémoniques en opcodes. À titre indicatif, pour `x86-64`, nous devons traduire 8 grandes classes de mnémoniques et la traduction prend ~ 600 lignes d'OCaml. Cependant, d'un point de vue pratique, la tâche serait rendue plus difficile pour une autre architecture car le compilateur OCaml n'a pas d'équivalent du module `X86_ast` pour les autres architectures et émet le code assembleur directement sous forme textuelle.

4 Application à l'exécution symbolique de BINSEC

Le moteur d'exécution de la plateforme BINSEC a sensiblement évolué depuis deux ans. Des progrès ont notamment été faits dans sa manière d'interpréter son langage intermédiaire [BHL⁺11]. La version actuelle (0.8.0) ne fait toutefois pas de Partage de Sous-expressions Communes (*local value numbering* [CT04]), que nous abrègerons en PSC. Or, avec l'aide de JITPSI, faire du PSC devient aussi facile que de faire du partage maximale d'expression (*hash consing*). En effet, JITPSI permet de réordonner toutes les opérations de lecture avant les opérations d'écriture. Ce scénario simplifie grandement la propagation d'expressions (comme illustré en Figure 4) qui n'a plus à se soucier des effets de bord des langages impératifs. Dans le but d'obtenir une base de comparaison et, bien que moins évident à mettre en place, nous avons également rajouté un PSC « classique » dans BINSEC.

Nous avons mesuré à l'aide de `Landmark` le temps de résolution du défi licorne avec trois configurations : une version classique de BINSEC, une version avec un PSC classique et une version avec JITPSI (incluant du PSC maximal). La Table 1 résume les résultats obtenus et montre le nombre et le coût des différents appels aux primitives de l'environnement symbolique. Nous pouvons y voir que la combinaison du PSC et de JITPSI permet d'accélérer de 43% l'exécution de ce programme par rapport à la version classique de BINSEC. Ce gain est en partie imputable au principe du PSC mais ne s'y limite pas. La version utilisant JITPSI est en effet 20% plus rapide que notre implémentation standard de PSC. Nous expliquons cela par le fait qu'il y a, d'une part, moins de variables temporaires (réduction du nombre d'écritures) et d'autre part, que JITPSI permet de factoriser les lectures de variables (réduction drastique du nombre de lectures), ce qui n'est pas possible par voie classique.

Table 1. Comparaison de différentes versions de BINSEC sur licorne

	Occurrences			# Cycles d'horloge		
	Classique	PSC	JITPSI	Classique	PSC	JITPSI
Écriture d'une variable	2 074 M	1 006 M	812 M	427 G	264 G	164 G
Lecture d'une variable	2 964 M	1 825 M	457 M	404 G	294 G	63 G
Écriture en mémoire	139 M	139 M	139 M	352 G	373 G	348 G
Lecture en mémoire	208 M	207 M	207 M	170 G	184 G	164 G
	Résolution complète			10m24s	8m42s	7m15s

PSC : partage des sous-expr. communes – la version JITPSI intègre le PSC et le JIT

5 Positionnement et discussion

JITPSI répond à un besoin précis. Il apporte une solution simple et efficace, mais qui, par la sobriété de son langage, limite à l'heure actuelle le champ des possibles. Nous envisageons

toutefois d'enrichir ce langage avec les expressions *if-then-else* et quelques opérations sur les types de bases (`bool`, `int`, etc.).

À notre connaissance, *JITPSI* est la seule approche à proposer une interface programmatique permettant de manipuler les valeurs du programme en toute autonomie à l'exécution.













D'autres travaux connexes traitent de la thématique de la compilation à la volée en OCaml. Le projet OCamlJIT2 [Meu11] vise à accélérer l'exécution d'OCaml en version *bytecode*. Il ne semble toutefois plus maintenu depuis plus d'une dizaine d'années.




MetaOCaml [Kis18] est une grande source d'inspiration pour l'interface de JITPSI. En théorie, MetaOCaml permet de faire tout ce pour quoi JITPSI a été conçu, et bien plus encore. En revanche, dans la pratique, nous avons eu le plus grand mal à expérimenter avec. Les problèmes rencontrés sont des erreurs qui ont lieu à l'exécution à cause d'une interface introuvable ou d'une valeur difficile à sérialiser (*cross-stage persistence*). Notons que MetaOCaml est naturellement multi-architecture puisqu'il génère du code OCaml. Pour ce qui est du temps d'émission de code, nous avons mesuré combien de cycles d'horloge étaient nécessaires à MetaOCaml et à JITPSI pour compiler la fonction `spower` présentée dans son exemple d'introduction. Il se trouve que JITPSI est bien plus rapide, avec une accélération allant de 20 (grandes valeurs de `n`) à 50 fois pour `n = 2`.

Le projet `ocaml-jit` est la source principale d'inspiration de l'émetteur de code de JITPSI. Sa cible, l'interpréteur interactif d'OCaml, est cependant assez éloigné de nos besoins.

La Table 2 résume les différences entre ces approches sur quelques aspects clés.

Table 2. Comparaison des approches

	 MetaOCaml	 ocaml-jit	 JITPSI
Autonomie à l'exécution			
Temps d'émission de code			
Multi-architecture			

 : relativement lent –  : très rapide –  : difficilement extensible

Applications potentielles. Le cadre d'usage classique de JITPSI est l'optimisation d'interpréteurs au sens large, et notamment la spécialisation de l'interprétation pour des séquences d'instructions données (et souvent ré-appelées). Des exemples possibles d'application incluent l'exécution symbolique, les simulateurs, émulateurs et autres machines virtuelles pour des gains en performance (vitesse), ou encore les interpréteurs abstraits, pour lesquels définir des transformateurs abstraits spécialisés pour les blocs du programme à analyser pourrait permettre des gains notables de précision. Nous sommes preneurs de retour pour tout autre domaine d'application intéressant.

6 Conclusion

Nous avons présenté dans cet article JITPSI, une bibliothèque de compilation à la volée pour OCaml. JITPSI expose une interface programmatique simple permettant la composition de plusieurs appels de fonctions dans le but de produire un code natif aussi performant que s'il avait été produit par compilation depuis des sources OCaml. Un domaine typique d'utilisation visé est l'optimisation d'interpréteurs (au sens large) via encodage dédié par blocs d'instructions plutôt que par instructions. Nous utilisons par exemple JITPSI pour optimiser l'exécuteur symbolique BINSEC sur de longs chemins d'exécution, permettant d'accélérer la résolution du défi de rétro-ingénierie licorne (tiré du *France CyberSecurity Challenge 2022* de l'ANSSI) d'environ 40%. Nous espérons ainsi raviver l'intérêt de la communauté OCaml pour la compilation à la volée et trouver de nouveaux cas d'usages et pistes d'améliorations.

Références

- [Bel73] James R. BELL : Threaded code. *Commun. ACM*, 1973.
- [BHL⁺11] Sébastien BARDIN, Philippe HERRMANN, Jérôme LEROUX, Olivier LY, Renaud TABARY et Aymeric VINCENT : The BINCOA framework for binary code analysis. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, 2011.
- [CS13] Cristian CADAR et Koushik SEN : Symbolic execution for software testing : Three decades later. *Commun. ACM*, 2013.
- [CT04] Keith D. COOPER et Linda TORCZON : *Engineering a Compiler*. Morgan Kaufmann, 2004.
- [DB15] Adel DJOUDI et Sébastien BARDIN : BINSEC : binary code analysis with low-level regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS, 2015*.
- [DL93] Damien DOLIGEZ et Xavier LEROY : A concurrent, generational garbage collector for a multithreaded implementation of ml. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1993.
- [FP11] Martin FOWLER et Rebecca PARSONS : *Domain-specific languages*. Addison-Wesley, 2011.
- [Fut99] Yoshihiko FUTAMURA : Partial evaluation of computation process—an approach to a compiler-compiler. *Higher Order Symbol. Comput.*, 1999.
- [Kis18] Oleg KISELYOV : Reconciling abstraction with high performance : A metaocaml approach. *Foundations and Trends in Programming Languages*, 2018.
- [LA04] Chris LATTNER et Vikram ADVE : Llmv : A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization : Feedback-Directed and Runtime Optimization*, 2004.
- [Meu11] Benedikt MEURER : Ocamljit 2.0 - faster objective caml, 2011.
- [MZH20] Barton P. MILLER, Mengxiao ZHANG et Elisa R. HEYMANN : The relevance of classic fuzz testing : Have we solved this one? *IEEE Transactions on Software Engineering*, 2020.
- [ND15] Anh Q. NGUYEN et Hoang V. DANG : Unicorn : Next generation cpu emulator framework. BlackHat USA, 2015. <https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf>.
- [PF20] Sebastian POEPLAU et Aurélien FRANCILLON : Symbolic execution with symcc : Don't interpret, compile! In *29th USENIX Security Symposium, USENIX Security*, 2020.
- [PF21] Sebastian POEPLAU et Aurélien FRANCILLON : Symqemu : Compilation-based symbolic execution for binaries. In *28th Annual Network and Distributed System Security Symposium, NDSS*, 2021.
- [RLV⁺96] Theodore H. ROMER, Dennis LEE, Geoffrey M. VOELKER, Alec WOLMAN, Wayne A. WONG, Jean-Loup BAER, Brian N. BERSHAD et Henry M. LEVY : The structure and performance of interpreters. *SIGOPS Oper. Syst. Rev.*, 1996.
- [SGS⁺16] Nick STEPHENS, John GROSEN, Christopher SALLS, Andrew DUTCHER, Ruoyu WANG, Jacopo CORBETTA, Yan SHOSHITAISHVILI, Christopher KRÜGEL et Giovanni VIGNA : Driller : Augmenting fuzzing through selective symbolic execution. In *Network and Distributed System Security Symposium*, 2016.
- [SYZZ⁺14] Gulfem SAVRUN YENICERI, Wei ZHANG, Huahan ZHANG, Eric SECKLER, Chen LI, Stefan BRUNTHALER, Per LARSEN et Michael FRANZ : Efficient hosted interpreters on the jvm. *ACM Trans. Archit. Code Optim.*, 2014.

- [Ver22] Mathéo VERGNOLLE : FCSC 2022 : Licorne. France Cyber-Security Challenge, 2022. https://github.com/binsec/binsec/blob/34e2897f2d813ae203b01a7273edafd747547813/doc/sse/fcsc_licorne.md.
- [WJG⁺23] Guannan WEI, Songlin JIA, Ruiqi GAO, Haotian DENG, Shangyin TAN et Oliver BRAČEVAC : Compiling parallel symbolic execution with continuations. In *45th International Conference on Software Engineering, ICSE*, 2023.
- [YLX⁺18] Insu YUN, Sangho LEE, Meng XU, Yeongjin JANG et Taesoo KIM : Qsym : A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium*, 2018.

SÉMANTIQUE

Resource Categories from Differential Categories

Lison Blondeau-Patissier¹

¹Aix Marseille Univ, CNRS, I2M and LIS, Marseille, France

Resource categories were recently introduced to capture the categorical structure of pointer concurrent games, and in particular to characterize morphisms behaving “linearly”. These linear morphisms correspond to (beta-normal, eta-expanded) terms of the resource lambda-calculus. Resource calculus is closely related to Ehrhard and Regnier’s differential lambda-calculus, which is usually interpreted in differential categories (defined by Blute, Cockett, Lemay and Seely). However, strategies of pointer concurrent games are not built from a model of linear logic, so their categorical structure is not a differential category.

Nevertheless, resource categories can be constructed from differential categories. We present such a construction in this paper, starting from an additive monoidal category and building a resource category from it.

1 Introduction

Resource categories were recently introduced in [BCVA23] as a way to capture the categorical structure of *pointer concurrent games*, a game semantics model also first presented in that paper. The aim was to obtain a categorical framework enabling the characterization of morphisms behaving “linearly”, to show that these morphisms in pointer concurrent games are in bijection with terms of the *resource lambda-calculus*.

Both game semantics and resource calculus have been well-studied lines of work for years, and both of them consider (multisets of) finite executions of programs to represent programs with possibly infinite behavior.

Game semantics model programs as processes, focusing on the interactions between the program and its environment. These interactions are represented as a *game* between two protagonists, one of them called Player representing the program and the other called Opponent representing the context. Information tokens exchanged between them are seen as moves, in a game whose rules depend on the type of the program. A *play* represents one possible execution of a program; programs themselves are represented by *strategies*, which are sets of plays (corresponding to every possible execution of the program). Game semantics is known for its many full-abstraction result, particularly in [AJM13] and [HO00] which introduced two standard game models, respectively called AJM games and HO games from the name of the authors. A key notion in HO games is the one of *innocence* of a strategy, corresponding to *pure* programs, or to lambda-terms without references. Such programs will react in the same way no matter how many times Opponent duplicates a request to evaluate a subprogram, or the order in which they chose to evaluate subprograms. This led to Mellies’ homotopy equivalence on plays introduced in [Mel06], quotienting plays by Opponent’s scheduling. Those quotiented plays can alternatively be seen as *augmentations*, another representation of plays introduced in [BC21] and inspired by concurrent games

(see [CCRW17] for an introduction to concurrent games). Later [BCVA23] considered composition of augmentations, defining strategies as sums of augmentations, forming the category of *pointer concurrent games*.

Resource calculus, on the other hand, arose from linear logic (introduced in [Gir87]) and quantitative semantics ([Gir88]). Unlike usual λ -calculus, where the substitution $M[N/x]$ can duplicate the term N as many times as the variable x occurs freely in the term M (or even more, if x itself is duplicated at some point of the execution), resource λ -calculus sees terms as *resources* which can each be used exactly once. Hence, the substitution is not defined with a term N anymore, but rather with a multiset of terms $[N_1, \dots, N_n]$, which will each replace exactly one occurrence of x in M (the exact bijection being chosen non-deterministically, *via* a sum of resource terms corresponding to all possible substitutions). This allows for a control of the number of copies of N , and for example ensures that the reduction terminates. Replacing arguments with multisets of terms in λ -calculi first emerged with the λ -calculus with multiplicities [Bou93], the term *resource* appearing a few years later in [BCL99]. Resource calculus is closely related to *differential λ -calculus*, developed in [ER03], which is a λ -calculus equipped with a formal differentiation. Intuitively, the derivative of a term $M: A \rightarrow B$ is the “linear” part of M , that is a function which uses its argument *exactly once* – this corresponds to the notion of differentiation in analysis, in which the derivative of a function is a linear approximation. Resource calculus is the finitary fragment of differential λ -calculus, not containing *pure* λ -terms (which are terms that can access their arguments “as needed”, possibly infinitely many times – represented in linear logic by the type $!A \multimap B$).

The correspondence between those two models was studied for instance by Tsukada and Ong in [TO16], where they show that normal, η -long resource terms are in bijection with Hyland-Ong plays up to Melliès’ homotopy equivalence. This correspondence was further developed in [BCVA23], establishing a denotational interpretation, invariant under reduction, of resource terms as strategies for pointer concurrent games. To prove this result, the categorical structure of pointer concurrent games was exposed in a way which enables the characterization of strategies interpreting resource terms. This led to the definition of *resource categories*, categories in which some morphisms can be identified as “linear”: either “using their resources” exactly once, or being able to be “used as a resource” exactly once.

Usually, models of linear logic are interpreted using differential categories, introduced in [BCS06] as a categorical framework for differential linear logic. But in [BCVA23] resource-calculus was studied in relation with games, and strategies are *not* linear: they do not have a linear behavior in general – the identity for pointer concurrent games is not even finite. Hence resource categories are *not* a category of resource terms¹ – in fact, this interpretation of resource terms lacks an identity. Nonetheless, resource categories are built using similar constructions to some differential categories, more precisely *monoidal storage categories* as described in [BCLS20]. The intuition behind these similarities is that the exponential $!$ of differential categories allows us to go from linear morphisms from A to B , to morphisms from $!A$ to $!B$, which behave linearly with respect to $!A$ and $!B$, but not with respect to the original objects A and B . These intuitions will guide us in our construction of resource categories from additive monoidal storage categories – which are the categories we mostly refer to when mentioning “differential categories” in this paper, although differential categories in general are a much wider notion.

Outline. We start by giving some general categorical definitions in Section 2, before introducing more precisely resource categories in Section 3. We then focus on differential categories in Section 4, defining additive monoidal storage categories – our main focus is not to give an exhaustive presentation of differential categories in this paper, but rather to present the particular category which we will use to build a resource category. We detail this construction in Section 5, stating in Theorem 1 that we obtain a resource category.

¹Nevertheless we call them resource categories because they contain morphisms acting like resource terms.

2 Categorical Preliminaries

This section presents some categorical notions which are used throughout this paper: symmetric monoidal categories, string diagrams, and (co)monoids. We direct the interested reader to [ML71] for an introduction to category theory.

Symmetric Monoidal Categories. A *monoidal category* is a category equipped with a *tensor product*; if it comes with a notion of commutativity of this tensor, the monoidal category is additionally *symmetric*.

Definition 1 (Monoidal Category). A *monoidal category* is a category \mathcal{C} equipped with:

- a functor $\otimes: \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ called the *tensor*;
- an object $I \in \mathcal{C}$ called the *unit*;
- the following isomorphisms natural in A, B, C :

$$\begin{aligned} \alpha_{A,B,C}: (A \otimes B) \otimes C &\rightarrow A \otimes (B \otimes C) && (\text{associator}) \\ \lambda_A: I \otimes A &\rightarrow A && (\text{left-unitor}) \\ \rho_A: A \otimes I &\rightarrow A && (\text{right-unitor}) \end{aligned}$$

such that for any objects A, B, C, D ,

$$(\text{id}_A \otimes \lambda_B) \circ \alpha_{A,I,B} = \rho_A \otimes \text{id}_B \quad (\text{triangle identity})$$

and the diagram of Figure 1 commutes.

$$\begin{array}{ccc} ((A \otimes B) \otimes C) \otimes D & \xrightarrow{\alpha_{A \otimes B, C, D}} & (A \otimes B) \otimes (C \otimes D) \\ \alpha_{A, B, C} \otimes \text{id}_D \downarrow & & \downarrow \alpha_{A, B, C \otimes D} \\ (A \otimes (B \otimes C)) \otimes D & & A \otimes (B \otimes (C \otimes D)) \\ \alpha_{A, B \otimes C, D} \searrow & & \nearrow \text{id}_A \otimes \alpha_{B, C, D} \\ & A \otimes ((B \otimes C) \otimes D) & \end{array}$$

Figure 1. Pentagon identity

Definition 2 (Symmetric Monoidal Category). A *symmetric monoidal category* (smc for short) is a monoidal category $(\mathcal{C}, \otimes, I)$ equipped with a natural isomorphism

$$\sigma_{A,B}: A \otimes B \rightarrow B \otimes A \quad (\text{symmetry})$$

such that $\sigma_{B,A} \circ \sigma_{A,B} = \text{id}_{A \otimes B}$ and the diagram of Figure 2 commutes.

$$\begin{array}{ccccc} (A \otimes B) \otimes C & \xrightarrow{\alpha_{A, B, C}} & A \otimes (B \otimes C) & \xrightarrow{\sigma_{A, B \otimes C}} & (B \otimes C) \otimes A \\ \sigma_{A, B} \otimes C \downarrow & & & & \downarrow \alpha_{B, C, A} \\ (B \otimes A) \otimes C & \xrightarrow{\alpha_{B, A, C}} & B \otimes (A \otimes C) & \xrightarrow{B \otimes \sigma_{A, C}} & B \otimes (C \otimes A) \end{array}$$

Figure 2. Hexagon identity

In this paper, all categories will be equipped with a symmetric monoidal structure (using \otimes for the tensor and I for the unit), unless stated otherwise. For the sake of readability, we mostly treat associator and unitors as identities, as justified by Mac Lane’s coherence theorem (see [ML63, Theorem 5.2] for the historical statement and [ML71, Chapter 7] for the more standard, textbook version).

String diagrams. As in [BCLS20], we use string diagrams, read from top to bottom, for a graphical representation of some categorical equations (see [JS91] for a historical introduction and [Sel10] for a survey of graphical languages). Given two morphisms $f: A \rightarrow B$ and $g: B \rightarrow C$, the composition $g \circ f: A \rightarrow C$ is presented in Figure 3a. The tensor of $f: A \rightarrow B$ and $g: C \rightarrow D$ is represented using two wires side by side as in Figure 3b, and the symmetry by crossing the wires as in Figure 3c. We will often omit the labels on wires if they are clear from the context; we also omit I wires because we treat unitors as identities.

Moreover, differential categories (introduced in Section 4) involve an endofunctor $!$. Following [BCS06], we represent $!(f): !A \rightarrow !B$ with a squared box (Figure 3d).

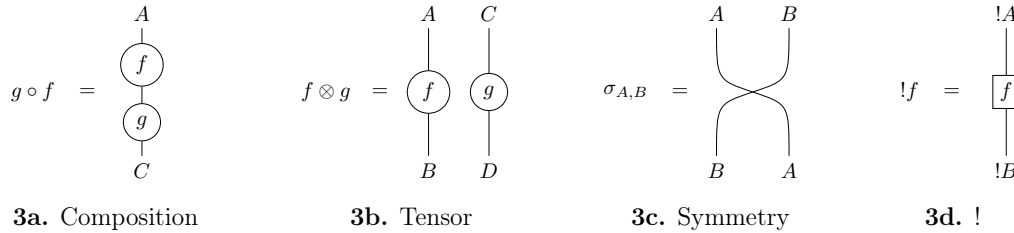


Figure 3. String diagrams

(Co)Monoids. Finally, most categories we present are equipped with (co)monoids.

Definition 3 (Monoids). A monoid in a smc \mathcal{C} is an object A equipped with:

$$\begin{aligned} \mu_A: A \otimes A &\rightarrow A && \text{(multiplication)} \\ \eta_A: I &\rightarrow A && \text{(unit)} \end{aligned}$$

satisfying the following equations:

$$\begin{aligned} \mu_A \circ (\mu_A \otimes \text{id}_A) &= \mu_A \circ (\text{id}_A \otimes \mu_A) && \text{(associativity of } \mu) \\ \mu_A \circ (\eta_A \otimes \text{id}_A) &= \text{id}_A = \mu_A \circ (\text{id}_A \otimes \eta_A) && \text{(neutrality of } \eta) \end{aligned}$$

which are presented in the string diagrams of Figure 4.

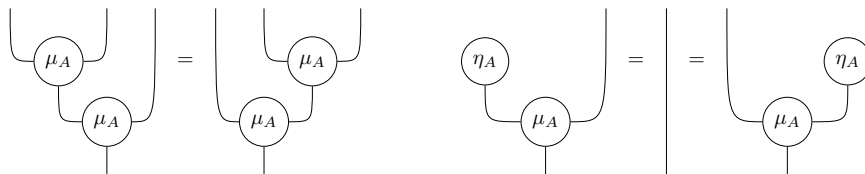


Figure 4. Monoid laws

Additionally, (A, μ_A, η_A) is *commutative* if $\mu_A \circ \sigma_{A,A} = \mu_A$.

Definition 4 (Comonoids). A comonoid in a smc \mathcal{C} is an object A equipped with:

$$\begin{aligned} \delta_A: A &\rightarrow A \otimes A && \text{(co-multiplication)} \\ \varepsilon_A: A &\rightarrow I && \text{(co-unit)} \end{aligned}$$

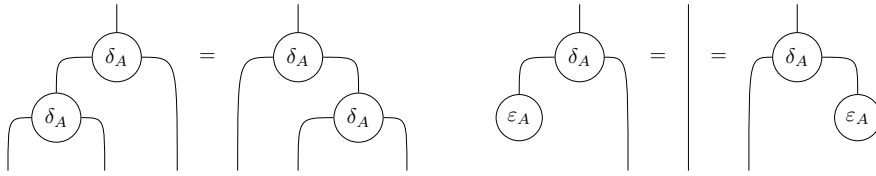


Figure 5. Comonoid laws

satisfying the equations of Figure 5.

Additionally, $(A, \delta_A, \varepsilon_A)$ is *commutative* if $\sigma_{A,A} \circ \delta_A = \delta_A$.

3 Resource Categories

Resource categories both capture the categorical structure of pointer concurrent games and allow us to identify morphisms behaving *linearly*, such as terms of resource calculus. First, composition in games generates sums of morphisms (which are also morphisms); likewise, substitution in the resource calculus generates sums of terms. Hence, resource categories have an *additive* structure. Moreover, resource terms are built using multisets of terms; we would like a way to “flatten” multisets of morphisms into one morphism. This operation is constructed with *bialgebra* morphisms. Finally, resource categories are *not* linear, because strategies, the morphisms in pointer concurrent games, do not have a linear behavior in general. However, we want to be able to characterize the morphisms that do behave linearly; which is achieved using the *pointed identities* morphisms.

Additivity. We call *additive* categories that are enriched over commutative monoids².

Definition 5 (Additive SMC (ASMC)). An *additive symmetric monoidal category* (asmc) is a symmetric monoidal category (see Definition 2) where each hom-set is a commutative monoid, with an addition $+$ and a zero 0 , such that composition and tensor distribute over the additive structure:

$$\begin{aligned} h \circ (f + g) \circ k &= h \circ f \circ k + h \circ g \circ k & h \circ 0 \circ k &= 0 \\ h \otimes (f + g) \otimes k &= h \otimes f \otimes k + h \otimes g \otimes k & h \otimes 0 \otimes k &= 0 \end{aligned}$$

for any morphisms k, f, g, h .

Bialgebras. Resource categories are equipped with *bialgebras*, which are a monoid and a comonoid with coherence laws between the morphisms.

Definition 6 (Bialgebra). Consider C an additive symmetric monoidal category.

A *bialgebra* on C is $(A, \delta_A, \varepsilon_A, \mu_A, \eta_A)$ with

- (A, μ_A, η_A) a commutative monoid (see Definition 3),
- $(A, \delta_A, \varepsilon_A)$ a commutative comonoid (see Definition 4),
- and additional bialgebra laws presented in Figure 6.

In resource categories, every object has a bialgebra structure. Intuitively, comonoids (A, δ_A, η_A) are a way to represent *duplications* and duplicable objects: if a request is made on the output of δ_A on either side of the tensor, the request is forwarded to its input. Monoids

²We follow the definition of [BCS06, Section 2], which differs from the one given in [ML71].

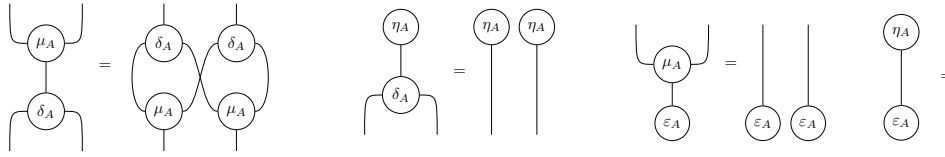


Figure 6. Bialgebra laws

$(A, \mu_A, \varepsilon_A)$ reflect the sums coming from compositions of strategies: requests made on the output of μ_A are forwarded non-deterministically (*via* a sum) to its input on either side of the tensor.

Strategies in pointer concurrent games are sums of augmentations, and augmentations have a forestial structure: they are, in a way, finite multisets of tree-like sub-augmentations. This matches the fact that in resource calculus, terms are applied to multiset of terms instead of terms. Bialgebra morphisms allow us to formalize this intuition and to flatten any multiset of morphisms into a single morphism. Indeed, for any objects A, B , we define the empty multiset of morphisms from A to B as:

$$1_{A,B} = \eta_B \circ \varepsilon_A \in \mathcal{C}(A, B),$$

and for any morphisms $f, g: A \rightarrow B$, the “union”:

$$f * g = \mu_B \circ (f \otimes g) \circ \delta_A \in \mathcal{C}(A, B),$$

capturing the idea of the union of the multisets f and g .

With these definitions, $(\mathcal{C}(A, B), *, 1_{A,B})$ is a commutative monoid (and $\mathcal{C}(A, B)$ is a commutative semiring, where the composition and the tensor only preserve the additive monoid). We define the n -ary union (unambiguously thanks to the associativity of $*$): given a multiset of morphisms $\bar{f} = [f_1, \dots, f_n]$ in $\mathcal{M}_f(\mathcal{C}(A, B))$, we set

$$\Pi \bar{f} = f_1 * \dots * f_n \in \mathcal{C}(A, B).$$

Hence, we send multisets of morphisms to single morphisms *via* Π .

Pointed identities. Finally, we wish to characterize morphisms that “behave linearly” (in pointer concurrent game, they correspond to singleton multisets of tree-like augmentations, using their argument exactly once). To do so, we define a morphism called *pointed identity*, which acts as an identity *only for “linear morphisms”*.³

Definition 7 (Pointed identity [BCVA23, Definition 22]). Consider \mathcal{C} an asmc where each object has a bialgebra structure. For any A , a *pointed identity* is $\text{id}_A^\bullet \in \mathcal{C}(A, A)$ satisfying:

$$\begin{aligned} \text{id}_A^\bullet \circ \text{id}_A^\bullet &= \text{id}_A^\bullet && (\text{idempotent}) \\ \varepsilon_A \circ \text{id}_A^\bullet &= 0 && (\text{non-erasable}) \\ \text{id}_A^\bullet \circ \eta_A &= 0 && (\text{non-erasing}) \end{aligned}$$

and the equations of Figure 7.

These equations express the following properties of id_A^\bullet :

non-duplicable: the post-composition with the co-multiplication δ_A is the sum of “ id_A^\bullet takes a request from the left-hand side of the tensor” and “ id_A^\bullet takes a request from the right-hand side”, but no situation in which id_A^\bullet takes requests from both sides simultaneously;

³The name *pointed identity* comes from the particular case of pointed identities in the resource categories of pointer concurrent games: tree-like augmentations corresponding to linear morphisms in games are called *pointed*, because their forestial structure has a unique root.

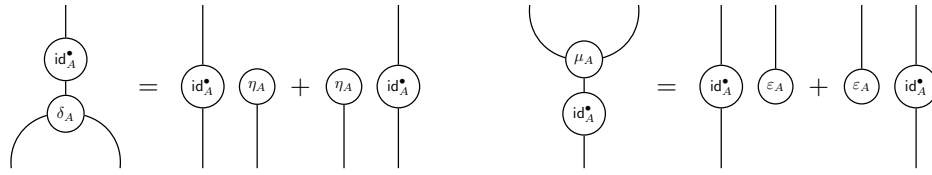


Figure 7. Laws for (co)multiplication and pointed identity

non-duplicative: the pre-composition with the multiplication μ_A is the sum of “ id_A^\bullet forwards a request to the left-hand side of the tensor” and “ id_A^\bullet forwards a request to the right hand side” but no “ id_A^\bullet forwards the request to both sides”.

This “strong linear” behavior of id^\bullet will allow us to characterize linear morphisms: those which are invariant by composition with the pointed identity.

Definition 8 ((Co-)Pointed Morphisms [BCVA23]). Consider A, B in an asmc \mathcal{C} equipped with bialgebras, and the pointed identities id_A^\bullet and id_B^\bullet .

Then $f \in \mathcal{C}(A, B)$ is *pointed* if $\text{id}_B^\bullet \circ f = f$. We write $f \in \mathcal{C}_\bullet(A, B)$.

Dually, f is *co-pointed* if $f \circ \text{id}_A^\bullet = f$. We write $f \in \mathcal{C}^\bullet(A, B)$.

Intuitively, pointed morphisms are morphisms behaving linearly for the substitution: they can only be used exactly once. Dually, co-pointed morphisms are morphisms behaving linearly with their arguments: they require exactly one resource.

Resource Categories We can now define resource categories.

Definition 9 (Resource Category [BCVA23, Definition 23]). Consider an asmc \mathcal{C} . It is a *resource category* if each object A has a bialgebra structure $(A, \delta_A, \varepsilon_A, \mu_A, \eta_A)$ with a pointed identity id_A^\bullet , and bialgebras are compatible with the monoidal structure of \mathcal{C} in the sense that the morphisms satisfy:

$$\varepsilon_{A \otimes B} = \lambda_I \circ (\varepsilon_A \otimes \varepsilon_B) \quad \eta_{A \otimes B} = (\eta_A \otimes \eta_B) \circ \lambda_I \quad \varepsilon_I = \eta_I = \text{id}_I$$

and the equations of Figure 8.

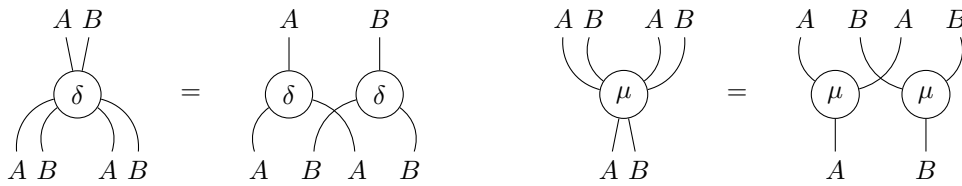


Figure 8. Compatibility of (co)monoids with the monoidal structure

Resource categories offer an interpretation of resource calculus, in which (singleton multisets of) terms are pointed morphisms. Linearity here is characterized using pointed identities; but linearity can also be linked to differential categories. Pointed identity laws are very similar to the dereliction and codereliction morphisms laws which occur in differential categories, which will guide us in our construction of a resource category in Section 5.

4 Differential Categories

Differential categories in general were introduced as a categorical framework for differential linear logic. In this paper, we focus on the *storage categories* of [BCLS20] – more precisely

on additive monoidal storage categories, which are the categories from which we construct resource categories in Section 5. In the current section we introduce all the components needed to define these storage categories (which we then define).

Coalgebra modality. *Coalgebra modalities* are similar to comonoid seen in Section 2, but they build over a comonad.

Definition 10 (Comonad). Consider a category \mathcal{C} . A *comonad* on \mathcal{C} is $(!, \text{dig}, \text{der})$ with

$$\begin{array}{ll} !: \mathcal{C} \rightarrow \mathcal{C} & \text{an endofunctor,} \\ \text{dig}_A: !A \rightarrow !!A & \text{a natural transformation,} \\ \text{der}_A: !A \rightarrow A & \text{a natural transformation,} \end{array}$$

satisfying the equations of Figure 9

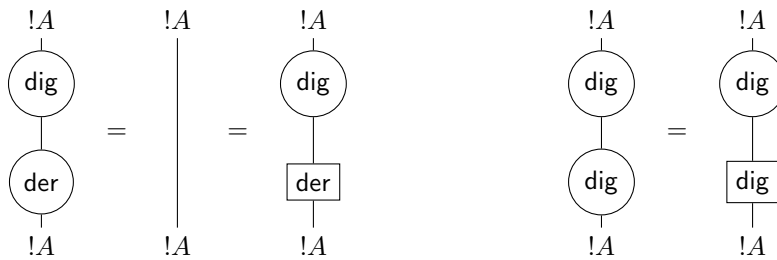


Figure 9. Comonad Laws

We write **dig** and **der** for the natural transformations because they match the *digging* and *derelection* rules of linear logic (introduced in [Gir87]).

Definition 11 (Coalgebra modality [BCLS20, Definition 1]). A *coalgebra modality* on a symmetric monoidal category \mathcal{C} is $(!, \text{dig}, \text{der}, \Delta, \text{e})$ with $(!, \text{dig}, \text{der})$ a comonad and two natural transformations

$$\Delta_A: !A \rightarrow !A \otimes !A \qquad \text{e}_A: !A \rightarrow I$$

such that for any A , $(!A, \Delta_A, \text{e}_A)$ is a comutative comonoid (Definition 4) and **dig** preserves Δ in the sense that

$$\Delta_{!A} \circ \text{dig}_A = (\text{dig}_A \otimes \text{dig}_A) \circ \Delta_A.$$

Bialgebra modality. Next, we define *bialgebra modalities*, which again are reminiscent of bialgebras seen in previous sections.

Definition 12 (Bialgebra modality [BCLS20, Definition 4]). Consider an asmc \mathcal{C} . A *bialgebra modality* on \mathcal{C} is $(!, \text{dig}, \text{der}, \Delta, \text{e}, \nabla, \text{u})$ with $(!, \text{dig}, \text{der}, \Delta, \text{e})$ a coalgebra modality and for any A , $(!A, \Delta_A, \text{e}_A, \nabla_A, \text{u}_A)$ is a bialgebra such that:

$$\text{der}_A \circ \nabla_A = (\text{der}_A \otimes \text{e}_A) + (\text{e}_A \otimes \text{der}_A).$$

Definition 13 (Additive bialgebra modality [BCLS20, Definition 5]). An *additive bialgebra modality* in an asmc \mathcal{C} is a bialgebra modality $(!, \text{dig}, \text{der}, \Delta, \text{e}, \nabla, \text{u})$ compatible with the additive structure in the sense of Figure 10.

Additive bialgebra modalities can be equipped with a *coderelection*, a natural transformation $\text{cod}_A: A \rightarrow !A$ named coderelection because it has the inverse type to der_A , but which is *not* an inverse of der_A .

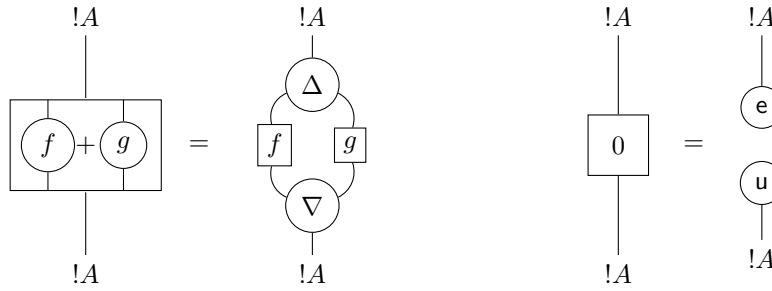


Figure 10. Additive Bialgebra Modality Laws

Definition 14 (Codereliction [BCLS20, Definition 9]⁴). Consider an asmc \mathcal{C} . A *codereliction* for an additive bialgebra modality $(!, \text{dig}, \text{der}, \Delta, e, \nabla, u)$ is a natural transformation $\text{cod}_A: A \rightarrow !A$ satisfying the following equations:

$$\begin{aligned} e_A \circ \text{cod}_A &= 0 && (\text{constant rule}) \\ \text{der}_A \circ \text{cod}_A &= \text{id}_A && (\text{linear rule}) \end{aligned}$$

as well as the equations of Figure 11.

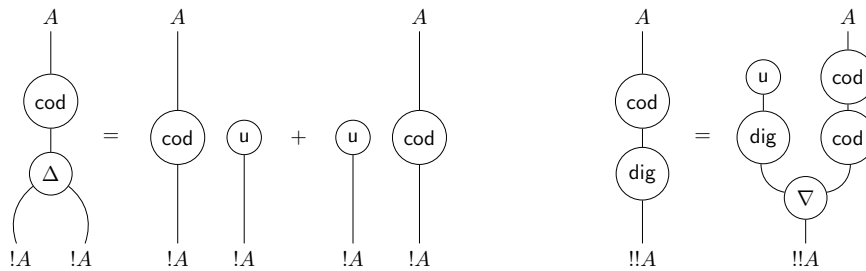


Figure 11. Product rule and chain rule of codereliction

Codereliction is a key notion of differential categories: in an asmc with a bialgebra modality, coderelictions induce deriving transformations⁵ ([BCS06, Theorem 4.12]). In an asmc with an *additive* bialgebra modality, coderelictions are *in bijection* with deriving transformations ([BCLS20, Theorem 4]).

Storage Categories. Now, we focus on *storage categories*, which are smc’s with a coalgebra modality and a cartesian product $\&$, with the following isomorphism:

$$!(A\&B) \cong !A\otimes!B$$

called *Seely isomorphism* (introduced as Δ iso in [See89]).

Recall that in a category \mathcal{C} , a *terminal object* is an object \top such that for any object $A \in \mathcal{C}$, there exists a unique morphism in $\mathcal{C}(A, \top)$, noted $!_A: A \rightarrow \top$. A category \mathcal{C} has *finite products* if it has a terminal object and for all objects $A, B \in \mathcal{C}$, there is a product $A\&B$ in \mathcal{C} satisfying the universal property of products.

⁴The chain rule equation given here is the version presented in [Fio07] and not the (slightly longer) version of [BCS06, Definition 4.11]; however both are equivalent in monoidal storage categories ([BCLS20, Lemma 7 and Corollary 5]), which are what we are interested in in this paper.

⁵More precisely they are in bijection with deriving transformations satisfying the ∇ -rule of [BCS06].

Definition 15 (Seely Isomorphism [BCLS20, Definition 10]). Consider an smc \mathcal{C} with a binary product $\&$, a terminal object \top , and a coalgebra modality $(!, \text{dig}, \text{der}, \Delta, \text{e})$. It has *Seely isomorphisms* if the map χ_\top and the natural transformation χ , respectively defined as:

$$\chi_\top: !\top \xrightarrow{\text{e}_\top} I \quad \chi_{A,B}: !(A\&B) \xrightarrow{\Delta_{A\&B}} !(A\&B) \otimes !(A\&B) \xrightarrow{! \pi_1 \otimes ! \pi_2} !A \otimes !B$$

are isomorphisms (with π_1, π_2 the projections of $\&$).

Definition 16 (Monoidal Storage Category [BCLS20, Definition 10]). A *monoidal storage category* is a smc with finite products and a coalgebra modality with Seely isomorphisms.

We can consider storage categories with an additive structure.

Definition 17 (Additive Monoidal Storage Category [BCLS20, Definition 11]). An *additive monoidal storage category* is a category \mathcal{C} that is a monoidal storage category and an additive symmetric monoidal category, with the same monoidal structure.

Additive storage categories are actually related to asmc's with a bialgebra modality.

Proposition 1 (from [BCLS20, Theorem 6]). *Consider an additive monoidal storage category \mathcal{C} . Then $(!, \text{dig}, \text{der}, \Delta, \text{e}, \nabla, \text{u})$ defined as:*

$$\begin{aligned} \Delta_A &: !A \xrightarrow{!(\text{id}_A, \text{id}_A)} !(A\&A) \xrightarrow{\chi_{A,A}} !A \otimes !A \\ \text{e}_A &: !A \xrightarrow{!0} !\top \xrightarrow{\chi_\top} I \\ \nabla_A &: !A \otimes !A \xrightarrow{\chi_{A,A}^{-1}} !(A\&A) \xrightarrow{\pi_1 + \pi_2} !A \\ \text{u}_A &: I \xrightarrow{\chi_\top^{-1}} !\top \xrightarrow{!0} !A \end{aligned}$$

is a bialgebra modality.

In [BCLS20], the authors even prove that those two notions are equivalent.

5 How to build your own resource category with only these simple ingredients

Construction. We start from an additive monoidal storage category, defined in Section 4.

Definition 18 ($\text{Res}(-)$). Consider an additive monoidal storage category \mathcal{C} with a codereliction cod . Using the notation of Proposition 1, we define the category $\text{Res}(\mathcal{C})$ with the same objects as \mathcal{C} and morphisms defined by:

$$\text{Res}(\mathcal{C})(A, B) = \mathcal{C}(!A, !B).$$

We define a bifunctor $\otimes_{\text{Res}(\mathcal{C})}$ in the following way:

$$\begin{aligned} A \otimes_{\text{Res}(\mathcal{C})} B &= A \& B \\ f \otimes_{\text{Res}(\mathcal{C})} g &= \chi_{C,D}^{-1} \circ (f \otimes g) \circ \chi_{A,B} \end{aligned}$$

for any objects A, B, C, D and morphisms $f \in \text{Res}(\mathcal{C})(A, C)$, $g \in \text{Res}(\mathcal{C})(B, D)$.

Indeed, morphisms of a resource category do not all behave linearly, which is why we define $\text{Res}(\mathcal{C})(A, B)$ as $\mathcal{C}(!A, !B)$: these are morphisms that are not necessarily linear with respect to A and B . To obtain a monoidal structure in $\text{Res}(\mathcal{C})$, we prove that $\otimes_{\text{Res}(\mathcal{C})}$ is a tensor, using Seely isomorphisms to see $!(A \otimes_{\text{Res}(\mathcal{C})} B)$ as $!A \otimes !B$. The additive bialgebra modality structure of \mathcal{C} easily induces a bialgebra structure in $\text{Res}(\mathcal{C})$ (which we will define precisely in next proof). Finally, recall the parting remark of Section 3: pointed identity laws are very similar to the dereliction and codereliction laws of differential categories. We will thus construct id^\bullet from der and cod .

Altogether, we obtain a resource category.

Theorem 1. Consider an additive monoidal storage category \mathcal{C} with a codereliction cod . Then $\text{Res}(\mathcal{C})$ is a resource category.

Proof. We use notations of Definition 18. To make the equations less cluttered, we write \mathcal{R} for $\text{Res}(\mathcal{C})$ and A for id_A , and we omit indices for χ when they are clear from the context.

SMC. First, we prove that $(\mathcal{R}, \otimes_{\mathcal{R}}, \top)$ is a smc (Definition 2). We define:

$$\begin{aligned} \alpha_{A,B,C}^{\mathcal{R}} &: !(A \& B) \& C \xrightarrow{\chi} !(A \& B) \otimes !C \xrightarrow{\chi \otimes !C} !(A \otimes B) \otimes !C \\ &\quad \xrightarrow{\alpha_{!A, !B, !C}^{\mathcal{C}}} !A \otimes !(B \otimes !C) \xrightarrow{!A \otimes \chi^{-1}} !A \otimes !(B \& C) \xrightarrow{\chi^{-1}} !(A \& (B \& C)) \\ \lambda_A^{\mathcal{R}} &: !(\top \& A) \xrightarrow{\chi} !\top \otimes !A \xrightarrow{\chi_{\top} \otimes !A} I \otimes !A \xrightarrow{\lambda_{!A}^{\mathcal{C}}} !A \\ \rho_A^{\mathcal{R}} &: !(A \& \top) \xrightarrow{\chi} !A \otimes !\top \xrightarrow{!A \otimes \chi_{\top}} !A \otimes I \xrightarrow{\rho_{!A}^{\mathcal{C}}} !A \\ \sigma_{A,B}^{\mathcal{R}} &: !(A \& B) \xrightarrow{\chi} !A \otimes !B \xrightarrow{\sigma_{!A, !B}^{\mathcal{C}}} !B \otimes !A \xrightarrow{\chi^{-1}} !(B \& A) \end{aligned}$$

and a direct diagram chasing, using smc properties of \mathcal{C} and the fact that χ is an isomorphism, shows that \mathcal{R} is a smc too.

Additivity. Direct from the definitions and the additive structure of \mathcal{C} .

Bialgebra structure. For any object A , we define the morphisms:

$$\begin{aligned} \delta_A^{\mathcal{R}} &: !A \xrightarrow{\Delta_A^{\mathcal{C}}} !A \otimes !A \xrightarrow{\chi^{-1}} !(A \& A) & \varepsilon_A^{\mathcal{R}} &: !A \xrightarrow{e_A} I \xrightarrow{\chi_{\top}^{-1}} !\top \\ \mu_A^{\mathcal{R}} &: !(A \& A) \xrightarrow{\chi} !A \otimes !A \xrightarrow{\nabla_A} !A & \eta_A^{\mathcal{R}} &: !\top \xrightarrow{\chi_{\top}} I \xrightarrow{u_A} !A \end{aligned}$$

Then one can check that $(A, \delta_A, \varepsilon_A, \mu_A, \eta_A)$ is a bialgebra by diagram chasing, using χ and the properties of the bialgebra modality of \mathcal{C} . Likewise, we check that it is compatible with the monoidal structure of \mathcal{R} (Figure 8).

Pointed Identity. Finally, for any object A , we define the pointed identity as:

$$\text{id}_A^{\bullet} : !A \xrightarrow{\text{der}_A} A \xrightarrow{\text{cod}_A} !A$$

and we check that it matches Definition 7:

- *idempotent:*

$$\begin{aligned} \text{id}_A^{\bullet} \circ \text{id}_A^{\bullet} &= \text{cod}_A \circ \text{der}_A \circ \text{cod}_A \circ \text{der}_A && \text{(definition of } \text{id}_A^{\bullet} \text{)} \\ &= \text{cod}_A \circ \text{id}_A \circ \text{der}_A && \text{(linear rule of Definition 14)} \\ &= \text{id}_A^{\bullet} && \text{(definition of } \text{id}_A^{\bullet} \text{)} \end{aligned}$$

- *non-erasable:*

$$\begin{aligned} \varepsilon_A^{\mathcal{R}} \circ \text{id}_A^{\bullet} &= \chi_{\top}^{-1} \circ e_A \circ \text{cod}_A \circ \text{der}_A && \text{(definition)} \\ &= \chi_{\top}^{-1} \circ 0 \circ \text{der}_A && \text{(constant rule of Definition 14)} \\ &= 0 && \text{(additivity)} \end{aligned}$$

- *non-erasing:*

$$\begin{aligned} \text{id}_A^{\bullet} \circ \eta_A^{\mathcal{R}} &= \text{cod}_A \circ \text{der}_A \circ u_A \circ \chi_{\top} && \text{(definition)} \\ &= \text{cod}_A \circ 0 \circ \chi_{\top} && \text{([BCLS20, Lemma 2]}^6 \text{)} \\ &= 0 && \text{(additivity)} \end{aligned}$$

- *non-duplicable*:

$$\delta_A^{\mathcal{R}} \circ \text{id}_A^\bullet = \chi_{A,A}^{-1} \circ \Delta_A \circ \text{cod}_A \circ \text{der}_A \quad (\text{definition})$$

and using string diagrams in $(\mathcal{C}, \otimes, I)$, we have:

by product rule (Definition 14, Figure 11) and additivity. Therefore,

$$\delta_A^{\mathcal{R}} \circ \text{id}_A^\bullet = (\text{id}_A^\bullet \otimes_{\mathcal{R}} u_A) + (u_A \otimes_{\mathcal{R}} \text{id}_A^\bullet)$$

again using Seely and the definition of id^\bullet .

- *non-duplicative*:

$$\text{id}_A^\bullet \circ \mu_A^{\mathcal{R}} = \text{cod}_A \circ \text{der}_A \circ \nabla_A \circ \chi_{A,A} \quad (\text{definition})$$

which gives us, using string diagrams in $(\mathcal{C}, \otimes, I)$:

by compatibility of der and ∇ (Definition 12) and additivity; that is

$$\text{id}_A^\bullet \circ \mu_A^{\mathcal{R}} = (\text{id}_A^\bullet \otimes_{\mathcal{R}} e_A) + (e_A \otimes_{\mathcal{R}} \text{id}_A^\bullet)$$

using Seely again and the definition of id^\bullet .

□

Closed Structure. In general, a category \mathcal{D} is closed if for any pair of objects A and B , $\mathcal{D}(A, B)$ can also be seen as an object of \mathcal{D} . In particular, for monoidal categories, \mathcal{D} is *monoidal closed* if there exists \multimap and an isomorphism natural in A, B, C such that:

$$\Lambda_{A,B,C}: \mathcal{D}(A \otimes B, C) \cong \mathcal{D}(A, B \multimap C)$$

⁶Actually $\text{der}_A \circ u_A = 0$ was part of the original definition of bialgebra modalities ([BCS06, Definition 4.8]), but it can be deduced from the other axioms and naturality of u and der ([BCLS20, Lemma 2]).

What happens if we consider \mathcal{C} as in Definition 18 a monoidal *closed* category? Does the closed structure also transport to $\text{Res}(\mathcal{C})$? Let us try to prove the isomorphism above for $\mathcal{R} = \text{Res}(\mathcal{C})$. Everything seems to go smoothly for the first part:

$$\begin{aligned} \mathcal{R}(A \otimes_{\mathcal{R}} B, C) &= \mathcal{C}(! (A \& B), !C) && \text{(definition)} \\ &\cong \mathcal{C}(!A \otimes !B, !C) && \text{(Seely isomorphism)} \\ &\cong \mathcal{C}(!A, !B \multimap !C) && \text{(closed structure of } \mathcal{C}) \end{aligned}$$

All that is left to do now is to define $\multimap_{\mathcal{R}}$ such that

$$\mathcal{R}(A, B \multimap_{\mathcal{R}} C) = \mathcal{C}(!A, !B \multimap !C),$$

but that is where the difficulty lies: there seems to be no obvious way to define $\multimap_{\mathcal{R}}$ such that $!(B \multimap_{\mathcal{R}} C) \cong !B \multimap !C$. In particular, it is clear that $!(B \multimap C)$ and $!B \multimap !C$ are not necessarily isomorphic. Hence, the question of whether or not we can build a *closed* resource category from a closed differential category remains open – at least with the construction presented in this paper.

6 Conclusion

We show a general construction of resource categories from differential categories; however, certain key properties such as closure do not seem to be preserved by this construction. All in all, Theorem 1 formalizes the expected link between resource categories and differential categories: resource calculus is the finitary fragment of differential λ -calculus, and resource categories are to resource calculus what differential categories are to differential λ -calculus. Thus it is not surprising that we can build resource categories from differential categories.

Yet there is still much to study on resource categories. For instance, we did not tackle the subject of cartesian structure for a resource category in this paper. However, the subcategory of comonoid morphisms is cartesian – to what strategies do they correspond in pointer concurrent games? Besides, morphisms interpreting finite resource terms do not form a subcategory, because they lack identities – how can we best describe their structure? What about finite strategies in general?

Finally, resource categories were introduced to better understand the links between resource terms and strategies: we hope to generalize this correspondence to the Taylor expansion of λ -terms (introduced in [ER03, Section 6]), an operation which translates a λ -term with a possibly infinite behavior to a multiset of resource terms, representing all its linear approximations. The first step of this correspondence was fleshed out in [BCVA23], and we wish to further investigate this question in future works.

References

- [AJM13] Samson ABRAMSKY, Radha JAGADEESAN et Pasquale MALACARIA : Full abstraction for PCF. *CoRR*, abs/1311.6125, 2013.
- [BC21] Lison BLONDEAU-PATISSIER et Pierre CLAIRAMBAULT : Positional injectivity for innocent strategies. In Naoki KOBAYASHI, éditeur : *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 de *LIPICs*, pages 17:1–17:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [BCL99] Gérard BOUDOL, Pierre-Louis CURIEN et Carolina LAVATELLI : A semantics for lambda calculi with resources. *Mathematical Structures in Computer Science*, 9:437 – 482, 1999.

- [BCLS20] Richard BLUTE, J. Robin B. COCKETT, Jean-Simon Pacaud LEMAY et Robert A. G. SEELY : Differential categories revisited. *Appl. Categorical Struct.*, 28(2):171–235, 2020.
- [BCS06] Richard BLUTE, J. Robin B. COCKETT et Robert A. G. SEELY : Differential categories. *Mathematical Structures in Computer Science*, 16:1049 – 1083, 2006.
- [BCVA23] Lison BLONDEAU-PATISSIER, Pierre CLAIRAMBAULT et Lionel VAUX AUCLAIR : Strategies as Resource Terms, and Their Categorical Semantics. In Marco GABOARDI et Femke van RAAMSDONK, éditeurs : *8th International Conference on Formal Structures for Computation and Deduction (FSCD 2023)*, volume 260 de *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:22, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [Bou93] Gérard BOUDOL : The lambda-calculus with multiplicities. In Eike BEST, éditeur : *CONCUR'93*, pages 1–6, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.
- [CCRW17] Simon CASTELLAN, Pierre CLAIRAMBAULT, Silvain RIDEAU et Glynn WINSKEL : Games and strategies as event structures. *Log. Methods Comput. Sci.*, 13(3), 2017.
- [ER03] Thomas EHRHARD et Laurent REGNIER : The differential lambda-calculus. *Theoretical Computer Science*, 309(1):1–41, 2003.
- [Fio07] Marcelo P. FIORE : Differential structure in models of multiplicative biadditive intuitionistic linear logic. In Simona Ronchi DELLA ROCCA, éditeur : *Typed Lambda Calculi and Applications*, pages 163–177, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.
- [Gir87] Jean-Yves GIRARD : Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987.
- [Gir88] Jean-Yves GIRARD : Normal functors, power series and λ -calculus. *Ann. Pure Appl. Log.*, 37:129–177, 1988.
- [HO00] J. M. E. HYLAND et C.-H. Luke ONG : On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.
- [JS91] André JOYAL et Ross STREET : The geometry of tensor calculus, i. *Advances in Mathematics*, 88:55–112, 1991.
- [Mel06] Paul-André MELLIÈS : Asynchronous games 2: The true concurrency of innocence. *Theor. Comput. Sci.*, 358(2-3):200–228, 2006.
- [ML63] Saunders MAC LANE : Natural associativity and commutativity. *Rice Institute Pamphlet - Rice University Studies*, 49:28–46, 1963.
- [ML71] Saunders MAC LANE : *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.
- [See89] Robert A. G. SEELY : Linear logic, *-autonomous categories and cofree coalgebras. 1989.
- [Sel10] Peter SELINGER : A survey of graphical languages for monoidal categories. In *New Structures for Physics*, pages 289–355. Springer Berlin Heidelberg, 2010.

- [TO16] Takeshi TSUKADA et C.-H. Luke ONG : Plays as resource terms via non-idempotent intersection types. In Martin GROHE, Eric KOSKINEN et Natarajan SHANKAR, éditeurs : *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science, LICS '16, New York, NY, USA, July 5-8, 2016*, pages 237–246. ACM, 2016.

Skeletal Semantics of a Fragment of Python

Martin Andrieux¹ and Alan Schmitt²

¹ENS Rennes, France

²INRIA, France

We present PySkel, a formalization of the semantics of a fragment of Python in Skel, a simple semantics description language. We describe a subset of the Python programming language including assignments, function calls, object oriented features, and exceptions. This subset is large enough to include challenges in the formalization of Python such as the handling of scopes. This formalization is used to generate an OCaml interpreter that can be used to run Python programs.

1 Introduction

The Python programming language is widely used, from teaching to research and industry. It is quite complex, however, for instance with its non-intuitive scoping rules. The closest artifacts to an official Python specification are the *Python Language Reference*, the *Python Standard Library*, and CPython, the reference implementation written in C. Unfortunately, neither of them are given as a formal semantics and only the latter is executable. For documentation and teaching purposes, and to provide the foundations for program analysis, it is most useful to have a formal and executable semantics of a language. We thus propose PySkel, a formal semantics of Python, written as a skeletal semantics [NS22] description, from which an OCaml interpreter can be extracted. Our work is based on Monat's denotational semantics of Python [Mon21].

The skeletal semantics description of a language can be thought of as an interpreter written in a strongly typed tiny functional language, called Skel. There are a few features that distinguish such a description from a usual interpreter. First, Skel allows for partial and non-deterministic programs with the `branch` construct. By partial, we mean that a Skel program can return no value, for instance with an empty set of branches. Next, it is not necessary to specify everything: some types or functions may be left unspecified, either because they are not crucial to the description of the language, because they are implementation dependent, or because they are specified later in an incremental definition. For instance, integers and associated operations such as addition are often left unspecified. Finally, Skel is a *syntax* to describe semantics. It is kept as simple as possible to be used with many tools, such as an OCaml interpreter generator, a debugger, or a Coq formalization generator.

Our contributions are a skeletal semantics of a fragment of Python that includes functions, classes, objects, and their complex scoping rules, as well as an OCaml interpreter derived from this semantics that can run Python programs and validate our semantics.

The paper is organized as follows. We introduce skeletal semantics in Section 2. We then focus on two challenging features of Python and their description in Skel: scopes in Section 3 and object-oriented aspects in Section 4. We show how to derive an interpreter in

```

type addr, heap, id
type expr
type stmt = | SAssign (id, expr)

val eval_expr : (heap, expr) -> (heap, addr)
val write : (heap, id, addr) -> heap

val eval_stmt ((h:heap), (s:stmt)) : heap =
  match s with
  | SAssign (id, expr) ->
    let (h', addr) = eval_expr (h, expr) in
    let h'' = write (h', id, addr) in
    h''
  end

```

Figure 1. A Simple Skeletal Semantics

Section 5 and in particular describe how we validate our approach in Section 5.3. We discuss related work in Section 6 and conclude in Section 7. Our formal semantics and interpreter are available [online](#).¹

2 Monadic Skeletal Semantics

We first describe Skel, the language of skeletal semantics, and how it leverages monadic features to model complex effectful computations in its simple functional language while keeping readability. We shortly describe the features we need for PySkel, a longer introduction to Skel and monads is available in [KS].

2.1 Skeletal Semantics

Consider the Python statement $x = e$. To provide the semantics of this statement, we write a Skel program containing an evaluation function `eval_stmt`, which takes a statement and other arguments such as the heap as input. A toy version of this Skel description is presented in Figure 1.

In this example, we start by declaring types. Some of them are unspecified (they do not have a definition): `addr`, `heap`, `id`, and `expr`. The first three are considered implementation-dependent, while the `expr` type for expressions is specified later in this paper. The `stmt` type is specified as a data type with a single constructor in this example. Note that it may depend on unspecified types, as is the case here. Then, we declare two unspecified functions `eval_expr` and `write` by giving their types. These functions cannot be specified as they manipulate data of unspecified types. Finally, we provide a specified function, `eval_stmt`, which inspects the statement using pattern matching and modifies some state.

To go in more details, the `eval_stmt` function takes as arguments a heap and a statement. It pattern-matches the statement, and in the (only) case it is an assignment of `id` to `expr`, it executes the relevant code. First, it evaluates `expr`, returning a new heap `h'` and an address. Indeed, as in Monat's work [Mon21], the evaluation of an expression returns a heap address. We then modify the heap according to the assignment and return it. All these steps are sequenced using the `let p = e1 in e2` operator, which evaluates `e1`, matches its result against `p`, then evaluates `e2`.

This simplified code (which does not include local environments nor control-flow issues such as returning from a function call or raising an exception) is already quite cumbersome, as we explicitly manipulate the heap at every step. Adding exceptions requires that we additionally check at every evaluation step that the returned address is not an exception,

¹<https://gitlab.inria.fr/skeletons/pyskel/-/tree/jfla2024>

```

type flag =
  | Ret addr
  | Brk
  | Cont
  | Exn addr
type exn<a> =
  | Cur a
  | Flag flag

type s = ( heap: python_heap
          , global_scope: global_scope
          , scope_heap: scope_heap (* functions (shared) *)
          , scope_stack: list<map<addr>> (* classes (not shared) *)
          )

type r = ( local_env: env
          , builtins: builtins
          )

type m<a> = (r, s) → (exn<a>, s)

val return<a> (v : a) : m<a> =
  λ(_, s):(r, s) → (Cur<a> v, s)

val bind<a, b> ((w: m<a>), (f: a → m<b>)) : m<b> =
  λ(r, s):(r, s) →
    let (vo, s') = w (r, s) in
    match vo with
    | Cur a → let fa = f a in fa (r, s')
    | Flag f → (Flag<b> f, s')
    end

binder @ = bind

```

Figure 2. Reader, State, and Exception Monad

and adding environments requires a new argument to the evaluation function. To deal with these issues, we show in the next section how to transform this code in a monadic version that seamlessly propagates such effects.

2.2 Monadic Style

The use of monads to add effects to a functional programming language is well-known [Wad90]. Skel, having first-order functions and polymorphic types, can be easily equipped with monads. We use some syntactic sugar to make the resulting code easier to read.

The monad we use in our semantics is given Figure 2. It combines three monads: a *reader monad* to give read-only access to a local environment and to built-in values (record type \mathbf{r}), a *state monad* for the read-write access to the heap and the scopes (record type \mathbf{s}), and an *exception monad* indicating whether the result is a normal value or not (algebraic type $\mathbf{exn}\langle a \rangle$, detailed below).

More precisely, the monadic type we consider is $\mathbf{m}\langle a \rangle$, where $\langle a \rangle$ is a polymorphic annotation. This type for computations is defined as $(\mathbf{r}, \mathbf{s}) \rightarrow (\mathbf{exn}\langle a \rangle, \mathbf{s})$: a computation takes a current environment and state as input, and it returns a possibly exceptional result and a new state. The type $\mathbf{exn}\langle a \rangle$ is either $\mathbf{Cur}\ a$, an actual result of type a , or $\mathbf{Flag}\ \mathit{flag}$, indicating a control-flow breaking result. This flag can signal the return of a function $\mathbf{Ret}\ \mathit{addr}$, where addr is the returned value, the breaking out or continuing of a loop, and an exception $\mathbf{Exn}\ \mathit{addr}$ where addr is the exception object.

```

val eval_expr : expr → m<addr>
val write : (id, addr) → m<()>

val eval_stmt (s: stmt) : m<()> =
  match s with
  | SAssign (x, v) →
    let addr =@ eval_expr v in
    write (x, addr)

```

Figure 3. Monadic Skeletal Semantics (excluding types)

The monad associated to this computation type has the usual two functions. The function `return<a>` takes a pure value of type `a` and puts it in the monad as a computation of type `m<a>`. Its code, depicted in Figure 2, is the anonymous function $\lambda(_, s):(r, s) \rightarrow (\text{Cur}\langle a \rangle v, s)$ that takes a pair $(_, s)$, ignoring environment and naming the state `s`. It then returns a normal result `Cur<a> v` alongside the unchanged state `s`. This code shows that constructors of polymorphic types must be annotated with the instantiation type. The function `bind<a,b>` takes a computation `w` of type `m<a>`, a continuation `f` of type `a → m` (given a pure value of type `a`, return a computation of type `m`), and it chains them together to return a computation of type `m`. To this end, it first takes the current environment and state $(\lambda(r, s):(r, s) \rightarrow \dots)$. It then runs the computation `w` by giving it the environment and the state, which returns a potentially exceptional value `vo` and a new state `s'`. If the value is a pure value `Cur a`, then the continuation is called with it, passing along the initial environment and the modified state.² If the result is a flag, then the continuation is not called and the flag is directly returned with the modified state. Note that the type of the flag is changed to the expected output type, as required by our strongly typed language.

To streamline notations, a symbol can be associated to a binder (`binder @ = bind`), so that code of the form `bind e1 ($\lambda v \rightarrow e2$)` can be simply written `let v =@ e1 in e2`. This is similar to OCaml's (`let*`) and Haskell's leftarrows in `do`-notation. The resulting Skel code is given in Figure 3, where the result of the `eval_stmt` function lives in the `m<()>` monadic type, i.e., computations that return unit.

2.3 Skeletal Description of Python Semantics

The skeletal description of the fragment of Python we consider is mostly contained in the `skeletal.sk` file. This file contains six main sections: the syntax of Python, the declaration of builtins, the definition of the state, environment, and flags types, the monad used by the interpreter, the read and write functions that deal with scopes, and finally the evaluation functions themselves.

The semantics of Python is quite usual, but some points deserve special attention. In particular, scope management differs from most functional languages (Section 3). We also focus on the object oriented features (Section 4) and the management of the initial state of the interpreter (Section 5).

3 Scopes

We describe in this section the way scopes are managed in Python and its implementation in Skel. We focus on functions in this section, the scopes for classes have some further subtleties that are detailed in Section 4.

²Skel is so simple that it does not have *n*-ary application, hence the application of `f` to `a` and (r, s') must be done in two steps with a `let`-binding.

```

x = 0
def foo():
    y = 1
    def bar():
        z = 2
        print(z)
        print(y)
        print(x)
    bar()
foo()

x = 0
def foo():
    x = 1
    print("foo:", x)
    foo()
    print("glb:", x)
foo()

x = 0
def foo():
    global x
    x = 1
    print("foo:", x)
    foo()
    print("glb:", x)
foo()

def foo():
    x = 0
    def bar():
        nonlocal x
        x = 1
        print("bar:", x)
    print("foo:", x)
    bar()
    print("foo:", x)
foo()

```

Figure 4. Scopes in Python

3.1 Scopes in Python

Scopes in Python are lexical: they correspond to the embedding of function declarations. The outermost scope is the *global scope*. When accessing a variable, the local scope is first explored, then the surrounding scope, up until the global scope. This is illustrated by the first program of Figure 4 which prints in order 2, then 1, then 0.

In Python, there is no notion of variable declaration to state that a variable should be added to a scope. The first syntactic occurrence of an assignment to a variable corresponds to a declaration of this variable in the current scope. To illustrate this, consider the second program of Figure 4. The inner function has a local `x` which is set to 1, so the scope of `foo` has this variable. Hence, the program first prints `foo:1` for the local `x` when `foo` is called, then `glb:0` for the global (different) `x`.

To tell Python that an assignment is not a variable declaration, one should state the scope of the variable before any assignment, so that this assignment does not declare the variable. One may state that a variable is global, as shown in the third program. This program prints `foo:1` and `glb:1`, illustrating that the assignment in `foo` did change the global `x`, hence they correspond to the same variable. Note that the global annotation for a variable also impacts how it is read. The following code for instance outputs 3. Indeed, as `x` is not defined in `baz`, it is looked up in `bar`, where it is said to be global, thus bypassing the declaration in `foo`.

```

x = 3
def foo():
    x = 1
    def bar():
        global x
        def baz():
            print(x)
        baz()
    bar()
foo()

```

Alternatively, a variable assignment may refer to an enclosing scope that is not the global scope. This is illustrated in the fourth program of Figure 4, where `x` is local to the function `foo`. In the enclosed function `bar`, `x` is declared as `nonlocal`, which means it must be declared in an enclosing function. It does not have to be the immediately enclosing function, but it cannot be the global scope. When executed, this program prints `foo:0`, `bar:1`, and `foo:1`.

Note that the `global` and `local` keywords only matter for determining which variable to access and that they are syntactic constructs. For instance, the variable written by an assignment is statically known when parsing the program. However, with the exception of the global scope, scopes are created at run-time (typically when calling a function), and they may persist after the function's completion. Consider code of Figure 5. After calling

```

def counter(init):
    x = init
    def reset():
        nonlocal x
        x = 0
        print(x)
    def incr():
        nonlocal x
        x = x + 1
        print(x)
    return(reset,incr)

r,i = counter(5)
i(); r(); i()

```

Figure 5. Lingering Scopes

`counter`, the two returned functions share its scope as enclosing scope, where the nonlocal variable `x` is defined: the scope still needs to be present after the call to `counter` is finished as `r` and `i` may be called. The code thus prints 6, 0, then 1.

To summarize, the *global scope* contains all the toplevel declarations, it is a mapping from variable names to heap addresses. Calling a function creates a new scope containing the parameters and the local variables of the function. Functions may need to access enclosing scopes to read or write non-local variables. We depict this dependency as a tree of scope, as shown in Figure 6. The dashed arrows to the global scope are only followed when reading variables that are not defined locally. In fact, the scopes of the `incr` and `wrapper` functions do not have an upper scope. We show on the right of each scope the names that are defined in this scope.

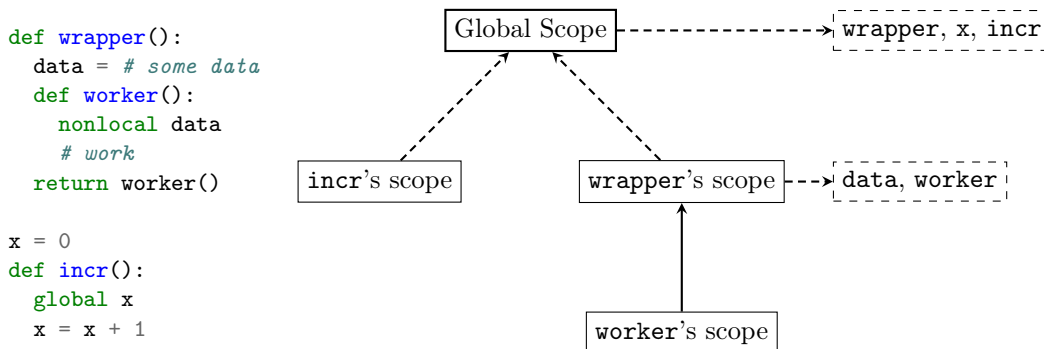


Figure 6. Simple Functions and Corresponding Scopes

Note that there is one scope per function *call*. The diagram in Figure 6 is valid if `wrapper` and `incr` have been called exactly once. The local variables are computed at parsing time, and are allocated at run time when the function call occurs. It is thus possible to read an allocated but not initialized variable. Such a read will raise `UnboundLocalError`, instead of the `NameError` exception that is raised when a variable does not exist. This is illustrated by the following code:

```

def f():
    x = undef # raises NameError as undef does not exist

def g():
    x = y # raises UnboundLocalError because y is local to g but not yet bound
    y = 0

```


3.2 Scopes in PySkel

PySkel extends Monat’s semantics with the previously described scopes. We distinguish two types of scopes, each representing a mapping from identifiers to values, but with some special properties.

First, we define a global scope, containing the top level functions and values. The global scope is unique and shared between all code, with a simple structure. In Skel, it is defined as the declaration `type global_scope = map<addr>`, i.e., a mapping from strings to addresses (see below). It is part of the mutable state described in Figure 2.

We next introduce the type of function scopes. As these scopes may contain uninitialized values, we define a `partial_addr` type, isomorphic to Haskell’s well-known `Maybe` type,³ with a `LocalUndef` constructor. In addition, function scopes can be shared and can access upper scopes. To account for this, we define an additional heap for scopes, so scopes are allocated and referenced with *scope identifiers*.

Scopes are a data structure to organize identifiers, but they are not sufficient. For example, to execute a variable assignment, one should know in which scope to look for it (i.e., is the variable local, global, or nonlocal). To this end, scopes are accompanied by a map from variable names to the kind of scope they should be searched in.

```
type var_scope = Local | Nonlocal | Global
```

This map is built as follows: global and nonlocal variables are explicitly given, and the remaining assignments are local variable declarations. So we need two pieces of information to manipulate variables in a function: the scope and the location map.

A function scope is thus a pair of a map from strings to `partial_addr` for the scope contents and an optional `scope_info` for the enclosing scope. The final type is presented below.

```
type heap<a> (* Unspecified heap, heap<a> contains values of type a *)
type heap_addr<a> (* heap_addr<a> are the addresses of heap<a> *)

type map<a> (* Unspecified map, map<a> is a mapping from strings to a *)
type maybe<a> = | Nothing | Just a

type partial_addr = | LocalUndef | Value addr
type var_scope = | Local | NonLocal | Global

type scope_info = (scope_id, map<var_scope>)
type partial_scope = (pmapping: map<partial_addr>, pscope_up: maybe<scope_info>)

type scope_id = heap_addr<partial_scope>
type scope_heap = heap<partial_scope>
```

The `var_scope` map is not part of the `partial_scope` type. This is because the map is read-only in a function, and we thus avoid to store it in the mutable state.

Note that the `heap` type is parameterized while the one presented in Figure 1 is not. As a consequence, we need to specify the type of values stored in the main Python heap. It is well described in Monat’s semantics (the two sets `ObjN` and `ObjS`, presented in Chapter 6 of his thesis), and can be directly translated to Skel. Heap values are composed of two elements, respectively of type `obj_n` and `obj_s`. The `obj_n` type represents the different kinds of values (`Int`, `String`, `Fun (...)`, etc). The `obj_s` part is a dictionary containing the fields of a value, with a special value `Locked` for primitive values. We thus define the `heap` type as follows.

```
type obj_n = | Int python_int | String python_string | ...
type obj_s = | Locked | Fields map<addr>
```

³We could also use the `maybe` type from PySkel, but we define a new one to clarify the code.

```

type addr = heap_addr<(obj_n, obj_s)>
type python_heap = heap<(obj_n, obj_s)>

```

Note that the `obj_n` type contains a constructor `Fun` for functions, this constructor includes a `scope` field indicating the scope where the function has been defined, i.e., its enclosing scope when called.

3.3 Function Calls and Memory Access

We now summarize the different actions performed to evaluate a function. The goal of this part is to provide a global understanding of functions and to justify the design of scopes.

3.3.1 Function Declaration

Evaluating a function declaration statement consists in the allocation of a `Fun` object on the heap. It is comprised of the name of the function, the list of its arguments, the list of its local variables, the code of its body, and the optional enclosing scope, which is set to the current scope unless the function is declared at top level.

3.3.2 Function Call

To call a function, we first extract the `Fun` object from the heap. We create a fresh scope containing the arguments of the function (bound to the values given in the function call) as well as the local variables (bound to `LocalUndef`). We set the `pscope_up` field of the fresh scope to the scope information stored in the `Fun` object.

Then, the variable map `var_map` is computed by inspecting the body of the function. It is currently not stored in the `Fun` object to keep the `obj_n` type close to Monat’s semantics. Once this is done, we evaluate the body with the new scope information, by locally setting the environment to `InFun` (`fun_scope`, `var_map`).

The environment, first declared in Figure 2, keeps track of the evaluation context. This context can have three forms: `Globl`, `InFun`, or `InClass`. `Globl` means “in the global scope” or “at top level”. It does not provide any additional information. The `InFun` state contains a `scope_info` and represents the evaluation of a function. The `InClass` is described in the next section. We need to be able to distinguish between contexts because the scoping rules are different in a function, a class, or at toplevel.

3.3.3 Memory Access

To read or write a variable in a function, we first look at the `var_scope` associated to the variable to know if it is local, nonlocal, or global. We can then look for the variable in the right scope, knowing that the identifier of the local scope is in the read-only state, the upper scope is stored in the local scope, and the global scope is stored in the mutable state.

4 Objects and Classes

Classes constitute a major aspect of Python. They are well described in Monat’s semantics, with the exception of the subtle interactions with function scopes. In this section, we first introduce the basic aspects of classes and objects in Python before detailing the scope issues and our implementation in Skel.

4.1 Classes in Python

A class defines a new *type*, consisting of a set of *fields*, declared in the class body. These fields include variables and function definitions. Classes can inherit from each other, the set of fields of a class and its super classes forms the *attributes* of the class.

The set of fields is given as a list of statements, just like a function body. It may contain global and non-local variables, declared with the `global` and `nonlocal` keywords. The variables so defined are not part of the fields of the class.

The notion of field is hidden in Python, but it is nevertheless necessary to understand the underlying mechanism of classes. In the code sample given in Figure 7, the class `Example` has three fields: `value`, `function`, and `method`. As the class inherits from `Super`, the set of attributes may be larger than the set of fields.

```
class Example(Super):
    value = 0
    def function():
        return 1

    def method(self, value):
        self.x = value
```

Figure 7. A Class in Python

To create an instance of some class, we call the class like a function: `Example()`. This creates a reference to the class called. Instances share the attributes of the superclass. These attributes can be accessed with the dot operator, as well as attributes of the instance itself: `Example().value`. Such access can have some side effects. In particular, if an instance accesses a function of its class, a new *method* is allocated and returned instead of the function. This method contains two addresses: the instance (i.e., calling object) and the function. Thus, the method call simply passes the instance as first parameter to the function. The given instance is modified according to the function body defined in the class. This mechanism is illustrated in Figure 8.

```
class Example:
    def method(self, value):
        self.x = value

instance = Example()

Example.method # is a function
instance.method # is a method, referencing `instance` and `Example.method`
instance.method(2) # is nearly the same as
Example.method(instance, 2) # "nearly" because no method is allocated here
```

Figure 8. Functions and Methods

As can be seen, instances can have additional fields that are not part of the class (`x` in the example). Those fields are stored on the heap, next to the object they belong to (as the structural part `ObjS` of the value representing the object). Note that some builtin object, like integers, are *locked*: they do not contain any such extra field. Trying to access or set them, as in `x = 1; x.y = 2` results in an attribute error.

As classes are well described in Monat's semantics, they are not studied any further in this article. However, we have also implemented nonlocal and global scopes in functions, so we need to explain how classes are affected by this addition.

4.2 Classes and Scopes in PySkel

In PySkel, the representation of classes contains a map from identifiers to addresses to store the values of the fields. This is the only non-trivial part of the evaluation of a class definition. The bodies of classes are similar to the ones of functions, including the declaration of `global` and `nonlocal` variables. A first approach for their evaluation would consist of allocating a

fresh *function* scope and evaluating the body in it, to populate the scope, hence to obtain the values of the fields of the class. We do something very similar in PySkel, the only difference is the scope type: functions scopes are designed for functions and are not adapted to class evaluation. To understand why, we next clarify the differences between classes and functions.

Firstly, class variables cannot be uninitialized, hence we do not need partial maps. Secondly, fields exist only in the class, they are not directly accessible from functions or classes defined inside the class (methods may still access them as fields of their `self` argument). Thus the scope chain need to bypass classes (see the body of the `bar` function in Figure 9, where the closest non-class scope where `x` is defined is the scope of `foo`). Finally, class scopes cannot be shared (as mutable fields only exist in the instance), so there is no need to store them on the heap.

```
def foo():
    x = 0
    class C:
        x = 1
        def bar():
            print(x)
    return C

foo().bar() # 0
```

Figure 9. Scopes Interaction Between Functions and Classes

Hence, we need a simple address map, like the global scope, with some upper scope for nonlocal variables. In case of nested classes, we may need to pause the construction of the current class scope to evaluate the subclass. This is why we have a `scope_stack` in the mutable state. At any point of the program, the length of the stack corresponds to the number of classes we entered to reach the program point. Once the body of the class is fully evaluated, we pop the scope from the stack to obtain the expected dictionary.

The final type of the environment is defined below. The `Globl` and `InFun` constructors have been introduced in the previous section. For classes, the context is composed of the `var_scope` map and the upper scope information. The current scope is at the top of the stack, hence we do not need a scope identifier.

```
type env =
  | Globl
    (* var_scope and current scope *)
  | InFun scope_info
    (* var_scope and scope_up *)
  | InClass (map<var_scope>, maybe<scope_info>)
```

5 Deriving an Interpreter

Once the semantics is written, we can use the various Skel backends to test and demonstrate the usability of the semantics. We describe in this section the process of deriving a Python interpreter using the OCaml generator backend, leaving the use of the Coq and debugger backends for future work.

5.1 Project Structure

The Skel toolchain contains a program called `necroml`. It takes a Skel semantics and translates it to an OCaml interpreter. The generated code takes the form of a functor

requiring an implementation of the unspecified types and terms. PySkel uses `necroml` to derive a simple Python interpreter which is used to test the semantics. For a better understanding of the PySkel project structure, we provide a summary diagram in Figure 10.

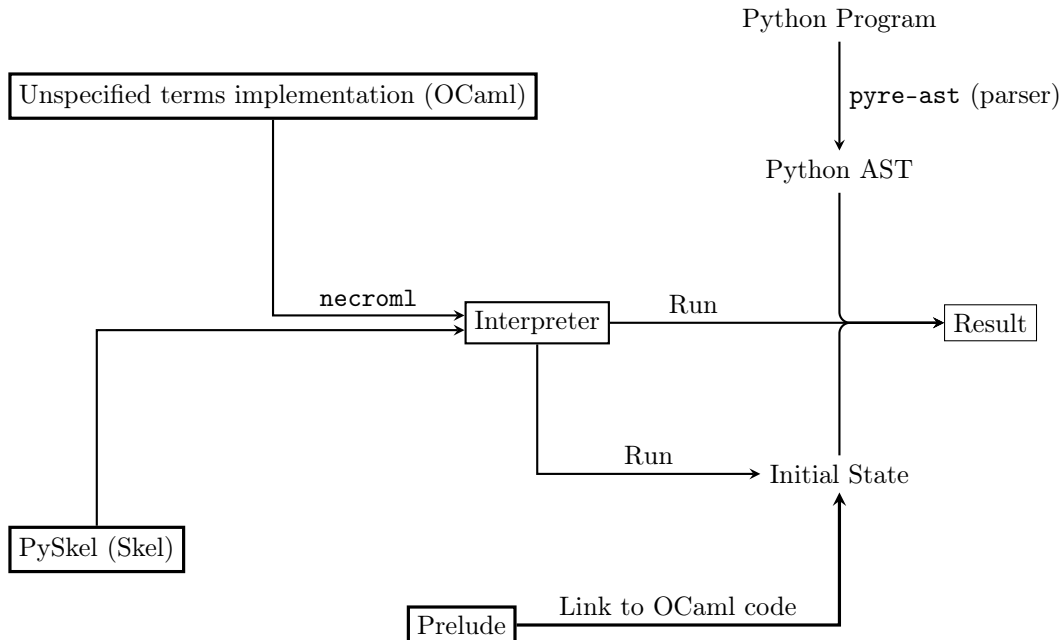


Figure 10. Structure of the PySkel Project

The `necroml` program generates specified terms and requires an implementation of unspecified ones. This implementation is written in OCaml and it is part of the PySkel project. With these two parts, a functor instantiation gives the Python interpreter. We can run this interpreter by providing a program and some initial state (which corresponds to the state part of the monad). This state is computed from a prelude file containing the builtin classes (such as exceptions) and some glue to fit well with the Skel semantics (see Section 5.2). Python programs are parsed with the `pyre-ast` OCaml library and then translated to the PySkel AST.

The unspecified part of PySkel is small. It is composed of primitive types (`bool`, `int`), functions that operate on these types, the heaps, and a polymorphic string map. The reason we chose to leave these parts unspecified is pragmatic: they are well-known basic features whose behavior does not need to be clarified in Skel. Note that if this came to change, one could simply replace the unspecified declarations with their chosen specification. The implementation of the unspecified parts is 100 lines long, compared to the 1200 lines of Skel (the instantiation files can be found in the `lib` directory, they are then used at the beginning of `utils.ml` to generate the interpreter).

5.2 Initial State and Internal Function

The evaluation of a Python program requires some initial state. This state must contain builtin objects, functions, and classes.

As the organization of builtin functions are not really part of Python evaluation mechanism, we do not want to specify this organization in the semantics. Note that functions are defined in the semantics, but not organized in classes. For example, the code to add integers is written in the Skel file, but not linked to any `int` class or `__add__` method. Hence, we need a way to organize existing Skel code into classes from outside the semantics.

This is done with a Python `prelude.py` file, containing declaration of classes and methods. We provide a way to relate it to Skel code with a decorator `@pyskel_internal`, as shown in Figure 11. The writer of the prelude file must specify a token name ("`IntAdd`" in the example) to identify the corresponding code. We can understand this as "when you try to call this function, call the Skel code associated to the `IntAdd` token instead". Then, in the semantics, the `call_internal` function redirects the call to execute the desired code, as shown in Figure 12. The purpose of this code is to specify the number of arguments, the order in which they are evaluated, the type of the arguments, and the behavior to follow if the builtin requirements are not met. As these are semantic aspects, we prefer to specify them directly in the code in Skel.

```
class int(object):
    @pyskel_internal("IntAdd")
    def __add__(self, other):
        pass
```

Figure 11. Declaring an Internal Function in a Python Class

```
type internal = | IntAdd

val call_internal ((internal : internal), (args : list<addr>)) : m<addr> =
  match internal with
  | IntAdd ->
    match args with
    | Cons (op1, Cons (op2, Nil)) ->
      let (objn1, _) =@ read op1 in
      let (objn2, _) =@ read op2 in
      match (objn1, objn2) with
      | (Int left, Int right) ->
        let result = internal_int_add (left, right) in
        alloc (Int result, Fields map_empty<addr>)
      | _ ->
        let r =@ ask in
        ret r.builtins.notImplemented
    end
  | _ ->
    raise_exn<addr> _TypeError
  end
end
```

Figure 12. `call_internal` Function in the Skel Semantics

The connection between the decorator and the internal call is done during the translation from the Pyre AST to the Skel AST.

5.3 Using the Derived Interpreter to Test the Semantics

Once the interpreter is derived, we are able to execute Python programs. The test suite of PySkel consists of a set of Python files, executed twice: once with the official CPython interpreter and once with PySkel. We then check if the results are the same. Any inconsistency means we need to correct our semantics, as the C interpreter of Python is considered as ground truth. We test semantics aspects as well as the construction of the initial state in the files found in the `class`, `exception`, `internal`, and `scope` directories. In addition to having a semantics that is very close to the one described by Monat, this gives us confidence to believe that our interpreter correctly evaluates Python programs, hence it can be used as a description of Python's semantics.

6 Related Work

There have been several proposals to formalize the semantics of Python. Monat gives a denotational semantics (on paper) in his PhD thesis [Mon21], which is used to build analyzers as part of the MOPSA platform [MOM21]. His semantics is quite faithful as it is derived from the official Python interpreter written in C. Our work is heavily based on this semantics, extended to deal with additional features absent from Monat’s thesis, such as nonlocal and global scopes, both for functions and classes. We also provide an interpreter, whereas Monat’s semantics is not executable. In addition, our Python semantics is written in Skel, so it can easily be reused for other tools.

An initial attempt to formalize the semantics of Python was done by Politz et al. as a small-step operational semantics [PMM⁺13]. This semantics is actually given as a translation, called *desugaring*, of Python in λ_π , a much smaller core language. The desugaring process and an interpreter for λ_π are both written in Racket [FFF⁺18]. When combined, they provide an interpreter for Python source code. We have relied on this work to grasp the subtlest aspects of Python’s scopes, in particular for classes defined inside functions. The distinguishing feature of our work compared to theirs is that we describe the semantics directly at the source code level. The desugared semantics can be quite different from the source, especially when it is CPS-transformed to deal with generators. Understanding the semantics of a Python program by looking at an execution of its desugared version may be challenging. Although we have not yet implemented generators, we believe we can seamlessly handle them, as discussed in Section 7.

Finally, some recent work aims at giving a semantics of Python’s bytecode through a formalization in F* [Kar22]. Once again, it is challenging to reason about a source program through the semantics of a translation, here into bytecode.

7 Conclusion and Future Work

We have presented PySkel, a formalization of the semantics of a fragment of Python in Skel, which we have used to derive a Python interpreter in OCaml. The development of this complex semantics has helped identify an area of improvement for Necro, mainly the inclusion of files, which is now available in a recent version of the tool. Although our formalization covers only a tiny subset of the standard library, we believe it is illustrative enough to be easily extended. We plan to add `for` loops next, as they pose some interesting questions regarding scopes and are not directly formalized in Monat’s thesis, before considering generators.

Generators can be seen as a way to interrupt the execution of a program, yielding an intermediate value and a computation to resume. They are quite complex to capture formally as they require handling partial computations and their continuations. Our plan to model their semantics in Skel is to reuse the work of Khayam et al. [KS] to use a delimited continuation monad. More precisely, they have shown that, since Skel is almost in administrative normal form [SF92], one may choose an appropriate monad to seamlessly capture side-effects, including the interruption of a computation because of a `yield`. The integration of their approach, only tested on a toy language, in this much larger semantics would challenge how seamless the approach is.

Finally, it would be most useful to be able to run CPython’s test suite, to more thoroughly test our work. Unfortunately, this test suite depends on the `unittest` framework, which uses native libraries and reflective features on modules. Adding these to PySkel is a long term goal.

References

- [FFF⁺18] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, 2018.
- [Kar22] Ammar Karkour. Py*: Formalization of Python’s Verifiable Bytecode and Virtual Machine in F*, 7 2022.
- [KS] Adam Khayam and Alan Schmitt. A practical approach for describing language semantics. https://people.rennes.inria.fr/Alan.Schmitt/papers/programming_draft.pdf.
- [MOM21] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A Multilanguage Static Analysis of Python Programs with Native C Extensions. In *Static Analysis Symposium (SAS)*, pages 1–23, Chicago, Illinois, United States, October 2021.
- [Mon21] Raphaël Monat. *Static Type and Value Analysis by Abstract Interpretation of Python Programs with Native C Libraries*. PhD thesis, Sorbonne Université, 2021.
- [Noi] Louis Noizet. Necro Debugger Generator, <https://gitlab.inria.fr/skeletons/necro-debug>.
- [NS22] Louis Noizet and Alan Schmitt. Semantics in Skel and Necro. In *ICTCS 2022 - Italian Conference on Theoretical Computer Science*, CEUR Workshop Proceedings, pages 1–17, Rome, Italy, September 2022.
- [PMM⁺13] Joe Gibbs Politz, Alejandro Martinez, Mae Milano, Sumner Warren, Daniel Patterson, Junsong Li, Anand Chitipothu, and Shriram Krishnamurthi. Python: the full monty. In Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes, editors, *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 217–232. ACM, 2013.
- [SF92] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the Conference on Lisp and Functional Programming, LFP 1992, San Francisco, California, USA, 22-24 June 1992*, pages 288–298. ACM, 1992.
- [Wad90] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming, LFP '90*, page 61–78, New York, NY, USA, 1990. Association for Computing Machinery.

Liveness Properties in Geometric Logic for Domain-Theoretic Streams

Colin Riba and Solal Stern

ENS Lyon, Université de Lyon, LIP*

We devise a version of Linear Temporal Logic (LTL) on a denotational domain of streams. We investigate this logic in terms of domain theory, (point-free) topology and geometric logic. This yields the first steps toward an extension of the “Domain Theory in Logical Form” paradigm to temporal liveness properties.

We show that the negation-free formulae of LTL induce sober subspaces of streams, but that this is in general not the case in presence of negation. We propose a direct, inductive, translation of negation-free LTL to geometric logic. This translation reflects the approximations used to compute the usual fixpoint representations of LTL modalities.

As a motivating example, we handle a natural input-output specification for the usual filter function on streams.

1 Introduction

We are interested in input-output properties of higher-order programs that handle infinite data, such as streams or non-wellfounded trees. Consider for instance the usual filter function

$$\begin{aligned} \text{filter} & : (A \rightarrow \text{Bool}) \longrightarrow \text{Str}A \longrightarrow \text{Str}A \\ \text{filter } p (a :: x) & = \text{if } (p a) \text{ then } a :: (\text{filter } p x) \text{ else } (\text{filter } p x) \end{aligned}$$

where $\text{Str}A$ stands for the type of streams on A . Assume $p : A \rightarrow \text{Bool}$ is a total function that tests for a property P . If x is a stream on A , then $(\text{filter } p x)$ retains those elements of x which satisfy P . The stream produced by $(\text{filter } p x)$ is thus only partially defined, unless x has infinitely many elements satisfying P .

Logics like LTL, CTL or the modal μ -calculus are widely used to formulate, on infinite objects, safety and liveness properties (see e.g. [HR07, BS07]). Safety properties state that some “bad” event will not occur, while liveness properties specify that “something good” will happen (see e.g. [BK08]). One typically uses temporal modalities like \square (*always*) or \diamond (*eventually*) to write properties of streams and specifications of programs over such data.

A possible specification for filter asserts that $(\text{filter } p x)$ is a totally defined stream whenever x is a totally defined stream with infinitely many elements satisfying P . We express this with the temporal modalities \square and \diamond . Let A be finite, and assume given, for each a of type A , a formula Φ_a which holds on $b : A$ exactly when b equals a .¹ Then $\square \bigvee_a \Phi_a$ selects

*UMR 5668 CNRS ENS Lyon UCBL INRIA

¹In the setting of [JR21], we would assume $A = \sum_{i=1}^n \mathbf{1}$, with Φ_i representing the image of the i th injection.

those streams on A which are totally defined. The formula $\Box\Diamond P$ expresses that a stream has infinitely many elements satisfying P . We can thus state that for all streams $x : \text{Str}A$,

$$x \text{ satisfies } \Box\bigvee_a \Phi_a \text{ and } \Box\Diamond P \implies (\text{filter } p \ x) \text{ satisfies } \Box\bigvee_a \Phi_a \quad (1)$$

The question we address in this paper is the following. Having in mind that a stream (as opposed to e.g. an integer) is inherently an infinite object, what do we mean exactly by “the stream x satisfies $\Box\Diamond P$ ”? In our view, the above specification for `filter` should hold for any stream whatsoever, and not only for those definable in a given programming language.

This leads us to investigate temporal properties on infinite datatypes at the level of denotational semantics. Logics on top of domains are known since quite a long time. Our reference is the paradigm of “Domain Theory in Logical Form” (DTLF) [Abr91] (see also [Zha91]), which allows one to systematically generate a logic from a domain representing a type. These logics are actually obtained by Stone duality, which is at the core of a rich interplay between domain theory, logic and (point-free) topology. This area is presented under various perspectives in a number of sources. We refer to [Abr91, AC98] and (e.g.) [Joh82, Vic89, Vic07, GL13, GvG23]. Some key ideas are put at work in [CZ00].

However, logics on domains given by Stone duality are usually restricted to safety properties. To our knowledge, there is no systematic investigation of liveness properties, such as the ones used in the specification for `filter` above.

This paper reports on preliminary works, mostly based on an internship of the second author during summer 2023. We devise a version of the logic LTL on a domain of streams $\llbracket \text{Str}A \rrbracket$ determined by the recursive type equation $\text{Str}A \cong A \times \text{Str}A$. Each formula Φ of LTL yields a subset $\llbracket \Phi \rrbracket \subseteq \llbracket \text{Str}A \rrbracket$. We investigate such LTL-definable subsets in terms of domain theory, of (point-free) topology and of a logic called geometric logic.

Our first step is to view domains as topological spaces, so as to benefit from the rich notion of subspace. For instance, (with A finite) the set of ω -words $A^\omega = \llbracket \Box\bigvee_a \Phi_a \rrbracket$ turns out to be a discrete sub-poset of $\llbracket \text{Str}A \rrbracket$. But as a subspace of $\llbracket \text{Str}A \rrbracket$, it becomes equipped with its usual product topology (in the sense of e.g. [PP04]). We observe that LTL formulae without negation induce subspaces of $\llbracket \text{Str}A \rrbracket$ which are sober, but that this may fail in presence of negation. The notion of sobriety originates from point-free topology, and has become quite important for the general (point-set) topology of domains (see e.g. [GL13]).

We then turn to geometric logic. The idea is roughly the following. DTLF rests on the fact that finite approximations in a domain can be represented in a propositional logic generated from the topology of the domain. But this is too weak to handle infinitary properties such as those definable with the modalities \Box and \Diamond . On the other hand, the sobriety of $\llbracket \Phi \rrbracket$ means that we can reason using an abstract notion of approximation induced by the subspace topology. Geometric logic is an infinitary propositional logic which allows for concrete representations of topologies. We provide a direct, inductive, translation of negation-free LTL to a geometric logic based on the domain $\llbracket \text{Str}A \rrbracket$. This translation reflects the approximations used to compute the usual fixpoint representations of \Box, \Diamond . This shows that for the negation-free fragment, the semantics of LTL can be concretely represented by approximations which live in a natural extension of DTLF for the domain $\llbracket \text{Str}A \rrbracket$.

We also check that our translation of negation-free LTL indeed conveys the good approximations to prove that the denotation of `filter` meets the specification (1) above.

Let us finally mention the scientific context of this work. It is undecidable whether a given higher-order program satisfies a given input-output temporal property written with formulae of the modal μ -calculus [KTU10]. A previous work with the first author provided a refinement type system for proving such properties [JR21]. This type system handles the alternation-free modal μ -calculus on (finitary) polynomial types, which includes LTL. But it is based on guarded recursion and does not allow for non-productive functions such as `filter`. We ultimately target a similar refinement type systems for a language based on FPC (which extends Plotkin’s seminal PCF [Plo77] with recursive types, see e.g. [Pie02]). We think that the present work is a significant step in this direction. On the one hand, DTLF allows for

reasoning on denotations using (finitary) type systems [Abr91]. On the other hand, it has been advocated in [KT14] that a form of oracle is needed to handle liveness properties in type systems. And indeed, [JR21] incorporates such oracles in a notion of “iteration term”, which in fact makes the system infinitary.² We think that our representation of negation-free LTL in geometric logic can lead to an infinitary type system which extends [Abr91], and whose infinitary part can be simulated using iteration terms.

Organization of the paper. The preliminary §2 introduces background on domain theory, and the logic LTL on $\llbracket \text{Str}A \rrbracket$. The (point-free) topological approach is presented in §3, and §4 is devoted to geometric logic. The specification of filter is discussed in §5. We conclude in §6. Proofs are available in the full version [RS23], which also contains additional material on deduction for geometric logic.

2 A Linear Temporal Logic on a Domain of Streams

Let A be a set. A (finite) *word* on A is an element of A^* . A^ω is the set of ω -words on A , i.e. the set of all functions $\sigma: \mathbb{N} \rightarrow A$. We write $u \subseteq v$ when $u \in A^*$ is a prefix of $v \in A^* \cup A^\omega$. The concatenation of $u \in A^*$ with $v \in A^* \cup A^\omega$ is denoted $u \cdot v$ or uv . Given $\sigma \in A^\omega$ and $k \in \mathbb{N}$, we let $\sigma \upharpoonright k \in A^\omega$ be the ω -word with $(\sigma \upharpoonright k)(n) = \sigma(k+n)$ for all $n \in \mathbb{N}$. For instance, $\sigma \upharpoonright 0$ is σ , while $\sigma \upharpoonright 1 = \sigma(1) \cdot \sigma(2) \cdots \sigma(n+1) \cdots$ is σ deprived from its first letter.

2.1 Domains

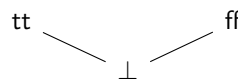
The basic idea of domain theory is to represent a type by partial order (X, \leq_X) thought about as an “information order”. The intuition is that $x \leq_X y$ means that y has “more information” than x , or that x is “less defined” than y . Domains are often required to have a least element (representing plain divergence), and are always asked to be stable under certain supremums (so that infinite objects can be thought about as limits of their finite approximations). Our presentation mostly follows [AC98, §1]. See also [Abr91, GL13].

Dcpo and Cpos. Let (X, \leq) be a partial order (or *poset*). An *upper bound* of a subset $S \subseteq X$ is an element $x \in X$ such that $(\forall s \in S)(s \leq x)$. A *least upper bound* (or *supremum*, *sup*) of S is an upper bound ℓ of S such that $\ell \leq x$ for every upper bound x of S . The sup of S is unique whenever it exists, and is usually denoted $\bigvee S$. The notion of *greatest lower bound* (or *infimum*, *inf*) is defined dually. A subset $D \subseteq X$ is *directed* if D is non-empty and for every $x, y \in D$, there is some $z \in D$ such that $x \leq z$ and $y \leq z$.

We say that (X, \leq) is a *dcpo* if every directed $D \subseteq X$ has a sup $\bigvee D \in X$. A *cpo* is a dcpo with a least element (usually denoted \perp). Note that each set A is a dcpo for the discrete order (in which x is comparable with y if, and only if, $x = y$). However, such a dcpo A is not a cpo unless A is a singleton.

Example 1 (Flat Domains). Given a set A , the *flat domain* $\llbracket A \rrbracket$ is the disjoint union $\{\perp\} + A$ equipped with the partial order $\leq_{\llbracket A \rrbracket}$, where $x \leq_{\llbracket A \rrbracket} y$ iff $x = y$ or $(x = \perp \text{ and } y \in A)$.

It is easy to see that $(\llbracket A \rrbracket, \leq_{\llbracket A \rrbracket})$ is a cpo whose directed subsets have at most one element from A . For instance, the domain $\llbracket \text{Bool} \rrbracket$ can be represented by the following Hasse diagram.



²Actually, as well as e.g. [NUKT18, SU23], despite a fundamentally different approach (see §6).

Scott-Continuous Functions. Let $X = (X, \leq_X)$ and $Y = (Y, \leq_Y)$ be dcpos. A function $f: X \rightarrow Y$ is *Scott-continuous* if f is monotone ($x \leq_X x'$ implies $f(x) \leq_Y f(x')$) and if moreover f preserves directed sups, in the sense that for each directed $D \subseteq X$, we have

$$f(\bigvee D) = \bigvee \{f(d) \mid d \in D\}$$

We write **CPO** (resp. **DCPO**) for the category with cpos (resp. dcpos) as objects and with Scott-continuous functions as morphisms. We say that $f \in \mathbf{CPO}[X, Y]$ is strict if $f(\perp_X) = \perp_Y$. A non-strict monotone map between flat domains is necessarily constant.

Given dcpos $X = (X, \leq_X)$ and $Y = (Y, \leq_Y)$, the set of Scott-continuous functions $\mathbf{DCPO}[X, Y]$ is itself a dcpo w.r.t. the *pointwise order*

$$f \leq_{\mathbf{DCPO}[X, Y]} g \quad \text{iff} \quad \forall x \in X, f(x) \leq_Y g(x)$$

If Y is actually a cpo, then $\mathbf{DCPO}[X, Y]$ is a cpo whose least element is the constant function $x \in X \mapsto \perp_Y \in Y$, where \perp_Y is the least element of Y .

Example 2 (Streams). Let A be a set. We let $\llbracket \text{Str}A \rrbracket$, the cpo of *streams over A* , be $\mathbf{DCPO}[\mathbb{N}, \llbracket A \rrbracket]$ with \mathbb{N} discrete. We unfold this important example. Since \mathbb{N} is discrete, $\llbracket \text{Str}A \rrbracket$ actually consists of the set $\llbracket A \rrbracket^\omega$ equipped with the partial order

$$x \leq_{\llbracket \text{Str}A \rrbracket} y \quad \text{iff} \quad \forall n \in \mathbb{N}, x(n) \leq_{\llbracket A \rrbracket} y(n)$$

A set $D \subseteq \llbracket \text{Str}A \rrbracket$ is directed if, and only if, D is non-empty and each $D(n) = \{x(n) \mid x \in D\}$ has at most one element from A . Then $\bigvee D \in \llbracket \text{Str}A \rrbracket$ takes $n \in \mathbb{N}$ to the largest element of $D(n)$. The least element of $\llbracket \text{Str}A \rrbracket$ is the stream \perp^ω of constant value $\perp \in \llbracket A \rrbracket$.

Note that $\llbracket \text{Str}A \rrbracket$ has “partially defined” elements. Besides the least element \perp^ω , we have e.g. the stream $u \cdot \perp^\omega$ (which agrees with $u \in A^*$ and then is \perp at all sufficiently large positions) or $(a \cdot \perp)^\omega$ (which is a at all even positions, and is \perp everywhere else). The ω -words on A are precisely those streams $x \in \llbracket \text{Str}A \rrbracket$ which never take the value \perp . Such streams are called *total*. Note that if x is total, then

$$x = \bigvee \{u \cdot \perp^\omega \mid u \in A^* \text{ and } u \subseteq x\}$$

Remark 1. The cpo $\llbracket \text{Str}A \rrbracket$ is the usual solution in the category **CPO** of the *domain equation*

$$X \cong \llbracket A \rrbracket \times X$$

(where $\llbracket A \rrbracket \times X$ is equipped with the pointwise order), see e.g. [AC98, Theorem 7.1.10 and Proposition 7.1.13]. In particular, the constructor $(- \ :: -)$ of the type $\text{Str}A$ is interpreted as the isomorphism taking $(a, x) \in \llbracket A \rrbracket \times \llbracket \text{Str}A \rrbracket$ to $a \cdot x \in \llbracket \text{Str}A \rrbracket$, with inverse $x \mapsto (x(0), x|1)$. Note that $\llbracket \text{Str}A \rrbracket$ differs from the usual *Kahn domain* $A^* \cup A^\omega$ (see e.g. [Vic89, Definition 3.7.5 and Example 5.4.4] or [DST19, §7.4], see also [VVK05]).

Remark 2. Each $f: X \rightarrow_{\mathbf{CPO}} X$ has a *least fixpoint* $Y(f) := \bigvee_{n \in \mathbb{N}} f^n(\perp) \in X$. In particular, `filter` is interpreted as the Scott-continuous function $\llbracket \text{filter} \rrbracket$ taking $p: \llbracket A \rrbracket \rightarrow_{\mathbf{CPO}} \llbracket \text{Bool} \rrbracket$ to the least fixpoint of the following function f_p , where X is the cpo $\llbracket \text{Str}A \rrbracket \rightarrow_{\mathbf{CPO}} \llbracket \text{Str}A \rrbracket$.

$$f_p := \lambda g. \lambda x. \text{if } p(x(0)) \text{ then } x(0) \cdot g(x|1) \text{ else } g(x|1) : X \rightarrow_{\mathbf{CPO}} X$$

Algebraicity. Among the many good properties of $\llbracket \text{Str}A \rrbracket$, algebraicity is the crucial one in this work. This property is not used right away, but will be the main assumption of various statements later on.

Let (X, \leq) be a dcpo. We say that $x \in X$ is *finite*³ if for every directed $D \subseteq X$ such that $x \leq \bigvee D$, there is some $d \in D$ such that $x \leq d$. We say that X is *algebraic* if for every $x \in X$, the set $\{d \in X \mid d \text{ finite and } d \leq x\}$ is directed and has sup x . Each discrete or flat dcpo is algebraic.

³Finite elements are called *compact* in [AC98].

Example 3 (Streams). The cpo $\llbracket \text{Str}A \rrbracket$ is algebraic, and its finite elements admit a particularly simple description. The *support* of $x \in \llbracket \text{Str}A \rrbracket$ is the set $\text{supp}(x)$ of “defined letters” of x :

$$\text{supp}(x) := \{n \in \mathbb{N} \mid x(n) \neq \perp\}$$

We say that a stream x has finite support when $\text{supp}(x)$ is a finite set. For instance, given a finite word $u \in A^*$ and $n \in \mathbb{N}$, the stream $\perp^n \cdot u \cdot \perp^\omega$ has finite support. On the other hand, total streams, as well as e.g. $(a \cdot \perp)^\omega$, do not have finite support.

For each $x \in \llbracket \text{Str}A \rrbracket$, the set $\{d \mid d \text{ of finite support and } d \leq_{\llbracket \text{Str}A \rrbracket} x\}$ is directed and has sup x . Moreover, the finite elements of $\llbracket \text{Str}A \rrbracket$ are exactly those of finite support.

2.2 Linear Temporal Logic (LTL)

Syntax and Semantics. Let A be a set. The formulae of $\text{LTL} = \text{LTL}(A)$ are given by

$$\Phi, \Psi ::= a \mid \text{True} \mid \text{False} \mid \Phi \wedge \Psi \mid \Phi \vee \Psi \mid \neg \Phi \mid \bigcirc \Phi \mid \Phi \text{ U } \Psi \mid \Phi \text{ W } \Psi$$

where $a \in A$. Hence, besides pure propositional logic, $\text{LTL}(A)$ has *atomic formulae* $a \in A$, and *modalities* $\bigcirc \Phi$ (read “next Φ ”), $\Phi \text{ U } \Psi$ (read “ Φ until Ψ ”) and $\Phi \text{ W } \Psi$ (read “ Φ weak until Ψ ” or “ Φ unless Ψ ”).

The LTL formulae over A are usually interpreted on ω -words over A , see e.g. [BK08, §5]. The interpretation of modalities actually implicitly relies on the bijection $A^\omega \cong A \times A^\omega$. We similarly rely on the isomorphism $\llbracket \text{Str}A \rrbracket \cong_{\text{CPO}} [A] \times \llbracket \text{Str}A \rrbracket$ for interpreting LTL(A) formulae in $\llbracket \text{Str}A \rrbracket$. We define $\llbracket \Phi \rrbracket \subseteq \llbracket \text{Str}A \rrbracket$ by induction on Φ . The propositional connectives of LTL are interpreted using the usual Boolean algebra structure of the powerset $\mathcal{P}(\llbracket \text{Str}A \rrbracket)$. For $a \in A$, we let $\llbracket a \rrbracket := \{x \in \llbracket \text{Str}A \rrbracket \mid x(0) = a\}$. The modalities are interpreted as follows.

$$\begin{aligned} \llbracket \bigcirc \Phi \rrbracket &:= \{x \in \llbracket \text{Str}A \rrbracket \mid x \upharpoonright 1 \in \llbracket \Phi \rrbracket\} \\ \llbracket \Phi \text{ U } \Psi \rrbracket &:= \{x \in \llbracket \text{Str}A \rrbracket \mid \exists i \in \mathbb{N}, x \upharpoonright 0, \dots, x \upharpoonright (i-1) \in \llbracket \Phi \rrbracket \text{ and } x \upharpoonright i \in \llbracket \Psi \rrbracket\} \\ \llbracket \Phi \text{ W } \Psi \rrbracket &:= \{x \in \llbracket \text{Str}A \rrbracket \mid \forall i \in \mathbb{N}, x \upharpoonright i \in \llbracket \Phi \rrbracket\} \cup \llbracket \Phi \text{ U } \Psi \rrbracket \end{aligned}$$

We say that $x \in \llbracket \text{Str}A \rrbracket$ *satisfies* a formula Φ (notation $x \Vdash \Phi$) when $x \in \llbracket \Phi \rrbracket$. It is often convenient to decompose $\llbracket \bigcirc \Phi \rrbracket$ as $\llbracket \bigcirc \rrbracket(\llbracket \Phi \rrbracket)$, where $\llbracket \bigcirc \rrbracket: \mathcal{P}(\llbracket \text{Str}A \rrbracket) \rightarrow \mathcal{P}(\llbracket \text{Str}A \rrbracket)$ takes S to $\{x \in \llbracket \text{Str}A \rrbracket \mid x \upharpoonright 1 \in S\}$. The modalities U and W may not be easy to grasp. Given LTL formulae Φ and Ψ , we let

$$\begin{aligned} \text{“eventually } \Psi \text{”} \quad \diamond \Psi &:= \text{True U } \Psi & (\llbracket \diamond \Psi \rrbracket = \{x \in \llbracket \text{Str}A \rrbracket \mid \exists i \in \mathbb{N}, x \upharpoonright i \in \llbracket \Psi \rrbracket\}) \\ \text{“always } \Phi \text{”} \quad \square \Phi &:= \Phi \text{ W False} & (\llbracket \square \Phi \rrbracket = \{x \in \llbracket \text{Str}A \rrbracket \mid \forall i \in \mathbb{N}, x \upharpoonright i \in \llbracket \Phi \rrbracket\}) \end{aligned}$$

Example 4. Consider a stream $x \in \llbracket \text{Str}A \rrbracket$.

- (1) We have $x \Vdash \bigcirc a$ if, and only if, $x(1) = a$. For instance, $\perp a \perp^\omega \Vdash \bigcirc a$ but $a \perp^\omega \not\Vdash \bigcirc a$.
- (2) We have $x \Vdash \diamond a$ if, and only if, $x(i) = a$ for some $i \in \mathbb{N}$. For instance, $\perp^n a \perp^\omega \Vdash \diamond a$ for every $n \in \mathbb{N}$. But $b^\omega \not\Vdash \diamond a$ if $b \neq a$.
- (3) We have $x \Vdash \square a$ if, and only if, $x = a^\omega$. If A is finite, then x is total iff $x \Vdash \square \bigvee_{a \in A} a$.
- (4) We have $x \Vdash \diamond \square a$ if, and only if, $x(i) = a$ for infinitely many $i \in \mathbb{N}$. E.g. $(\perp a)^\omega \Vdash \diamond \square a$.
- (5) We have $x \Vdash \diamond \square a$ if, and only if, $x(i) = a$ for “ultimately all $i \in \mathbb{N}$ ”. This means that for some $n \in \mathbb{N}$, we have $x(i) = a$ for all $i \geq n$. For instance, $\perp^n a^\omega \Vdash \diamond \square a$ for all $n \in \mathbb{N}$. But $(\perp a)^\omega \not\Vdash \diamond \square a$.

Say that Φ and Ψ are (logically) *equivalent*, notation $\Phi \equiv \Psi$, if $\llbracket \Phi \rrbracket = \llbracket \Psi \rrbracket$. LTL has many redundancies w.r.t. logical equivalence. Besides the usual De Morgan laws, we have e.g.

$$\neg \square \Phi \equiv \diamond \neg \Phi \quad \neg \diamond \Phi \equiv \square \neg \Phi \quad \Phi \text{ W } \Psi \equiv (\Phi \text{ U } \Psi) \vee \square \Phi$$

Remark 3. The modalities \mathbf{U} and \mathbf{W} are also “De Morgan” duals, in the following sense. Given Φ and Ψ , it is well-known that $\llbracket \Phi \mathbf{U} \Psi \rrbracket$ and $\llbracket \Phi \mathbf{W} \Psi \rrbracket$ are respectively the least and the greatest fixpoint of the (monotone) map on $\mathcal{P}(\llbracket \text{Str}A \rrbracket)$ taking S to $\llbracket \Psi \rrbracket \cup (\llbracket \Phi \rrbracket \cap \llbracket \mathbf{O} \rrbracket(S))$. See e.g. [BK08, Lemmas 5.18 and 5.19]. But $\mathcal{P}(\llbracket \text{Str}A \rrbracket)$ is a complete atomic Boolean algebra, and given a monotone endo-function f on such a Boolean algebra, the least and greatest fixpoints of f are related by $\text{lfp}(f) = \neg \text{gfp}(b \mapsto \neg f(\neg b))$ and $\text{gfp}(f) = \neg \text{lfp}(b \mapsto \neg f(\neg b))$.

Negation-Free LTL. Our main positive results only hold for the negation-free fragment of LTL. An LTL formula is *negation-free* (*n.-f.*) if it contains no negation ($\neg(-)$). Hence, the negation-free formulae of LTL are generated by the above grammar for LTL, but without the production $\neg\Phi$.

Example 5. All formulae of Example 4 are negation-free. Moreover, the negation-free fragment is closed under $\Box(-)$ and $\Diamond(-)$.

Assume A is finite. For any $S \subseteq A$, there is a negation-free formula Ψ_S such that $x \Vdash \Psi_S$ iff $x(0) \in S$. It follows that for any Scott-continuous $p: \llbracket A \rrbracket \rightarrow \llbracket \text{Bool} \rrbracket$, there is a negation-free formula Ψ_p such that $x \Vdash \Box \Diamond \Psi_p$ if, and only if, x has infinitely many elements satisfying p .

Most redundancies of LTL mentioned above disappear in the negation-free fragment. This is why we have chosen this set of connectives from the start. In negation-free $\text{LTL}(A)$, all connectives have a De Morgan dual. But negated atomic formulae ($\neg a$ for $a \in A$) are not available. Hence, in contrast with positive normal forms (see e.g. [BK08, Definition 5.20]), negation is *not* definable in negation-free LTL. This positive character is reflected in the following fundamental fact, proved by induction on formulae.

Lemma 1. *If Φ is n.-f. then $\llbracket \Phi \rrbracket$ is upward-closed (if $x \in \llbracket \Phi \rrbracket$ and $x \leq_{\llbracket \text{Str}A \rrbracket} y$ then $y \in \llbracket \Phi \rrbracket$).*

Corollary 1. *Let Φ be negation-free. Then $\llbracket \Phi \rrbracket$ is closed in $\llbracket \text{Str}A \rrbracket$ under directed sups. Moreover, the inclusion $\llbracket \Phi \rrbracket \hookrightarrow \llbracket \text{Str}A \rrbracket$ is a Scott-continuous order-embedding.*

Hence, $\llbracket \Phi \rrbracket$ is a sub-dcpo of $\llbracket \text{Str}A \rrbracket$ when Φ is n.-f. But this may not give much information on $\llbracket \Phi \rrbracket$. For instance, $A^\omega = \llbracket \Box \bigvee_{a \in A} a \rrbracket$ (Example 4(3), A finite) is a discrete dcpo. Building on Lemma 1, we are going to exhibit much more structure on such inclusions $\llbracket \Phi \rrbracket \hookrightarrow \llbracket \text{Str}A \rrbracket$. But before, we note that Lemma 1 and Corollary 1 may fail in presence of negation.

Example 6. Consider the formula $\neg \Box a$. Note that $a^\omega \not\Vdash \neg \Box a$. But for every finite $d \leq_{\llbracket \text{Str}A \rrbracket} a^\omega$, we have $d \Vdash \neg \Box a$. Hence $\llbracket \neg \Box a \rrbracket$ is not upward-closed. Moreover, $\{d \text{ finite} \mid d \leq_{\llbracket \text{Str}A \rrbracket} a^\omega\}$ is a directed subset of $\llbracket \neg \Box a \rrbracket$ which has no sup in $\llbracket \neg \Box a \rrbracket$. Hence $\llbracket \neg \Box a \rrbracket$ is not a dcpo w.r.t. the restriction of $\leq_{\llbracket \text{Str}A \rrbracket}$.

3 The Topological Approach

We shall now look at inclusions $\llbracket \Phi \rrbracket \hookrightarrow \llbracket \text{Str}A \rrbracket$ from a topological perspective. We recall in §3.1 that the categories $(\mathbf{D})\mathbf{CPO}$ can be embedded in the category \mathbf{Top} of topological spaces and continuous functions. The highlight is that \mathbf{Top} has a much richer notion of substructures (called subspaces) than $(\mathbf{D})\mathbf{CPO}$.

Actually, when looking at (d)cpos as topological spaces, the notion of sobriety from point-free (or “element-free”) topology comes to the front. Ample mathematical justifications for the importance of sober spaces in domain theory are gathered in [GL13]. We shall content ourselves with more informal motivations in §3.2. In §3.3, we abstractly prove that $\llbracket \Phi \rrbracket$ induces a sober subspace of $\llbracket \text{Str}A \rrbracket$ when Φ is negation-free. This will be refined to concrete representations in §4 (and also [RS23, §5]), using geometric logic.

3.1 Topological Spaces

A *topological space* is a pair $(X, \Omega(X))$ of a set X and a collection $\Omega = \Omega(X)$ of subsets of X , called *open sets*. Ω is called a *topology* on X , and is asked to be stable under arbitrary

unions and under finite intersections. In particular, \emptyset and X are open in X (respectively as the empty union and the empty intersection).

A set $C \subseteq X$ is *closed* if its complement $X \setminus C$ is open. Closed sets are stable under finite unions and arbitrary intersections. Hence, any $S \subseteq X$ is contained in a least closed set $\bar{S} \subseteq X$. Each space (X, Ω) is equipped with a *specialization* (pre)order \leq_Ω on X , defined as

$$x \leq_\Omega y \quad \text{iff} \quad (\forall U \in \Omega)(x \in U \implies y \in U)$$

Given $x \in X$, we have $\overline{\{x\}} = \downarrow x := \{y \in X \mid y \leq_\Omega x\}$ (see e.g. [GL13, Lemma 4.2.7]). A topology Ω is T_0 when \leq_Ω is a partial order (see e.g. [GL13, Proposition 4.2.3]).

Given spaces $(X, \Omega(X))$ and $(Y, \Omega(Y))$, a function $f: X \rightarrow Y$ is *continuous* when its inverse image $f^{-1}: \mathcal{P}(Y) \rightarrow \mathcal{P}(X)$ restricts to a function $\Omega(Y) \rightarrow \Omega(X)$, i.e. when $f^{-1}(V) \in \Omega(X)$ for all $V \in \Omega(Y)$. We write **Top** for the category of topological spaces and continuous functions. An *homeomorphism* is an isomorphism in **Top**.

The Scott Topology. The following is well-known. See e.g. [AC98, §1.2] or [GL13, §4].

Let (X, \leq_X) be a dcpo. A subset $U \subseteq X$ is *Scott-open* if U is upward-closed, and if moreover U is inaccessible by directed sups, in the sense that if $\bigvee D \in U$ with $D \subseteq X$ directed, then $D \cap U \neq \emptyset$. This equips X with a T_0 topology, called the *Scott topology*, whose specialization order coincides with \leq_X . Note that $C \subseteq X$ is Scott-closed precisely when C is downward-closed and stable under directed sups.

Example 7. When (X, \leq) is algebraic, the sets $\uparrow d := \{x \in X \mid d \leq x\}$ with d finite form a sub-basis for the Scott topology. For instance, the Scott-open subsets of $\llbracket \text{Str}A \rrbracket$ are arbitrary unions of sets of the form

$$\uparrow d = \{x \in \llbracket \text{Str}A \rrbracket \mid \forall i \in \text{supp}(d), x(i) = d(i)\}$$

with $\text{supp}(d)$ finite. In particular, given $x \in U$ with $U \subseteq \llbracket \text{Str}A \rrbracket$ Scott-open, there is a *finite* set $\{i_1, \dots, i_k\} \subseteq \mathbb{N}$ such that $\{y \mid y(i_1) = x(i_1), \dots, y(i_k) = x(i_k)\} \subseteq U$.

Beware that $\llbracket \Phi \rrbracket$ may not be an open nor a closed subset of $\llbracket \text{Str}A \rrbracket$, even when Φ is negation-free. Consider for instance $A^\omega = \llbracket \square \bigvee_{a \in A} a \rrbracket$ (with A non-empty and finite), which contains a^ω but no finite $d \leq_{\llbracket \text{Str}A \rrbracket} a^\omega$.

A function $f: X \rightarrow Y$ between dcpos is Scott-continuous precisely when f is continuous w.r.t. the Scott-topologies on X and Y . It follows that **DCPO** and **CPO** are full subcategories of **Top**. From now on, we shall mostly look at **(D)CPO** in this way. Unless stated otherwise, dcpos will always be equipped with their Scott topology.

Subspaces. Our motivation for moving from **(D)CPO** to **Top** is that **Top** has a rich notion of subspace. We refer to [BBT20, §1.2]. Given a space (X, Ω) and a subset $P \subseteq X$, the *subspace topology* on P is

$$\Omega \upharpoonright P := \{U \cap P \mid U \in \Omega\}$$

The subspace topology on P makes the inclusion function $\iota: P \hookrightarrow X$ continuous. It is the “best possible” topology on P in the following sense: given a space $(Y, \Omega(Y))$, a function $f: Y \rightarrow P$ is continuous if, and only if, the composition $\iota \circ f: Y \rightarrow X$ is continuous.

$$\begin{array}{ccc} & P & \\ f \nearrow & & \searrow \iota \\ Y & \xrightarrow{\quad \iota \circ f \quad} & X \end{array}$$

Example 8. Generalizing Example 4(3) and Lemma 1, A^ω is a discrete sub-dcpo of $\llbracket \text{Str}A \rrbracket$.

On the other hand, the subspace topology $\Omega(\llbracket \text{Str}A \rrbracket) \upharpoonright A^\omega$ is the usual *product topology* on A^ω (see e.g. [PP04, §III] or [Kec95]). The sets of the form $A^\omega \cap \uparrow d$ (with d finite) form a sub-basis for the subspace topology. In fact, its opens are unions of sets of the form $\{\sigma \in A^\omega \mid \sigma(i_1) = a_1, \dots, \sigma(i_k) = a_k\}$, where $i_1, \dots, i_k \in \mathbb{N}$ and $a_1, \dots, a_k \in A$ with $k \geq 0$.

3.2 The Element-Free Setting

The topological setting comes with an intrinsic notion of approximation.

Consider for instance ω -words $\sigma \in A^\omega$ (Example 8). Similarly as with streams in Example 7, given an ω -word σ and an open U , if σ belongs to U , then this fact is witnessed by the knowledge of a finite number of elements of σ .⁴ We view the opens U such that $\sigma \in U$ as approximations of σ .

Given a space (X, Ω) , we are interested in describing the elements of X by their approximations, represented as suitable sets of opens $\mathcal{F} \subseteq \Omega$. This is the realm of *element-free* (or *point-free*) *topology*. Its central objects, called *frames* (or *locales*), abstract away from the elements of spaces, and only retain the lattice structure of open sets. Besides [GL13], we refer to [Joh82, Joh83, Vic89, PP12, PP21].

Frames. A *complete lattice* is a poset having all sups and all infs. But recall (from e.g. [DP02, Theorem 2.31]) that a poset has all sups if, and only if, it has all infs. Hence, we can see complete lattices indifferently as posets with all sups or as posets with all infs.

A *frame* is a poset L with all sups (and thus all infs), and which satisfies the following *frame distributive law*: for all $S \subseteq L$ and all $a \in L$,

$$a \wedge \bigvee S = \bigvee \{a \wedge s \mid s \in S\}$$

Not every complete lattice is a frame.⁵ But every $(\mathcal{P}(X), \subseteq)$ is a frame, and so is the two-elements poset $\mathbf{2} := \{0 \leq 1\}$.

Given frames L and K , a *frame morphism* $f: L \rightarrow K$ is a function which preserves all sups and all *finite* infs. Note that frame morphisms are automatically monotone. We write **Frm** for the category of frames and frame morphisms.

Example 9. Let $(X, \Omega(X))$ be a space. Then $\Omega(X)$ has all sups and they are given by unions. Hence $\Omega(X)$ is a complete lattice. Beware that the inf in $\Omega(X)$ of an arbitrary $S \subseteq \Omega(X)$ is in general not its intersection $\bigcap S$, but the *interior* of $\bigcap S$ (the largest open set contained in $\bigcap S$). However, *finite* infs in $\Omega(X)$ are given by intersections, and $\Omega(X)$ is a frame. In fact, the topologies on a given set X correspond exactly to the sub-frames of $\mathcal{P}(X)$.

Moreover, if $f: X \rightarrow Y$ is continuous, then its inverse image f^{-1} restricts to a function $\Omega(Y) \rightarrow \Omega(X)$. This function is actually a frame morphism $\Omega(f) \in \mathbf{Frm}[\Omega(Y), \Omega(X)]$. In other words, the operation $(X, \Omega(X)) \mapsto \Omega(X)$ extends to a functor Ω from the category **Top** to **Frm**^{op}, the opposite of **Frm**. The category **Frm**^{op} is the category of *locales*.

The Space of Points. We see a frame L as a collection of formal approximations. Suitable subsets of L describe “converging” sets of formal approximations, and constitute the elements of a space, the space of points of L . The idea is as follows. Given a space X and $x \in X$, let

$$\mathcal{F}_x := \{U \in \Omega(X) \mid x \in U\}$$

Note the following properties of \mathcal{F}_x w.r.t. the frame structure of $\Omega(X)$. First, \mathcal{F}_x is stable under finite intersections ($x \in X$, and $x \in U \cap V$ iff $x \in U$ and $x \in V$). Second, given $S \subseteq \Omega(X)$ with $\bigcup S \in \mathcal{F}_x$, we have $U \in \mathcal{F}_x$ for some $U \in S$ (if $x \in \bigcup S$ then $x \in U$ for some $U \in S$). Hence, the characteristic function of $\mathcal{F}_x \subseteq \Omega(X)$ is a frame morphism $\Omega(X) \rightarrow \mathbf{2}$.

A *point* of a frame L is an element of $\text{pt}(L) := \mathbf{Frm}[L, \mathbf{2}]$. We shall always identify a point $\mathcal{F} \in \mathbf{Frm}[L, \mathbf{2}]$ with the set $\{a \in L \mid \mathcal{F}(a) = 1\}$. Given $a \in L$, let

$$\text{ext}(a) := \{\mathcal{F} \in \text{pt}(L) \mid a \in \mathcal{F}\}$$

The function $\text{ext}: L \rightarrow \mathcal{P}(\text{pt}(L))$ is a frame morphism (see e.g. [Joh82, Lemma II.1.6]). In particular, its image is a sub-frame of $\mathcal{P}(\text{pt}(L))$, and is thus a topology $\Omega(\text{pt}(L))$ on $\text{pt}(L)$. The *space of points* of L is $(\text{pt}(L), \Omega(\text{pt}(L)))$.

⁴Dually, the knowledge of the whole ω -word σ may be needed to testify that $\sigma \notin U$.

⁵Consider e.g. a finite (and thus complete) non-distributive lattice, see [DP02, Example 4.6(6)].

The operation $L \mapsto \text{pt}(L)$ extends to a functor $\text{pt}: \mathbf{Frm}^{\text{op}} \rightarrow \mathbf{Top}$ which is right adjoint to Ω (see e.g. [Joh82, Theorem II.1.4]). The action of $\text{pt}: \mathbf{Frm}^{\text{op}} \rightarrow \mathbf{Top}$ on a frame morphism $f: L \rightarrow K$ is the continuous function $\text{pt}(K) \rightarrow \text{pt}(L)$ which takes $\mathcal{F} \in \mathbf{Frm}[K, \mathbf{2}]$ to $\mathcal{F} \circ f \in \mathbf{Frm}[L, \mathbf{2}]$. The unit at $X \in \mathbf{Top}$ of the adjunction $\Omega \dashv \text{pt}$ is the continuous function $\eta_X: X \rightarrow \text{pt}(\Omega(X))$ taking x to \mathcal{F}_x (see e.g. [Joh82, §II.1.6]).

Sober Spaces. The function $\eta_X: X \rightarrow \text{pt}(\Omega(X))$ continuously maps the space X to its space of “converging formal approximations” $\text{pt}(\Omega(X))$. But $\text{pt}(\Omega(X))$ may not correctly represent X . A space X is *sober* if η_X is a bijection (in which case η_X is automatically an homeomorphism, see [Joh82, §II.1.6]).

Given a frame L , the space of points $\text{pt}(L)$ is always sober ([Joh82, Lemma II.1.7]). Hence (by functoriality of Ω and pt), if X is homeomorphic to $\text{pt}(L)$, then X is sober as well. It follows from [Joh82, Lemma II.1.6(ii)] that A^ω is sober for its product topology.⁶ But not every dcpo is sober ([Joh82, II.1.9]). For the following, see e.g. [Joh82, Theorem VII.2.6].

Proposition 1. *Algebraic dcpos are sober.*

Remark 4. In fact, a sober space is always T_0 , and is moreover a dcpo w.r.t. its specialization order ([Joh82, Lemmas II.1.6(i) and II.1.9]). This provides a functor $\mathbf{Frm}^{\text{op}} \rightarrow \mathbf{DCPO}$ which is actually right adjoint to the composite $\mathbf{DCPO} \hookrightarrow \mathbf{Top} \rightarrow \mathbf{Frm}^{\text{op}}$, yielding the *Scott adjunction* of [DL22].

In particular, a sober dcpo is completely determined by the specialization order of its space of points. On the other hand, beware that the composite $\mathbf{Frm}^{\text{op}} \rightarrow \mathbf{DCPO} \rightarrow \mathbf{Frm}^{\text{op}}$ may loose a lot of structure (e.g. it takes the product topology on A^ω to the discrete (Scott) topology).

3.3 Sobriety of Subspaces

Proposition 1 and Remark 4 imply that for an algebraic dcpo X , the topological notion of approximation coincides with the domain-theoretic one. But we are interested in subspaces of the algebraic cpo $\llbracket \text{Str}A \rrbracket$. Discussing the sobriety of such subspaces involves going further into the point-free setting. While the main results of this §3.3 are important for this paper, the technical developments are used again only in [RS23, §5].

Let (X, Ω) be a space, and consider some $P \subseteq X$. The subspace inclusion $\iota: (P, \Omega \upharpoonright P) \hookrightarrow (X, \Omega)$ induces the surjective frame morphism $\iota^* := \Omega(\iota): \Omega \rightarrow \Omega \upharpoonright P$ which takes $U \in \Omega$ to $(U \cap P) \in \Omega \upharpoonright P$. The following is a handy reformulation of sobriety for $(P, \Omega \upharpoonright P)$.

Lemma 2. *Assume that (X, Ω) is sober. Then the following are equivalent.*

- (i) $(P, \Omega \upharpoonright P)$ is sober.
- (ii) For each $x \in X$, we have $x \in P$ if, and only if, $\mathcal{F}_x = \mathcal{G} \circ \iota^*$ for some $\mathcal{G} \in \text{pt}(\Omega \upharpoonright P)$.

$$\begin{array}{ccc} \Omega & \xrightarrow{\iota^*} & \Omega \upharpoonright P \\ & \searrow \mathcal{F}_x & \swarrow \mathcal{G} \\ & \mathbf{2} & \end{array}$$

Let L be a frame. A *quotient frame* of L is an isomorphism-class of surjective frame morphisms $L \twoheadrightarrow K$. We are going to discuss an abstract but mathematically powerful representation of the quotient frame $\Omega \twoheadrightarrow \Omega \upharpoonright P$. We use tools from [Joh82, §II.2] and [PP12, §VI.1] on the dual notion of *sub-locale*.

Everything starts from Galois connections and related adjointness properties, for which we refer to [DP02, 7.23–7.34]. Fix a frame morphism $f: L \rightarrow K$. Since $f: L \rightarrow K$ preserves all sups, it has an *upper adjoint* $f_*: K \rightarrow L$. This means that for all $a \in L$ and all $b \in K$,

$$f(a) \leq_K b \quad \text{if, and only if,} \quad a \leq_L f_*(b)$$

⁶Actually, [Joh82, Lemma II.1.6(ii)] states that each T_2 space is sober.

The pair (f, f_*) thus forms a Galois connection, and f_* is (uniquely) determined by

$$f_*(b) = \bigvee_L \{a \mid f(a) \leq_K b\}$$

The function f_* is in general not a frame morphism, but it always preserves all infs.

The composition $j := f_* \circ f: L \rightarrow L$ is a *nucleus* in the sense of [Joh82, §II.2.2]: we have (i) $j(a \wedge a') = j(a) \wedge j(a')$, (ii) $a \leq j(a)$ and (iii), $j(j(a)) \leq j(a)$. Nuclei are monotone and idempotent. If $j: L \rightarrow L$ is a nucleus, then the set $L_j := \{a \in L \mid j(a) = a\}$ of *j-fixpoints* is a frame and $j: L \rightarrow L_j$ is a frame morphism ([Joh82, Lemma II.2.2]). Note that the finite infs in L_j are those of L . But the sup of $S \subseteq L_j$ in L_j is $j(\bigvee_L S)$.

Consider now a subspace inclusion $\iota: (P, \Omega \upharpoonright P) \hookrightarrow (X, \Omega)$. Following [PP12, §VI.1.1], we write \tilde{P} for the frame of *j-fixpoints*, where $j := \iota_* \circ \iota^*$ and ι_* is the upper adjoint of ι^* .

$$\Omega \xrightarrow{\iota^*} \Omega \upharpoonright P \xrightarrow{\iota_*} \Omega$$

We rely on the following description of the nucleus j (from which [PP12, §VI.1.1] gives an explicit representation of *j-fixpoints* that we shall not use directly).

Remark 5. Given an open $U \in \Omega$ of X , we have

$$j(U) = \bigcup \{V \in \Omega \mid V \cap P = U \cap P\}$$

The proof of [Joh82, Theorem II.2.3] gives Lemma 3 below. Recall that order-isomorphisms preserve all existing sups and infs ([DP02, Lemma 2.27(ii)]).

Lemma 3. *The function $\iota_*: \Omega \upharpoonright P \rightarrow \Omega$ co-restricts to a frame isomorphism $\iota_*: \Omega \upharpoonright P \rightarrow \tilde{P}$.*

We use $j = \iota_* \iota^*: \Omega \rightarrow \tilde{P}$ to represent the quotient frame induced by the subspace inclusion $\iota: (P, \Omega \upharpoonright P) \hookrightarrow (X, \Omega)$. The frame \tilde{P} turns out to be a good tool for studying the sobriety of $(P, \Omega \upharpoonright P)$. Following [PP12, VI.1.3], given $x \in X$ we let $\tilde{x} := X \setminus \overline{\{x\}} = X \setminus \downarrow x$. We shall now see that it is useful to characterize when $\tilde{x} \in \tilde{P}$ (i.e. when $j(\tilde{x}) = \tilde{x}$).

Remark 6. Given $x \in X$ and $U \in \Omega$, we have $U \subseteq \tilde{x}$ if, and only if, $x \notin U$.

Lemma 4. *Let $x \in X$ and $\tilde{F}_x := \{U \in \tilde{P} \mid x \in U\}$. Then $\tilde{F}_x \in \text{pt}(\tilde{P})$ if and only if $\tilde{x} \in \tilde{P}$.*

Proposition 2. *Let $P \subseteq X$ with (X, Ω) sober. Then the following are equivalent.*

- (i) $(P, \Omega \upharpoonright P)$ is sober.
- (ii) For each $x \in X$, we have $x \in P$ if, and only if, $\tilde{x} \in \tilde{P}$.

In condition (ii) above, we actually always have $\tilde{x} \in \tilde{P}$ when $x \in P$ (see [PP12, VI.1.3.1]).

Proposition 2 will yield a general sufficient condition for the sobriety of $(P, \Omega \upharpoonright P)$ (Theorem 1 below), from which we will obtain the case of negation-free LTL (Corollary 3).

One further step into the point-free setting gives us sharper results. A space (X, Ω) is T_D when for each $x \in X$, there is some open $U \in \Omega$ such that $x \in U$ and $(U \setminus \{x\}) \in \Omega$. See [PP12, §I.2]. It is shown in [PP12, Proposition VI.1.3.1] that if X is a (possibly not sober) T_D space, then condition (ii) of Proposition 2 holds for *any* $P \subseteq X$. It follows that if X is sober *and* T_D , then each $P \subseteq X$ induces a sober subspace.

Consider now the case of a sober space (X, Ω) which is not T_D . Hence, there is some $x \in X$ such that for all open U with $x \in U$, the set $U \setminus \{x\}$ is not open.

Lemma 5. *Let $x \in X$ as above and set $P := X \setminus \{x\}$. Then $(P, \Omega \upharpoonright P)$ is not sober.*

It follows from [PP12, §I.2.1] that A^ω is T_D for the product topology.⁷ But $[\text{Str}A]$ is not T_D , unless $A = \emptyset$. Consider $a^\omega \in [\text{Str}A]$. Then any Scott-open U containing a^ω contains also some finite $d \leq [\text{Str}A] a^\omega$. Hence $U \setminus \{a^\omega\}$ is not upward-closed and thus not Scott-open.

⁷Actually, each T_1 space is T_D ([PP12, §I.2.1]).

Corollary 2. $\llbracket \neg \Box a \rrbracket = \llbracket \text{Str}A \rrbracket \setminus \{a^\omega\}$ is not a sober subspace of $\llbracket \text{Str}A \rrbracket$.

Let (X, Ω) be sober, and let $P \subseteq X$ be upward-closed for \leq_Ω . Assume $x \notin P$. Then $P \setminus \downarrow x = P$, and thus $P \cap \tilde{x} = P$. Hence $j(\tilde{x}) = \bigcup \{V \in \Omega \mid V \cap P = P\}$, so $j(\tilde{x}) = X$ and $\tilde{x} \notin \tilde{P}$. It follows that $x \in P$ precisely when $\tilde{x} \in \tilde{P}$, and Proposition 2 gives the following.

Theorem 1. If (X, Ω) is sober and $P \subseteq X$ is upward-closed for \leq_Ω , then $(P, \Omega \upharpoonright P)$ is sober.

Corollary 3. If (X, Ω) is a sober dcpo and if $P \subseteq X$ is upward-closed, then $(P, \Omega \upharpoonright P)$ is sober. In particular, if Φ is negation-free, then $\llbracket \Phi \rrbracket$ is a sober subspace of $\llbracket \text{Str}A \rrbracket$.

The importance we give to the negation-free fragment of LTL ultimately rests on Corollaries 2 and 3. But the frame \tilde{P} seems too abstract to be used concretely.

4 Geometric Logic

Geometric logic is an infinitary propositional logic which describes frames. Very roughly, the idea is that if a theory Th in geometric logic represents a frame L , then the models of Th can be organized in a space which is homeomorphic to $\text{pt}(L)$, the space of points of L .

We will represent the (sober) space $\llbracket \text{Str}A \rrbracket$ by a theory $\text{T}[\llbracket \text{Str}A \rrbracket]$ in geometric logic. Further, for each negation-free formula Φ of LTL, we shall (inductively) devise a theory $\text{T}[\llbracket \Phi \rrbracket]$ such that $\text{T}[\llbracket \text{Str}A \rrbracket] \cup \text{T}[\llbracket \Phi \rrbracket]$ represents the (sober) subspace induced by $\llbracket \Phi \rrbracket \leftrightarrow \llbracket \text{Str}A \rrbracket$. This will provide a concrete presentation of the corresponding quotient frame.

Our approach to geometric logic here is not the usual one, as presented in e.g. [Vic07] (see also [Joh02, §D]). The relations between the two approaches are discussed in [RS23, §5].

4.1 Geometric Theories

Formulae and Valuations. Let At be a set of *atomic propositions*. The *conjunctive* and the *geometric* formulae over At are respectively defined as

$$\gamma, \gamma' \in \text{Conj}(At) ::= p \mid \text{true} \mid \gamma \wedge \gamma' \quad \text{and} \quad \varphi, \psi, \theta \in \text{Geom}(At) ::= \bigvee S$$

where $p \in At$ and $S \subseteq \text{Conj}(At)$. A *valuation* of At is a function $\nu: At \rightarrow \mathbf{2}$. Given $\chi \in \text{Conj}(At) \cup \text{Geom}(At)$, the *satisfaction relation* $\nu \models \chi$ is defined by

$$\begin{array}{ll} \nu \models \text{true} & \nu \models \gamma \wedge \gamma' \quad \text{iff} \quad \nu \models \gamma \text{ and } \nu \models \gamma' \\ \nu \models p & \text{iff } \nu(p) = 1 \quad \nu \models \bigvee S \quad \text{iff} \quad \text{there exists } \gamma \in S \text{ such that } \nu \models \gamma \end{array}$$

We let **false** be the geometric formula $\bigvee \emptyset$. We may write γ for the geometric formula $\bigvee \{\gamma\}$. Given conjunctive formulae $(\gamma_i \mid i \in I)$, we write $\bigvee_{i \in I} \gamma_i$ for the geometric formula $\bigvee \{\gamma_i \mid i \in I\}$. Note that $\bigvee_{i \in I} \gamma_i = \bigvee_{j \in J} \gamma'_j$ if there is a bijection $f: I \rightarrow J$ with $\gamma_i = \gamma'_{f(i)}$.

Remark 7. There is no primitive notion of conjunction or disjunction on geometric formulae, but they can be defined. Given $(\varphi_i \mid i \in I)$ with $\varphi_i = \bigvee \{\gamma_{i,j} \mid j \in J_i\}$, we define $\bigvee_{i \in I} \varphi_i$ to be the geometric formula $\bigvee \{\gamma_{i,j} \mid i \in I \text{ and } j \in J_i\}$. We then have $\nu \models \bigvee_{i \in I} \varphi_i$ iff $\nu \models \varphi_i$ for some $i \in I$.

Similarly, given $\varphi = \bigvee_{i \in I} \gamma_i$ and $\psi = \bigvee_{j \in J} \gamma'_j$, we define $\varphi \wedge \psi := \bigvee \{\gamma_i \wedge \gamma'_j \mid (i, j) \in I \times J\}$. Then $\nu \models \varphi \wedge \psi$ iff $\nu \models \varphi$ and $\nu \models \psi$.

Sequents and Theories. A *sequent* over At is a pair $\psi \vdash \varphi$ of geometric formulae $\varphi, \psi \in \text{Geom}(At)$. A valuation ν of At is a *model* of $\psi \vdash \varphi$ if $\nu \models \psi$ implies $\nu \models \varphi$. Note that ν is a model of the sequent $\varphi \vdash \text{false}$ if, and only if, $\nu \not\models \varphi$.

A *geometric theory* over At is a set T of sequents over At . A valuation ν of At is a *model* of T if ν is a model of all the sequents of T . We write $\mathbf{Mod}(\text{T})$ for the set of models of T .

An *antecedent-free sequent* has the form $\text{true} \vdash \varphi$, and is denoted $\vdash \varphi$ (or even φ) for short. An *antecedent-free theory* consists of antecedent-free sequents only.

Algebraic Dcpo. Algebraic dcpo have a natural representation by geometric theories. This relies on the following well-known facts, for which we refer to [AC98, §1.1].

An *ideal* on a poset (P, \leq) is a subset $J \subseteq P$ which is downward-closed and directed. The set $\text{Idl}(P)$ of ideals on P is an algebraic dcpo for inclusion. The operation $P \mapsto \text{Idl}(P)$ is left adjoint to the forgetful functor from **DCPO** to the category of posets and monotone maps.

Let X be a dcpo, and let $\text{Fin}(X)$ be its sub-poset of finite elements. Given an ideal $J \in \text{Idl}(\text{Fin}(X))$, since J is directed we have $\bigvee J \in X$. For the following, see e.g. [AC98, Proof of Proposition 1.1.21(2)].

Lemma 6. *Let X be an algebraic dcpo. The function $J \in \text{Idl}(\text{Fin}(X)) \mapsto \bigvee J \in X$ is an order-isomorphism. Its inverse $X \rightarrow \text{Idl}(\text{Fin}(X))$ takes x to $\{d \in \text{Fin}(X) \mid d \leq x\}$.*

We represent an algebraic dcpo $X = (X, \leq_X)$ by a geometric theory $\mathbf{T}(X)$ over $\text{At} = \text{Fin}(X)$. The theory $\mathbf{T}(X)$ consists of $\vdash \bigvee \text{Fin}(X)$ together with all the sequents

$$d \vdash d' \quad (\text{if } d' \leq_X d) \qquad d \wedge d' \vdash \bigvee \{d'' \in \text{Fin}(X) \mid d \leq_X d'' \text{ and } d' \leq_X d''\}$$

where $d, d' \in \text{Fin}(X)$. Note that $J \subseteq \text{Fin}(X)$ is an ideal if, and only if, its characteristic function $\text{Fin}(X) \rightarrow \mathbf{2}$ is a model of $\mathbf{T}(X)$. Combined with Lemma 6, this yields Proposition 3 below. If $x \in X$, let $\nu(x): \text{At} \rightarrow \mathbf{2}$ be the characteristic function of $\{d \in \text{Fin}(X) \mid d \leq_X x\}$.

Proposition 3. *The map $x \mapsto \nu(x)$ is a bijection $X \rightarrow \mathbf{Mod}(\mathbf{T}(X))$.*

When X is actually an algebraic cpo, let $\mathbf{T}_\perp(X)$ be the theory obtained from $\mathbf{T}(X)$ by replacing the antecedent-free sequent $\vdash \bigvee \text{Fin}(X)$ with $\vdash \perp_X$ (where $\perp_X \in \text{Fin}(X)$ is the least element of X). The theories $\mathbf{T}(X)$ and $\mathbf{T}_\perp(X)$ have exactly the same models. Hence Proposition 3 also holds with $\mathbf{T}_\perp(X)$ in place of $\mathbf{T}(X)$.

Example 10 (Streams). We further simplify the theory representing the cpo $\llbracket \text{Str}A \rrbracket$.

Consider $x, y \in \llbracket \text{Str}A \rrbracket$ such that $\uparrow x \cap \uparrow y \neq \emptyset$ (i.e. such that $x, y \leq_{\llbracket \text{Str}A \rrbracket} z$ for some $z \in \llbracket \text{Str}A \rrbracket$). Then, since $\llbracket A \rrbracket$ is a flat cpo, we have $x(n) = y(n)$ for all $n \in \text{supp}(x) \cap \text{supp}(y)$. It follows that the set $\{x, y\}$ has a sup $x \vee_{\llbracket \text{Str}A \rrbracket} y$ in $\llbracket \text{Str}A \rrbracket$. Note that $x \vee_{\llbracket \text{Str}A \rrbracket} y$ is finite whenever so are x and y .

We can thus represent $\llbracket \text{Str}A \rrbracket$ with the following theory $\mathbf{T}[\llbracket \text{Str}A \rrbracket]$, where $d, d' \in \text{Fin}(\llbracket \text{Str}A \rrbracket)$.

$$\begin{array}{ll} d \vdash d' & (\text{if } d' \leq_{\llbracket \text{Str}A \rrbracket} d) \\ \vdash \perp^\omega & \end{array} \qquad \begin{array}{ll} d \wedge d' \vdash \mathbf{false} & (\text{if } \uparrow d \cap \uparrow d' = \emptyset) \\ d \wedge d' \vdash d \vee_{\llbracket \text{Str}A \rrbracket} d' & (\text{if } \uparrow d \cap \uparrow d' \neq \emptyset) \end{array}$$

Remark 8. Note that the theory $\mathbf{T}[\llbracket \text{Str}A \rrbracket]$ of Example 10 only involves finite geometric formulae. Actually, this amounts to the fact that $\llbracket \text{Str}A \rrbracket$ is *spectral* in its Scott topology (see [GvG23, Corollary 7.48 and Definition 6.2]).⁸ Roughly, being a spectral space means that the topology can be generated from a distributive lattice (as opposed to a frame). See e.g. [GvG23, Proposition 3.26 and Theorem 6.1] for details.

Spectral cpos include those known as ‘‘SFP’’ domains (see e.g. [Abr91, §2.2]). SFP domains are also called ‘‘bifinite’’ domains (see e.g. [AC98, Definition 5.2.2 and Theorem 5.2.7]). They are stable under most common domain operations and have solutions for recursive type equations (see e.g. [Abr91, §2.2]).

In the paradigm of ‘‘Domain Theory in Logical Form’’, spectral domains are particularly important because the logic of the underlying distributive lattice can be incorporated into finitary type systems [Abr91] (see also [AC98, §10.5] for a simple instance).⁹

⁸Note that $\llbracket \text{Str}A \rrbracket$ is always compact, since any Scott-open containing \perp^ω contains the whole of $\llbracket \text{Str}A \rrbracket$.

This contrasts with the product topology on A^ω , which is compact iff A is finite (see e.g. [PP04, §III.3.5]).

⁹In the terminology of [RS23, §5], this is because frames induced by distributive lattices are always spatial (see e.g. [Joh82, Theorem II.3.4]).

4.2 The Sober Space of Models

Given a geometric theory \mathbf{T} over At , we will now equip $\mathbf{Mod}(\mathbf{T})$ with a sober topology induced by a quotient of $\mathbf{Geom}(At)$. In particular, this will extend the bijection of Proposition 3 to an homeomorphism between an algebraic dcpo X and the space of models $\mathbf{Mod}(\mathbf{T}(X))$.

In view of Example 10, we may have $\mathbf{Mod}(\mathbf{T}) = \mathbf{Mod}(\mathbf{U})$ for different theories \mathbf{T} and \mathbf{U} over At . The topology on $M = \mathbf{Mod}(\mathbf{T})$ depends on M (and At), but not on the theory \mathbf{T} such that $M = \mathbf{Mod}(\mathbf{T})$. Fix a set of atomic propositions At and let M be a set of the form $\mathbf{Mod}(\mathbf{T})$ for some theory \mathbf{T} over At . Define

$$\text{mod}_M: \mathbf{Geom}(At) \rightarrow \mathcal{P}(M), \quad \varphi \mapsto \{\nu \in M \mid \nu \models \varphi\}$$

Let $\Omega(M) \subseteq \mathcal{P}(M)$ be the image of mod_M . We have $\text{mod}_M(\mathbf{true}) = M$, and (via Remark 7),

$$\text{mod}_M(\varphi \wedge \psi) = \text{mod}_M(\varphi) \cap \text{mod}_M(\psi) \quad \text{and} \quad \text{mod}_M\left(\bigvee_{i \in I} \varphi_i\right) = \bigcup_{i \in I} \text{mod}_M(\varphi_i)$$

It follows that $(\Omega(M), \subseteq)$ is stable under the sups and the finite infs of $(\mathcal{P}(M), \subseteq)$. Hence $(\Omega(M), \subseteq)$ is a sub-frame of $(\mathcal{P}(M), \subseteq)$, and $(M, \Omega(M))$ is a topological space.

Given a theory \mathbf{T} , we write $\mathbf{Mod}(\mathbf{T})$ for $(\mathbf{Mod}(\mathbf{T}), \Omega(\mathbf{Mod}(\mathbf{T})))$, the *space of models of \mathbf{T}* . Since the space $\mathbf{Mod}(\mathbf{T})$ only depends on the models of \mathbf{T} , the following directly applies to the theory $\mathbf{T}[\mathbf{Str}A]$ of Example 10.

Proposition 4. *Let X be an algebraic dcpo. The bijection $x \mapsto \nu(x)$ of Proposition 3 extends to an homeomorphism from X to $\mathbf{Mod}(\mathbf{T}(X))$.*

Spaces of models are always sober. To this end, given M as above, we quotient $\mathbf{Geom}(At)$ under the preorder \preceq_M with $\varphi \preceq_M \psi$ iff $\text{mod}_M(\varphi) \subseteq \text{mod}_M(\psi)$. The relation \sim_M of M -equivalence on $\mathbf{Geom}(At)$ is defined as $\varphi \sim_M \psi$ iff $\varphi \preceq_M \psi$ and $\psi \preceq_M \varphi$ (i.e. $\text{mod}_M(\varphi) = \text{mod}_M(\psi)$). We let $[\varphi]_M$ be the \sim_M -class of φ , and $\mathbf{Geom}(At)/M$ be the set of \sim_M -classes of geometric formulae. We write \leq_M for the partial order on $\mathbf{Geom}(At)/M$ induced by the preorder \preceq_M (see e.g. [GL13, §2.3.1]).

The function mod_M yields an order-isomorphism $(\mathbf{Geom}(At)/M, \leq_M) \rightarrow (\Omega(M), \subseteq)$. Since order-isomorphisms preserve all existing sups and infs ([DP02, Lemma 2.27(ii)]), we obtain

Lemma 7. *$(\mathbf{Geom}(At)/M, \leq_M)$ is a frame with greatest element $[\mathbf{true}]_M$, and*

$$\begin{aligned} \left[\bigvee_{i \in I} \gamma_i\right]_M \wedge \left[\bigvee_{j \in J} \gamma'_j\right]_M &= \left[\bigvee \{\gamma_i \wedge \gamma'_j \mid i \in I \text{ and } j \in J\}\right]_M \\ \bigvee_{i \in I} \left[\bigvee_{j \in J_i} \gamma_{i,j}\right]_M &= \left[\bigvee \{\gamma_{i,j} \mid i \in I \text{ and } j \in J_i\}\right]_M \end{aligned}$$

Theorem 2. *Let \mathbf{T} be a geometric theory over At . The function taking $\nu \in \mathbf{Mod}(\mathbf{T})$ to $\{[\varphi]_{\mathbf{Mod}(\mathbf{T})} \mid \nu \models \varphi\}$ is an homeomorphism from $\mathbf{Mod}(\mathbf{T})$ to $\text{pt}(\mathbf{Geom}(At)/\mathbf{Mod}(\mathbf{T}))$.*

Corollary 4. *Let \mathbf{T} be a geometric theory. The space $\mathbf{Mod}(\mathbf{T})$ is sober.*

Subspaces. Consider a (sober) space (X, Ω) . Assume that X is represented by a geometric theory \mathbf{T} over At , in the sense that X is homeomorphic to the space $\mathbf{Mod}(\mathbf{T})$. Given a subset $P \subseteq X$, there might be a theory \mathbf{U} over At such that the bijection $X \cong \mathbf{Mod}(\mathbf{T})$ restricts to a bijection $P \cong \mathbf{Mod}(\mathbf{T} \cup \mathbf{U})$. In this case, Proposition 5 below implies that the subspace $(P, \Omega \upharpoonright P)$ is homeomorphic to the space $\mathbf{Mod}(\mathbf{T} \cup \mathbf{U})$, so that $(P, \Omega \upharpoonright P)$ is sober and $\Omega \upharpoonright P$ is isomorphic to $\mathbf{Geom}(At)/\mathbf{Mod}(\mathbf{T} \cup \mathbf{U})$. In such situations, we write $\mathbf{Mod}_{\mathbf{T}}(\mathbf{U})$ for the space $\mathbf{Mod}(\mathbf{T} \cup \mathbf{U})$.

Proposition 5. *Given geometric theories \mathbf{T} and \mathbf{U} on At , the space $\mathbf{Mod}_{\mathbf{T}}(\mathbf{U})$ is equal to the subspace induced by the inclusion $\mathbf{Mod}(\mathbf{T} \cup \mathbf{U}) \subseteq \mathbf{Mod}(\mathbf{T})$.*

Example 11. Let X be an algebraic dcpo, and let $P := \uparrow y$ for some fixed $y \in X$. Given $x \in X$, we have $x \in P$ precisely when $\nu(x)$ is a model of $\mathsf{U}(y) := \{\vdash d \mid d \leq_X y\}$. Hence, the subspace induced by $P \subseteq X$ is homeomorphic to $\mathbf{Mod}_{\mathsf{T}(X)}(\mathsf{U}(y))$.

Assume now $X = \llbracket \mathsf{Str}A \rrbracket$, and let $a \in A$. Since the subspace induced by the LTL formula $\neg \Box a$ is not sober (Corollary 2), it follows from Proposition 5 and Corollary 4 that there is no geometric theory T over $\mathsf{Fin}(\llbracket \mathsf{Str}A \rrbracket)$ such that for all $x \in \llbracket \mathsf{Str}A \rrbracket$, we have $x \in \llbracket \neg \Box a \rrbracket$ iff $\nu(x) \in \mathbf{Mod}(\mathsf{T})$.

4.3 Operations on Theories

We shall now see that for each negation-free formula Φ of LTL, there is a geometric theory $\mathsf{T}[\Phi]$ such that for all streams x , we have $x \in \llbracket \Phi \rrbracket$ iff $\nu(x) \in \mathbf{Mod}(\mathsf{T}[\Phi])$ (where $\nu(x)$ is as in Propositions 3 and 4). This may not be possible if Φ contains negations (Example 11).

To this end, we devise operations on theories which represent unions and intersections of sets of models. Given theories $(\mathsf{T}_i \mid i \in I)$ over At , we let $\bigwedge_{i \in I} \mathsf{T}_i := \bigcup_{i \in I} \mathsf{T}_i$. Then we have

$$\mathbf{Mod}(\bigwedge_{i \in I} \mathsf{T}_i) = \bigcap_{i \in I} \mathbf{Mod}(\mathsf{T}_i)$$

Intersections of sets of models can thus be represented by unions of theories. It is more difficult to devise an operation on theories for *unions* of sets of models. A solution is provided by the following crucial construction. Let $(\mathsf{T}_i \mid i \in I)$ be theories, all over At , with $\mathsf{T}_i = \{\psi_{i,j} \vdash \varphi_{i,j} \mid j \in J_i\}$.

(1) If I is finite, we let $\bigvee_{i \in I} \mathsf{T}_i := \{\bigwedge_{i \in I} \psi_{i,f(i)} \vdash \bigvee_{i \in I} \varphi_{i,f(i)} \mid f \in \prod_{i \in I} J_i\}$.

(2) If I is infinite, and all T_i 's are antecedent-free, $\bigvee_{i \in I} \mathsf{T}_i := \{\vdash \bigvee_{i \in I} \varphi_{i,f(i)} \mid f \in \prod_{i \in I} J_i\}$.

Note that if $(\mathsf{T}_i \mid i \in I)$ consists of countably many countable (antecedent-free) theories, then $\bigwedge_{i \in I} \mathsf{T}_i$ is always countable while $\bigvee_{i \in I} \mathsf{T}_i$ may be uncountable.

Proposition 6. *In both cases above, we have (using the Axiom of Choice when I is infinite)*

$$\mathbf{Mod}(\bigvee_{i \in I} \mathsf{T}_i) = \bigcup_{i \in I} \mathbf{Mod}(\mathsf{T}_i)$$

Example 12. Let X be an algebraic dcpo, and let $P \subseteq X$ be upward-closed. Then $P = \bigcup_{y \in P} \uparrow y$. Hence, given $x \in X$, we have $x \in P$ exactly when $\nu(x)$ is a model of $\bigvee_{y \in P} \mathsf{U}(y)$, where $\mathsf{U}(y)$ is as in Example 11.

4.3.1 Translation of Negation-Free LTL Formulae

Recall from Lemma 1 that if Φ is a negation-free formula of $\mathsf{LTL}(A)$, then $\llbracket \Phi \rrbracket$ is upward-closed in $\llbracket \mathsf{Str}A \rrbracket$. Example 12 thus provides a geometric theory over $\mathsf{Fin}(\llbracket \mathsf{Str}A \rrbracket)$ for $\llbracket \Phi \rrbracket$.¹⁰ But we shall get more information by explicitly defining a geometric theory $\mathsf{T}[\Phi]$ by induction on Φ . Actually, it is even better to work with a stratified presentation of negation-free LTL.

Our stratification of negation-free LTL formulae is based on the following expected fact. Recall from §2.2 the map $\llbracket \circ \rrbracket$ taking $S \in \mathcal{P}(\llbracket \mathsf{Str}A \rrbracket)$ to $\{x \mid x \upharpoonright 1 \in S\} \in \mathcal{P}(\llbracket \mathsf{Str}A \rrbracket)$.

Lemma 8. *Fix set A .*

(1) *The function $\llbracket \circ \rrbracket : \mathcal{P}(\llbracket \mathsf{Str}A \rrbracket) \rightarrow \mathcal{P}(\llbracket \mathsf{Str}A \rrbracket)$ preserves all unions and all intersections.*

(2) *Given LTL formulae Φ, Ψ , let $H_{\Phi, \Psi}$ take $S \in \mathcal{P}(\llbracket \mathsf{Str}A \rrbracket)$ to $\llbracket \Psi \rrbracket \cup (\llbracket \Phi \rrbracket \cap \llbracket \circ \rrbracket(S))$. Then we have $\llbracket \Phi \cup \Psi \rrbracket = \bigcup_{n \in \mathbb{N}} H_{\Phi, \Psi}^n(\llbracket \mathsf{False} \rrbracket)$ and $\llbracket \Phi \mathsf{W} \Psi \rrbracket = \bigcap_{n \in \mathbb{N}} H_{\Phi, \Psi}^n(\llbracket \mathsf{True} \rrbracket)$.*

Figure 1 presents a stratified grammar for negation-free LTL(A). We let $G = G(A)$ be the set of all formulae Φ_1, Ψ_1 from the second layer in Figure 1. $G_\delta = G_\delta(A)$ consists of formulae Φ_2, Ψ_2 from the third layer. The negation-free LTL(A) formulae are the those from the last layer.

¹⁰In fact, when A is countable, if $P \subseteq \llbracket \mathsf{Str}A \rrbracket$ induces a sober subspace, then this subspace is homeomorphic to $\mathbf{Mod}_{\mathsf{T}[\llbracket \mathsf{Str}A \rrbracket]}(\mathsf{U})$ for some (abstractly given) theory U (see [RS23, §5]).

$$\begin{array}{l}
\Phi_0, \Psi_0 \quad ::= \quad \text{False} \quad | \quad \text{True} \quad | \quad a \\
\quad \quad \quad | \quad \Phi_0 \vee \Psi_0 \quad | \quad \Phi_0 \wedge \Psi_0 \quad | \quad \bigcirc \Phi_0 \\
\\
G(A) \ni \Phi_1, \Psi_1 \quad ::= \quad \Phi_0 \\
\quad \quad \quad | \quad \Phi_1 \vee \Psi_1 \quad | \quad \Phi_1 \wedge \Psi_1 \quad | \quad \bigcirc \Phi_1 \quad | \quad \Phi_1 \text{ U } \Psi_1 \\
\\
G_\delta(A) \ni \Phi_2, \Psi_2 \quad ::= \quad \Phi_1 \in G(A) \\
\quad \quad \quad | \quad \Phi_2 \vee \Psi_2 \quad | \quad \Phi_2 \wedge \Psi_2 \quad | \quad \bigcirc \Phi_2 \quad | \quad \Phi_2 \text{ W } \Psi_2 \\
\\
\text{n.-f. LTL}(A) \ni \Phi_3, \Psi_3 \quad ::= \quad \Phi_2 \in G_\delta(A) \\
\quad \quad \quad | \quad \Phi_3 \vee \Psi_3 \quad | \quad \Phi_3 \wedge \Psi_3 \quad | \quad \bigcirc \Phi_3 \quad | \quad \Phi_3 \text{ U } \Psi_3 \quad | \quad \Phi_3 \text{ W } \Psi_3
\end{array}$$

Figure 1. Stratified grammar for negation-free LTL(A), where $a \in A$.

Example 13. Recall from §2.2 that $\diamond\Psi = (\text{True U } \Psi)$ and $\square\Phi = (\Phi \text{ W False})$. Hence, G is closed under $\diamond(-)$ and G_δ is closed under $\square(-)$. But G_δ is (crucially) *not* closed under $\diamond(-)$. In particular, we have $\bigcirc a, \diamond a \in G$ and $\square a, \square\diamond a \in G_\delta$. On the other hand, the negation-free formula $\diamond\square a$ is not a G_δ formula.

When looking at $\diamond\Psi$ and $\square\Phi$ via Lemma 8(2), it is convenient to simplify the functions $H_{\text{True}, \Psi}$ and $H_{\Phi, \text{False}}$ to respectively $\llbracket \Psi \rrbracket \cup \llbracket \bigcirc \rrbracket(-)$ and $\llbracket \Phi \rrbracket \cap \llbracket \bigcirc \rrbracket(-)$. This amounts to restate Lemma 8(2) as $\llbracket \diamond\Psi \rrbracket = \bigcup_{m \in \mathbb{N}} \llbracket \Psi \rrbracket \cup \llbracket \bigcirc\Psi \rrbracket \cup \dots \cup \llbracket \bigcirc^m\Psi \rrbracket$, and similarly for $\llbracket \square\Phi \rrbracket$.

Remark 9. The interpretations of formulae from G or G_δ have the expected topological complexity. Namely, if $\Phi_1 \in G$, then $\llbracket \Phi_1 \rrbracket$ is Scott-open in $\llbracket \text{Str}A \rrbracket$. If $\Phi_2 \in G_\delta$, then $\llbracket \Phi_2 \rrbracket$ is a countable intersection of Scott-opens (i.e. a G_δ subset of $\llbracket \text{Str}A \rrbracket$).

This stratification of negation-free LTL allows for a stratified translation to geometric theories. In fact, each LTL formula $\Phi_1 \in G$ can be translated to a single geometric formula $\mathbf{F}[\Phi_1]$, with $\mathbf{F}[\Phi_0]$ finite when Φ_0 is from the first layer. Formulae Φ from the last two layers will be translated to antecedent-free theories $\mathbf{T}[\Phi]$, with $\mathbf{T}[\Phi_2]$ countable when $\Phi_2 \in G_\delta$.

Fix a set A and let $At := \text{Fin}(\llbracket \text{Str}A \rrbracket)$. We devise operations on geometric formulae and theories which mimic the action of $\llbracket \bigcirc \rrbracket(-)$ on $\mathcal{P}(\llbracket \text{Str}A \rrbracket)$. We begin with geometric formulae. The idea is that given $x \in \llbracket \text{Str}A \rrbracket$ and $d \in \text{Fin}(\llbracket \text{Str}A \rrbracket)$, we have $d \leq_{\llbracket \text{Str}A \rrbracket} x \upharpoonright 1$ exactly when $(\perp \cdot d) \leq_{\llbracket \text{Str}A \rrbracket} x$. The geometric formula $\bigcirc\varphi$ is then defined by propagating the stream operation $d \mapsto \perp \cdot d$ in φ . We set $\bigcirc d := \perp \cdot d$ and

$$\bigcirc \text{true} := \text{true} \quad \bigcirc(\gamma \wedge \gamma') := (\bigcirc\gamma) \wedge (\bigcirc\gamma') \quad \bigcirc \bigvee_{i \in I} \gamma_i := \bigvee_{i \in I} \bigcirc\gamma_i$$

Given a theory Th over At , we let $\bigcirc\text{Th} := \{\bigcirc\psi \vdash \bigcirc\varphi \mid (\psi \vdash \varphi) \in \text{Th}\}$. Note that $\bigcirc\text{Th}$ is antecedent-free whenever so is Th . Recall the map $x \mapsto \nu(x)$ of Propositions 3 and 4.

Lemma 9. *Let $x \in \llbracket \text{Str}A \rrbracket$.*

- (1) *We have $\nu(x) \models \bigcirc\varphi$ if, and only if, $\nu(x \upharpoonright 1) \models \varphi$.*
- (2) *We have $\nu(x) \in \mathbf{Mod}(\bigcirc\text{Th})$ if, and only if, $\nu(x \upharpoonright 1) \in \mathbf{Mod}(\text{Th})$.*

We now define a geometric formula $\mathbf{F}[\Phi_1]$ over At by induction on $\Phi_1 \in G$:

$$\begin{array}{l}
\mathbf{F}[a] \quad := \quad a \cdot \perp^\omega \\
\mathbf{F}[\text{True}] \quad := \quad \text{true} \quad \quad \mathbf{F}[\Phi_1 \wedge \Psi_1] \quad := \quad \mathbf{F}[\Phi_1] \wedge \mathbf{F}[\Psi_1] \\
\mathbf{F}[\text{False}] \quad := \quad \text{false} \quad \quad \mathbf{F}[\Phi_1 \vee \Psi_1] \quad := \quad \mathbf{F}[\Phi_1] \vee \mathbf{F}[\Psi_1] \\
\mathbf{F}[\bigcirc\Phi_1] \quad := \quad \bigcirc\mathbf{F}[\Phi_1] \quad \quad \mathbf{F}[\Phi_1 \text{ U } \Psi_1] \quad := \quad \bigvee_{n \in \mathbb{N}} \mathbf{H}_{\mathbf{F}[\Phi_1], \mathbf{F}[\Psi_1]}^n(\text{false})
\end{array}$$

where $\mathbf{H}_{\varphi, \psi}(\theta) := \psi \vee (\varphi \wedge \bigcirc(\theta))$. (We silently included the case of Φ_0 from the first layer.)

Lemma 10. *Let $\Phi_1 \in G$. Given $x \in \llbracket \text{Str}A \rrbracket$, we have $x \in \llbracket \Phi_1 \rrbracket$ if, and only if, $\nu(x) \models \mathbf{F}[\Phi_1]$.*

Finally, the antecedent-free theory $\mathbf{T}[\Phi_3]$ is defined by induction on Φ_3 as follows:

$$\begin{aligned} \mathbf{T}[\Phi_1] &:= \{\vdash \mathbf{F}[\Phi_1]\} & \mathbf{T}[\bigcirc\Phi_3] &:= \bigcirc\mathbf{T}[\Phi_3] \\ \mathbf{T}[\Phi_3 \wedge \Psi_3] &:= \mathbf{T}[\Phi_3] \wedge \mathbf{T}[\Psi_3] & \mathbf{T}[\Phi_3 \mathbf{W} \Psi_3] &:= \bigwedge_{n \in \mathbb{N}} \mathbf{TH}_{\mathbf{T}[\Phi_3], \mathbf{T}[\Psi_3]}^n(\{\vdash \mathbf{true}\}) \\ \mathbf{T}[\Phi_3 \vee \Psi_3] &:= \mathbf{T}[\Phi_3] \vee \mathbf{T}[\Psi_3] & \mathbf{T}[\Phi_3 \mathbf{U} \Psi_3] &:= \bigvee_{n \in \mathbb{N}} \mathbf{TH}_{\mathbf{T}[\Phi_3], \mathbf{T}[\Psi_3]}^n(\{\vdash \mathbf{false}\}) \end{aligned}$$

where $\mathbf{TH}_{\mathbf{T}, \mathbf{V}}(\mathbf{V}) := \mathbf{U} \vee (\mathbf{T} \wedge \bigcirc\mathbf{V})$. (We silently included the case of $\Phi_2 \in G_\delta$.)

Theorem 3. *Let Φ be negation-free. For $x \in \llbracket \text{Str}A \rrbracket$, we have $x \in \llbracket \Phi \rrbracket$ if, and only if, $\nu(x) \in \mathbf{Mod}(\mathbf{T}[\Phi])$.*

Remark 10. A direct inspection reveals that $\mathbf{F}[\Phi_0]$ is indeed a finite geometric formula when Φ_0 is from the first layer. Similarly, the geometric theory $\mathbf{T}[\Phi_2]$ contains only countably-many sequents when $\Phi_2 \in G_\delta$.

Remark 11. Recall that LTL formulae Φ, Ψ are *equivalent*, notation $\Phi \equiv \Psi$, when $\llbracket \Phi \rrbracket = \llbracket \Psi \rrbracket$. The following standard equivalences are obtained similarly as in [BK08, §5.1.4].

$$\begin{aligned} \bigcirc\mathbf{False} &\equiv \mathbf{False} & \bigcirc(\Phi \vee \Psi) &\equiv \bigcirc\Phi \vee \bigcirc\Psi & \bigcirc(\Phi \mathbf{U} \Psi) &\equiv (\bigcirc\Phi) \mathbf{U} (\bigcirc\Psi) \\ \bigcirc\mathbf{True} &\equiv \mathbf{True} & \bigcirc(\Phi \wedge \Psi) &\equiv \bigcirc\Phi \wedge \bigcirc\Psi & \bigcirc(\Phi \mathbf{W} \Psi) &\equiv (\bigcirc\Phi) \mathbf{W} (\bigcirc\Psi) \end{aligned}$$

Hence, up to equivalence, we can push the \bigcirc 's to atoms $a \in A$. In particular, we may assume that \bigcirc occurs only in first layer's formulae of the form $\bigcirc^n a$.

Example 14. Let Φ_0 be an LTL formula from the first layer in Figure 1. Up to equivalence (Remark 11), we can assume that Φ_0 is in disjunctive normal form, and actually that Φ_0 is a disjunction of conjunctions of formulae of the form $\bigcirc^n a$. Then $\mathbf{F}[\Phi_0]$ is simply a disjunction of conjunctions of atomic propositions of the form $(\perp^n \cdot a \cdot \perp^\omega) \in \mathbf{Fin}(\llbracket \text{Str}A \rrbracket)$.

Consider the formula $\Phi_1 := \diamond\Phi_0 \in G$. Recall that $\diamond\Phi_0 = (\mathbf{True} \mathbf{U} \Phi_0)$, and note that $\mathbf{H}_{\mathbf{true}, \varphi}(\theta) = \varphi \vee (\mathbf{true} \wedge \bigcirc\theta)$. Up to the replacement of $\mathbf{true} \wedge \bigcirc\theta$ by $\bigcirc\theta$, we get that $\mathbf{F}[\Phi_1]$ is the geometric formula $\bigvee_{m \in \mathbb{N}} (\mathbf{F}[\Phi_0] \vee \bigcirc\mathbf{F}[\Phi_0] \vee \dots \vee \bigcirc^m \mathbf{F}[\Phi_0])$. This mirrors the formulation of Lemma 8(2) in Example 13.

We turn to the formula $\Phi_2 := \square\diamond\Phi_0 \in G_\delta$. We have $\square\Phi_1 = (\Phi_1 \mathbf{W} \mathbf{False})$ and we simplify $\mathbf{H}_{\varphi, \mathbf{false}}(\theta)$ to $\varphi \wedge \bigcirc\theta$. The theory $\mathbf{T}[\Phi_2]$ then consists of all the sequents

$$\vdash \bigwedge_{n \leq N} \bigvee_{m \in \mathbb{N}} \bigcirc^n \mathbf{F}[\Phi_0] \vee \bigcirc^{n+1} \mathbf{F}[\Phi_0] \vee \dots \vee \bigcirc^{n+m} \mathbf{F}[\Phi_0]$$

where N ranges over \mathbb{N} . This mirrors the fact that $\llbracket \square\diamond\Phi_0 \rrbracket$ is the set of those streams $x \in \llbracket \text{Str}A \rrbracket$ such that for each $n \in \mathbb{N}$, there is some $m \in \mathbb{N}$ with $x \upharpoonright (n+m) \in \llbracket \Phi_0 \rrbracket$.

Example 15. Continuing Example 14, we now consider the case of $\Phi_3 := \diamond\square a$ with $a \in A$. For this more involved example, we allow ourselves some simplifications that we deliberately avoided in Example 14. Namely, for $\mathbf{T}[\square\Phi]$ and $\mathbf{T}[\diamond\Phi]$ we take respectively $\bigwedge_{n \in \mathbb{N}} \bigcirc^n \mathbf{T}[\Phi]$ and $\bigvee_{m \in \mathbb{N}} \bigcirc^m \mathbf{T}[\Psi]$. Then we have

$$\begin{aligned} \mathbf{T}[\square a] &= \{\bigcirc^n \mathbf{F}[a] \mid n \in \mathbb{N}\} \\ \mathbf{T}[\diamond\square a] &= \{\bigvee_{m \in \mathbb{N}} \bigcirc^{m+f(m)} \mathbf{F}[a] \mid f: \mathbb{N} \rightarrow \mathbb{N}\} \end{aligned}$$

The uncountable theory $\mathbf{T}[\diamond\square a]$ relies on the (classical) choice principle behind Proposition 6. It expresses that given a stream $x \in \llbracket \text{Str}A \rrbracket$, we have $x \notin \llbracket \diamond\square a \rrbracket$ if, and only if, there exists a function $f: \mathbb{N} \rightarrow \mathbb{N}$ such that $x(m+f(m)) \neq a$ for all $m \in \mathbb{N}$. In particular, if $x \notin \llbracket \diamond\square a \rrbracket$, then the function $g: m \mapsto m+f(m)$ finds arbitrary large $n = g(m)$ such that $x(n) \neq a$.

Consider a negation-free $\Phi \in \text{LTL}(A)$. We see the subset $\llbracket \Phi \rrbracket \subseteq \llbracket \text{Str}A \rrbracket$ as a subspace rather than as a sub-dcpo (cf. Example 8). This subspace $\llbracket \Phi \rrbracket = (\llbracket \Phi \rrbracket, \Omega(\llbracket \text{Str}A \rrbracket)) \upharpoonright \llbracket \Phi \rrbracket$ is always sober (Corollary 3). With geometric logic, we gained a description of the subspace $\llbracket \Phi \rrbracket$ as the space of models of the theory $\mathbf{T}[\Phi]$. Namely, the subspace $\llbracket \Phi \rrbracket$ is homeomorphic

to $\mathbf{Mod}_{[\text{Str}A]}(\mathbb{T}[\Phi])$ (Theorem 3 and Proposition 5), while the frame $\Omega([\text{Str}A]) \upharpoonright [\Phi]$ is isomorphic to $\mathbf{Geom}(At)/\mathbf{Mod}_{[\text{Str}A]}(\mathbb{T}[\Phi])$ (Theorem 2). Note that in the latter, two geometric formulae are equivalent exactly when they have the same $\mathbb{T}[\Phi]$ -models. Hence geometric formulae generate the frame $\Omega([\text{Str}A]) \upharpoonright [\Phi]$. Moreover, we have seen in Examples 14 and 15 concrete cases in which the theory $\mathbb{T}[\Phi]$ explicitly represents approximations of $[\Phi]$.

However, a limitation of this approach is that the frame $\mathbf{Geom}(At)/\mathbf{Mod}_{[\text{Str}A]}(\mathbb{T}[\Phi])$ is defined by purely semantic means. We discuss this in [RS23, §5], which gives a complete deduction system for $\mathbb{T}[\Phi]$ in the G_δ case. We now comment on potential extensions to LTL with negation.

Remark 12. Say that an LTL formula Φ is an F formula if Φ is the negation of a G formula. The F_σ formulae are the negations of the G_δ ones. For instance, $\neg a$ (with $a \in A$) is a simple non-trivial F formula, while $\neg \Box a \equiv \Diamond \neg a$ is an F_σ formula. It follows from Remark 9 that F formulae induce Scott-closed subsets of $[\text{Str}A]$, and that the F_σ ones induce countable unions of Scott-closed sets (i.e. F_σ sets).

Now, if $\Phi = \neg \Phi_1$ with $\Phi_1 \in G$, then the subspace $[\Phi]$ is represented by the geometric theory $\{\mathbb{F}[\Phi_1] \vdash \mathbf{false}\}$. Hence Theorem 3 extends to F formulae (so that Proposition 5 and Theorem 2 can be applied in this case). But beware that this does not hold in general for F_σ formulae, since the subspace $[\neg \Box a] = [\Diamond \neg a]$ is not representable in geometric logic (in the sense of Example 11). In particular, there is no geometric theory \mathbb{T} such that $\mathbf{Mod}(\mathbb{T}) = \bigcup_{m \in \mathbb{N}} \mathbf{Mod}\{\bigcirc^m \mathbb{F}[a] \vdash \mathbf{false}\}$, and Proposition 6 does not extend to infinitely many arbitrary theories.

5 A Specification for the Denotation of Filter

The core of this paper consists of the results presented above concerning LTL on streams. However, the long term goal of this work is to reason on input-output (negation-free) LTL properties of functions. We now briefly sketch how our results can help to handle our motivating example, namely the filter function on streams. This is mostly preliminary; we leave as future work the elaboration of a general solution.

We work with the function $[\text{filter}]$ of Remark 2. Fix a finite set A and a Scott-continuous function $p: [A] \rightarrow [\text{Bool}]$. Assume that for all $a \in A$, we have $p(a) \neq \perp_{[\text{Bool}]}$. Let $\Psi = \Psi_p$ as in Example 5 and set $\Phi := \bigvee_{a \in A} a$. The specification (1) for filter leads to the following specification for $[\text{filter}]$:

$$\forall x \in [\text{Str}A], x \text{ total}, \quad x \Vdash \Box \Diamond \Psi \implies ([\text{filter}] p x) \Vdash \Box \Phi \quad (2)$$

where we refrained from writing $x \Vdash \Box \Phi$ for the assumption that x is total.

We use the notations of Remark 2. In particular, $[\text{filter}]p$ is the least fixpoint of the Scott-continuous function $f_p: X \rightarrow X$, where $X := \mathbf{CPO}([\text{Str}A], [\text{Str}A])$. In symbols, we have $[\text{filter}]p = \mathbf{Y}(f_p) = \bigvee_{n \in \mathbb{N}} f_p^n(\perp_X)$.

The standard method to reason on such fixpoints is the rule of *fixpoint induction* (see e.g. [AC98, §6.2]). This rule asserts that given a subset S of a cpo X , and a morphism $f: X \rightarrow_{\mathbf{CPO}} X$, we have $\mathbf{Y}(f) \in S$ provided (i) $\perp_X \in S$, (ii) S is stable under sups of ω -chains, and (iii) $f(x) \in S$ whenever $x \in S$. In our case, the subset of interest is $S := \{f \mid x \text{ total and } x \Vdash \Box \Diamond \Psi \implies f(x) \Vdash \Box \Phi\}$. But fixpoint induction cannot be applied since $\perp_X \notin S$ (as \perp_X takes any $x \in [\text{Str}A]$ to $\perp^\omega \not\Vdash \Box \Phi$).

We can proceed as follows, with the help of §4.3.1. Given $k, n \in \mathbb{N}$ with $k \leq n$, let

$$\begin{aligned} \psi_{n,k} &:= \bigvee \left\{ \bigwedge_{1 \leq j \leq k} \bigcirc^{i_j} \mathbb{F}[\Psi] \mid 0 \leq i_1 < \dots < i_k < n \right\} \\ \varphi_k &:= \bigwedge_{m < k} \bigcirc^m \mathbb{F}[\Phi] \end{aligned}$$

Note that $\nu(x) \in \mathbf{Mod}(\mathbb{T}[\Box \Diamond \Psi])$ if, and only if, $(\forall k \in \mathbb{N})(\exists n \geq k)(\nu(x) \Vdash \psi_{n,k})$. It follows that condition (2) can be obtained from the following.

$$\forall x \in [\text{Str}A], x \text{ total}, \forall k \in \mathbb{N}, \forall n \geq k, \quad \nu(x) \Vdash \psi_{n,k} \implies \nu([\text{filter}] p x) \Vdash \varphi_k \quad (3)$$

Condition (3) is a consequence of Lemma 11 below. The main inductive argument is encapsulated in item (1).

Lemma 11. *Write g_n for $f_p^n(\perp_X): \llbracket \text{Str}A \rrbracket \rightarrow_{\mathbf{CPO}} \llbracket \text{Str}A \rrbracket$. Let $x \in \llbracket \text{Str}A \rrbracket$ be a total stream.*

(1) *Assume $k \leq n$. If $\nu(x) \models \psi_{n,k}$, then $\nu(g_n(x)) \models \varphi_k$.*

(2) *Let $n, k \in \mathbb{N}$. If $\nu(g_n(x)) \models \varphi_k$, then $\nu(\llbracket \text{filter} \rrbracket p x) \models \varphi_k$.*

6 Conclusion

In this paper, we conducted a semantic study of a logic LTL on a domain of streams $\llbracket \text{Str}A \rrbracket$. We showed that the negation-free formulae of LTL induce sober subspaces of $\llbracket \text{Str}A \rrbracket$, and that this may fail in presence of negation. We proposed an inductive translation of negation-free LTL to geometric logic. This translation reflects the semantics of LTL, and we use it to prove that the denotation of `filter` satisfies the specification (1).

Further Works. First, the logic LTL on $\llbracket \text{Str}A \rrbracket$ deserves further studies, in particular regarding decidability and possible axiomatizations.

We think an important next step would be to propose a refinement type system in the spirit of [JR21], but for an extension of PCF with streams. More precisely, the system of [JR21] crucially relies on controlled unfoldings of (formula level) fixpoints. We think that our translation to geometric logic could provide a domain-theoretic analogue for that, yielding a system grounded on DTLF (in the form of [Abr91, §4.3] or e.g. [AC98, §10.5]). This may rely on a deduction system for either LTL or geometric logic.¹¹ In any case, we expect to need an analogue of the iteration terms of [JR21], which actually could simulate (enough of) the infinitary aspects of geometric logic. Also, an important task in this direction would be to formulate sufficiently general reasoning principles for program-level fixpoints.

Further, we expect to handle alternation-free modal μ -properties¹² on (finitary) polynomial types, thus targeting a system which as a whole would be based on FPC. But polynomial types involve sums, and sums are not universal in \mathbf{CPO} , in contrast with \mathbf{DCPO} and with the category \mathbf{CPO}_\perp of *strict* functions. We think of working with Call-By-Push-Value (CBPV) [Lev03, Lev22] for the usual adjunction between \mathbf{DCPO} and \mathbf{CPO}_\perp . On the long run, it would be nice if this basis could extend to enriched models of CBPV, so as to handle further computational effects. Print and global store are particularly relevant, as an important trend in proving temporal properties considers programs generating streams of events. Major works in this line include [SSVH08, HC14, HL17, NUKT18, KT14, UST18, NUKT18, SU23]. In contrast with ours, these approaches are based on trace semantics of syntactic expressions rather than denotational domains.¹³

In a different direction, we think our approach based on geometric logic could extend to linear types [HJK00], for instance targeting systems like [NW03, Win04], and relying on the categorical study of [BF06].

Acknowledgements. This work was partially supported by the ANR-21-CE48-0019 – RECIPROG and by the LABEX MILYON (ANR-10-LABX-0070) of Université de Lyon. It started as a spin-off of ongoing work with Guilhem Jaber and Kenji Maillard. G. Jaber proposed the `filter` function as a motivating example. Thomas Streicher pointed to us the reference [Hec15] (see [RS23, §5]).

¹¹For geometric logic, see [RS23, §5] which completely axiomatizes the countable (e.g. G_δ) case.

¹²This corresponds to “alternation depth 1” in [BW18, §2.2]. See also [BS07, §7] and [SV10].

¹³See e.g. [NUKT18, Theorem 4.1 (and Figure 6)] or [SU23, Theorem 1 (and Definition 20 from the full version)].

References

- [Abr91] S. ABRAMSKY : Domain Theory in Logical Form. *Ann. Pure Appl. Log.*, 51(1-2):1–77, 1991.
- [AC98] R. M. AMADIO et P.-L. CURIEN : *Domains and Lambda-Calculi*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
- [BBT20] T.-D. BRADLEY, T. BRYSON et J. TERILLA : *Topology: A Categorical Approach*. The MIT Press, Cambridge (Mass.) London, 2020.
- [BF06] M. BUNGE et J. FUNK : *Singular Coverings of Toposes*, volume 1890 de *Lecture Notes in Mathematics*. Springer Berlin Heidelberg, 2006.
- [BK08] C. BAIER et J.-P. KATOEN : *Principles of Model Checking*. The MIT Press, 2008.
- [BS07] J. BRADFIELD et C. STIRLING : Modal Mu-Calculi. In P. BLACKBURN, J. VAN BENTHEM et F. WOLTER, éditeurs : *Handbook of Modal Logic*, volume 3 de *Studies in Logic and Practical Reasoning*, pages 721–756. Elsevier, 2007.
- [BW18] J. C. BRADFIELD et I. WALUKIEWICZ : The mu-calculus and Model Checking. In E. M. CLARKE, T. A. HENZINGER, H. VEITH et R. BLOEM, éditeurs : *Handbook of Model Checking*, pages 871–919. Springer, 2018.
- [CZ00] T. COQUAND et G.-Q. ZHANG : Sequents, Frames, and Completeness. In *Proceedings of the 14th Annual Conference of the EACSL on Computer Science Logic*, pages 277–291, London, UK, UK, 2000. Springer-Verlag.
- [DL22] I. DI LIBERTI : General Facts on the Scott Adjunction. *Applied Categorical Structures*, 30:569–591, 2022.
- [DP02] B.A. DAVEY et H.A. PRIESTLEY : *Introduction to Lattices and Order*. Cambridge University Press, 2nd édition, 2002.
- [DST19] M. DICKMANN, N. SCHWARTZ et M. TRESSL : *Spectral Spaces*. New Mathematical Monographs. Cambridge University Press, Cambridge, 2019.
- [GL13] J. GOUBAULT-LARRECQ : *Non-Hausdorff Topology and Domain Theory: Selected Topics in Point-Set Topology*. New Mathematical Monographs. Cambridge University Press, Cambridge, 2013.
- [GvG23] M GEHRKE et S. van GOOL : Topological duality for distributive lattices: Theory and applications, 2023. To appear in the book series Cambridge Tracts in Theoretical Computer Science, Cambridge University Press. Available on arXiv (2203.03286).
- [HC14] M. HOFMANN et W. CHEN : Abstract interpretation from Büchi automata. In T. A. HENZINGER et D. MILLER, éditeurs : *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 51:1–51:10. ACM, 2014.
- [Hec15] R. HECKMANN : Spatiality of countably presentable locales (proved with the Baire category theorem). *Mathematical Structures in Computer Science*, 25(7):1607–1625, 2015.

- [HJK00] M. HUTH, A. JUNG et K. KEIMEL : Linear types and approximation. *Mathematical Structures in Computer Science*, 10(6):719–745, 2000.
- [HL17] M. HOFMANN et J. LEDENT : A cartesian-closed category for higher-order model checking. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017.
- [HR07] I. HODKINSON et M. REYNOLDS : Temporal Logic. In P. BLACKBURN, J. VAN BENTHEM et F. WOLTER, éditeurs : *Handbook of Modal Logic*, volume 3 de *Studies in Logic and Practical Reasoning*, pages 655–720. Elsevier, 2007.
- [Joh82] P.T. JOHNSTONE : *Stone Spaces*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 1982.
- [Joh83] P.T. JOHNSTONE : The point of pointless topology. *Bulletin (New Series) of the American Mathematical Society*, 8(1):41 – 53, 1983.
- [Joh02] P.T. JOHNSTONE : *Sketches of an Elephant: A Topos Theory Compendium*. Oxford Logic Guides. Clarendon Press, 2002.
- [JR21] G. JABER et C. RIBA : Temporal Refinements for Guarded Recursive Types. In N. YOSHIDA, éditeur : *Proceedings of ESOP’21*, volume 12648 de *Lecture Notes in Computer Science*, pages 548–578. Springer, 2021.
- [Kec95] A. S. KECHRIS : *Classical Descriptive Set Theory*, volume 156 de *Graduate Texts in Mathematics*. Springer, 1995.
- [KT14] E. KOSKINEN et T. TERAUCHI : Local Temporal Reasoning. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS’14, New York, NY, USA, 2014. Association for Computing Machinery.
- [KTU10] N. KOBAYASHI, N. TABUCHI et H. UNNO : Higher-Order Multi-Parameter Tree Transducers and Recursion Schemes for Program Verification. In *POPL ’10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 495–508, New York, NY, USA, 2010. Association for Computing Machinery.
- [Lev03] P. B. LEVY : *Call-By-Push-Value*. Semantics Structures in Computation. Springer, Dordrecht, 2003.
- [Lev22] P. B. LEVY : Call-by-Push-Value. *ACM SIGLOG News*, 9(2):7–29, may 2022.
- [NUKT18] Y. NANJO, H. UNNO, E. KOSKINEN et T. TERAUCHI : A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS’18*, pages 759–768, New York, NY, USA, 2018. Association for Computing Machinery.
- [NW03] M. NYGAARD et G. WINSKEL : Full Abstraction for HOPLA. In R.M. AMADIO et D. LUGIEZ, éditeurs : *CONCUR 2003 - Concurrency Theory, 14th International Conference, Marseille, France, September 3-5, 2003, Proceedings*, volume 2761 de *Lecture Notes in Computer Science*, pages 378–392. Springer, 2003.
- [Pie02] B. C. PIERCE : *Types and Programming Languages*. The MIT Press, 1st édition, 2002.

- [Plo77] G. PLOTKIN : LCF Considered as a Programming Language. *Theoretical Computer Science*, 5:223–256, 1977.
- [PP04] D. PERRIN et J.-É. PIN : *Infinite Words: Automata, Semigroups, Logic and Games*. Pure and Applied Mathematics. Elsevier, 2004.
- [PP12] J PICADO et A. PULTR : *Frames and Locales: Topology without points*. Frontiers in Mathematics. Birkhäuser Basel, 2012.
- [PP21] J PICADO et A. PULTR : *Separation in Point-Free Topology*. Birkhäuser Cham, 2021.
- [RS23] C. RIBA et S. STERN : Liveness Properties in Geometric Logic for Domain-Theoretic Streams. Full version, available on arXiv (<https://arxiv.org/abs/2310.12763>), Dec. 2023.
- [SSVH08] C. SKALKA, S. SMITH et D. VAN HORN : Types and Trace Effects of Higher Order Programs. *J. Funct. Program.*, 18(2):179–249, mars 2008.
- [SU23] T. SEKIYAMA et H. UNNO : Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023. Full version available on arXiv at <https://arxiv.org/abs/2207.10386>.
- [SV10] L. SANTOCANALE et Y. VENEMA : Completeness for flat modal fixpoint logics. *Ann. Pure Appl. Logic*, 162(1):55–82, 2010.
- [UST18] H. UNNO, Y. SATAKE et T. TERAUCHI : Relatively complete refinement type system for verification of higher-order non-deterministic programs. *Proc. ACM Program. Lang.*, 2(POPL):12:1–12:29, 2018.
- [Vic89] S. VICKERS : *Topology via Logic*. Cambridge University Press, USA, 1989.
- [Vic07] S. VICKERS : Locales and Toposes as Spaces. In M. AIELLO, I. PRATT-HARTMAN et J. van BENTHEM, éditeurs : *Handbook of Spatial Logics*, chapitre 8, pages 429–496. Springer, 2007.
- [VVK05] H. VÖLZER, D. VARACCA et E. KINDLER : Defining Fairness. In Martín ABADI et Luca de ALFARO, éditeurs : *CONCUR 2005 - Concurrency Theory, 16th International Conference, CONCUR 2005, San Francisco, CA, USA, August 23-26, 2005, Proceedings*, volume 3653 de *Lecture Notes in Computer Science*, pages 458–472. Springer, 2005.
- [Win04] G. WINSKEL : Linearity and nonlinearity in distributed computation. In T. EHRHARD, J.-Y. GIRARD et P. RUET, éditeurs : *Linear Logic in Computer Science*, London Mathematical Society Lecture Note Series. Cambridge University Press, 2004.
- [Zha91] G. ZHANG : *Logic of Domains*. Progress in Theoretical Computer Science. Birkhäuser, Boston, MA, 1991.

Executable semantics of Arm's Architecture Specification Language

Hadrien Renaud¹

¹University College London, London, United Kingdom

The behaviour of Arm instructions is specified using a pseudocode language called ASL that does not yet have semantics. We present an interpreter for this language in both sequential and concurrent setting. This allows us to compare the behaviours of AArch64 instructions as implemented by *herd7*, the tool which Arm uses for describing its memory model, against the behaviours of the same instructions as described in ASL. We find discrepancies between those semantics for the Compare-And-Swap instruction.

1 Introduction

The specification of the Arm architecture is a long document called the Arm Architecture Reference Manual (Arm ARM) [Arm23]. In the Arm ARM, one can find a list of all AArch64 instructions with a representative description of its sequential behaviours in a language called Architecture Specification Language (ASL). The Arm ARM also gives the definition of the Arm memory model, or in other words the rules that govern the execution of a concurrent program written in Arm assembly. The Arm memory model is a formal and executable artefact, written in the domain-specific language *cat* [AMT14, ACM16].

ASL The ASL written in the Arm ARM does not have a formal definition, but rather an informal understanding of how it should work, explained at the end of the Arm ARM (Appendix K 16 page 12795 [Arm23]). After some efforts to build verification flows on top of this pseudocode [Rei16], Arm has decided to transition between this informal pseudocode (referred to as ASLv0) to a new language called ASLv1.

ASLv1 is an imperative language, strongly typed, with loops, exceptions, and functions. The main specificity of this language is its encoding of bit-vectors: bit-vector types are dependent on a length and operations on those are checked during type-checking. The language allows some polymorphism on the lengths of bit-vectors: a function can be declared to work on bits of length N and all the subsequent operations will work on this symbolically defined bit-vector length. Although there is a (still in review and private) specification for the language, this is not a semantics for the language, i.e. a specification of how to execute ASL code, merely a formal syntax with explanations and some type-checking rules.

The memory model The *cat* file which gives the definition of the Arm memory model can be executed by the *herd7* tool [AMT14]. This tool executes symbolically all the instructions in a test, which create events (or Effects as Arm calls them). For example, a load (resp. store) instruction creates a Memory Read (resp. Write) Effect. It also keeps track of the ordering constraints between the different Effects. Those ordering constraints can come

from explicit instructions such as fences, or from data-flow constraints. The *herd7* tool takes as input a model written in *cat*, and uses it to filter out the executions where those ordering constraints don't comply with this model, e.g. when they form a cycle.

The Arm memory model uses two kinds of intra-instruction ordering constraints: *Intrinsic data* dependencies indicate that the value produced by the first event has been used by the other; and *Intrinsic control* dependencies indicate that a binary decision made in a first event allows or not the other event to happen. There is no precise definition of those ordering constraints, and earlier work [AMT14] and the Arm ARM only mention their existence. The Arm ARM also lists intra-instruction dependencies for some instructions, e.g. loads, stores and Compare-And-Swap (see Section 3).

The semantics of AArch64 instructions in *herd7* are devised in tandem by the Arm staff and the *herd7* authors. When possible, this is done using the listing provided by the Arm ARM; otherwise, this work is a loose interpretation of the Arm ARM. This work is a first step in providing semantics to the ASL language, and thus equip the Arm ARM with a semantics of instructions.

Motivation The *herd7* model thus relies on handwritten semantics of instructions. By providing semantics to the ASL code in the Arm manual, we aim at clarifying the semantics of the instructions and enabling comparison of those semantics and the ones given in *herd7*. With an executable ASL semantics, *herd7*'s ASL interpreter can build instruction semantics directly from the Arm ARM's companion artefacts.

2 Contribution

We present two interpreters for ASL, called *ASLRef*, provided alongside the *herd7* tool [AM23], in the directory `asllib`. We use monadic style OCaml to build interpreters which helps building a purely functional interpreter that implements imperative features. They also make our implementation very flexible: by keeping the interpretation monad abstract, we can give the language different semantics by using different monads. We will use this feature to define both sequential and concurrent semantics. For the latter, the monadic style also helps implementing symbolic execution at a low cost. For concurrent semantics, we instantiate our interpreter with a symbolic execution monad, adding a state monad keeping track of the ordering constraints induced by the ASL code. For example, an Intrinsic data dependency is added between two events when there is a data-flow chain between the two.

With our interpreter for ASLv1, also comes a tool to transliterate ASLv0 into ASLv1. Although not complete, this makes it possible to execute ASLv0 pseudocode with our interpreter. For example, it can parse the 37000-line-long ASLv0 library which comes with the pseudocode in the Arm ARM. We also present a working type-checker for ASLv1. It has not yet been the subject of any theoretical study.

3 Validation

For some instructions, the Arm ARM comes with a listing of intra-instruction dependencies. The *herd7* tool follows the Arm ARM in that respect, building Effects and the Intrinsic Dependencies that order them as per those listings.

Our work provides another way to build Effects and Intrinsic Dependencies for a given instruction: we can use our interpreter on the ASL code given for this instruction. One might expect those two ways of building the semantics of an instruction to coincide. Our work demonstrates that this is not always the case and aims to remedy this state of affairs by reconciling both semantics sources. In the following, we will compare the intra-instruction dependencies for the Compare-And-Swap (CAS) instruction, for which the Arm ARM provide a list of intra-instruction dependencies.

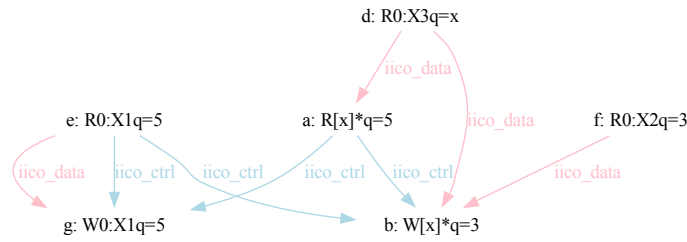


Figure 1. Intrinsic dependencies for CAS (ok case)

```

let address = X[n];
let data = Mem[address];
if data == X[s] then
    Mem[address] = X[t];
end
X[s] = data;
    
```

Figure 2. Simplified ASL code for CAS

Example of the CAS instruction Informally, the instruction CAS X1, X2, [X3] first reads the address indicated by X3; then, if the value read is identical as the value in X1, it writes to the address indicated by X3 the value in register X2; in the end, it writes the value read from memory into the register X1. In the following, we say that CAS is in the *ok* case when the comparison is successful and the write to memory is done.

The intrinsic dependencies graph of herd for this instruction, shown in Figure 1, matches the list of intrinsic dependencies of the Arm ARM for this case. On the other hand, Figure 2 shows a simplified version of ASL pseudocode for the CAS instruction, taken from the Arm ARM. With our semantics of ASL, we can identify where there should be intrinsic dependencies from this ASL code, and we find some differences with the semantics given by herd. One notable difference is intrinsic data dependencies that arrives on the Write Register effect (line 6 of Figure 2, or arrows $e \rightarrow g$ and $a \rightarrow g$ on Figure 1). For each of those potential dependencies, we can construct small AArch64 assembly programs called *litmus tests* that can be simulated [AMT14] or executed on hardware [AMSS11]. Three tests [MR23a, MR23b, MR23c] helped Arm architects to decide which intra-instruction dependencies should exist: the ones from *herd7*, the ones from the Arm ARM, or both? The conclusion reached by Arm is that there should be a non-deterministic choice between either dependency, which is novel for Arm instructions. We have implemented this in *herd7* [Ren23], and we have suggested changes to the ASL code for CAS that are currently being discussed within Arm.

4 Conclusion

We have proposed two interpreters for ASL, one sequential and one concurrent. This has allowed the automatic extraction of intra-instruction dependencies from the ASL code associated with an AArch64 instruction. We have exhibited discrepancies between the *herd7* semantics and the ASL semantics of some instructions, such as CAS, that have been fixed in *herd7* and fixes for the corresponding ASL code are in discussion within Arm.

Further work Pre-existing ASLv0 interpreters exist, both internally and externally to Arm [Rei16, Rei20]. We are currently working on regressing our interpreter against those. To do so, we are building a tool called *ASLCarpenter*, provided alongside our interpreters, which generates ASL code for fuzzing those pre-existing interpreters with respect to ours.

We also aim to automate the work of comparing the *herd7* intra-instruction dependencies and the ones extracted from the ASL code from an instruction. For a given AArch64 litmus test, we can replace the *herd7* semantics of AArch64 instructions by executing the ASL code associated with an instruction. Both way of executing tests should behave the same way with respect to the memory model: any discrepancy should reveal a divergence in the way intra-instruction dependencies are computed. This experiment has been devised with Jade Alglave and is now carried on by Luc Maranget.

References

- [ACM16] Jade ALGLAVE, Patrick COUSOT et Luc MARANGET : Syntax and semantics of the weak consistency model specification language cat. *arXiv preprint arXiv:1608.07531*, 2016.
- [AM23] Jade ALGLAVE et Luc MARANGET : herdtools7, 2023. <https://github.com/herd/herdtools7>.
- [AMSS11] Jade ALGLAVE, Luc MARANGET, Susmit SARKAR et Peter SEWELL : Litmus: Running tests against hardware. *In International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 41–44. Springer, 2011.
- [AMT14] Jade ALGLAVE, Luc MARANGET et Michael TAUTSCHNIG : Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 36(2):1–74, 2014.
- [Arm23] ARM LTD. : Arm architecture reference manual for A-profile architecture., 2023. <https://developer.arm.com/documentation/ddi0487/latest>.
- [MR23a] Luc MARANGET et Hadrien RENAUD : *MP+rel+CAS-ok-bothRs-addr.litmus*, 2023. <https://github.com/herd/herdtools7/blob/8a794d05db101563517720885d31fbce2b778a2b/catalogue/aarch64/tests/MP%2Brel%2BCAS-ok-bothRs-addr.litmus>.
- [MR23b] Luc MARANGET et Hadrien RENAUD : *MP+rel+CAS-ok-MRs-addr.litmus*, 2023. <https://github.com/herd/herdtools7/blob/8a794d05db101563517720885d31fbce2b778a2b/catalogue/aarch64/tests/MP%2Brel%2BCAS-ok-MRs-addr.litmus>.
- [MR23c] Luc MARANGET et Hadrien RENAUD : *MP+rel+CAS-ok-RsRs-addr.litmus*, 2023. <https://github.com/herd/herdtools7/blob/8a794d05db101563517720885d31fbce2b778a2b/catalogue/aarch64/tests/MP%2Brel%2BCAS-ok-RsRs-addr.litmus>.
- [Rei16] Alastair REID : Trustworthy specifications of ARM® v8-A and v8-M system level architecture. *In 2016 Formal Methods in Computer-Aided Design (FMCAD)*, pages 161–168. IEEE, 2016.
- [Rei20] Alastair REID : ASL interpreter, 2020. <https://github.com/alastairreid/asl-interpretter>.
- [Ren23] Hadrien RENAUD : [herd] new semantics for CAS, 2023. <https://github.com/herd/herdtools7/commit/8a794d05db101563517720885d31fbce2b778a2b>.

Towards a linear functional translation for borrowing

Sidney Congard

We present a functional translation of a subset of safe Rust programs, building upon the results of Aeneas [HP22]. It preserves linearity and captures a new feature, namely lifetime bounds. This is a work in progress: in particular, translation rules are not set yet. We discuss perspectives for this work at the end of the paper.

1 Introduction

Rust is, like C++, a systems programming language. It has references called mutable and immutable borrows, giving respectively a read & write and a read-only access to their underlying value. The Rust type system comes with a borrow checker ensuring that they're safely used, notably that values can be either mutated or aliased (i.e. shared) through borrows, but not both at the same time. It eliminates a large class of errors, such as data races or use-after-free.

Recent works exploit this guarantee to give high-level semantics to Rust programs, to facilitate their formal verification. Among those works, most give a logical encoding of Rust programs such as Creusot [DJM22] (using prophecy variables), Verus [LHC⁺23] (using ghost linear types) and Prusti [AMPS19] (using Rust types to apply some rules automatically). However, there is also Aeneas which instead translates Rust programs into pure functional programs, to embed them in proof assistants (initially F-Star, but now also Lean4, Coq and HOL4). Those works are complementary to RustBelt [JJKD18]: it also gives a semantic to *unsafe* code ignoring the borrow checker, but results in more difficult verification.

This paper builds upon the results of Aeneas, to propose a functional translation for a subset of safe Rust programs that handles lifetime bounds. Our version preserves the source program *linearity*, meaning that it doesn't duplicate or discard the borrowed values. For that, we present preliminary results centered around examples of translated programs:

- The second section “framework” presents the translation framework, starting with Rust borrow checker, Aeneas and then showing our translation.
- The third section “lifetime bounds” presents lifetime bounds and shows how the translation handles them. They aren't supported in Aeneas.
- The last section “work in progress and future perspectives” speaks about:
 - The soundness and expressivity of the translation with respect to source programs.
 - Effectful destructors and exceptions, leveraging the (for now unused) linearity from the translation.
 - Two features that aren't supported in Aeneas: nested borrows and polymorphism.

2 Framework

For simplicity, our source language is a stripped-down version of safe Rust: in particular, we’re removing all features that we’ll confidently be able to incorporate later in the translation. That includes several features supported by Aeneas such as loops, inductive types, immutable borrows, pair projections, arrays, the ambient state monad and some syntactic sugar. Also, assertions are only used to access values and to illustrate the current program state: their interpretation in the functional language doesn’t matter. Finally, polymorphism is restricted to types without borrows: this limitation is discussed in the last section. In the rest of this paper, “borrow” will always mean “mutable borrow”. Commented “drop” instructions are inserted by the compiler: they indicate that the dropped variable becomes inaccessible, either because it’s going out of scope or because of borrow checker restrictions.

Polonius [Mat18], a new formulation for the borrow checker, ensures the borrow guarantees by binding each borrow to its origins with regions (also called lifetimes), i.e. sets of loans where a loan is a tag at a possible location of the borrowed value. Each region tells us that to access any loaned value in it, all borrows attached to this region must have been dropped: we can then “end” the region to retrieve the loaned values. That allows great flexibility compared to simple linear or affine types, as long as we respect this discipline of “stacked” borrows [JDKD19] where only the most recent borrows are available.

In the example below, after defining `a`, `b` and `c`, Polonius has two regions: `{a}`, attached to `b` and `{*b}`, attached to `c`. Only `c` does not contain any loan and is thus directly accessible. To access `b`, we must drop `c` first and similarly, to access `a` we must drop `b` and so drop also `c` (as we must access a value to drop it). After running the borrow checker, we obtain the inserted drop annotations for borrows. They do nothing at run-time as they only prevent subsequent borrow uses at compile-time.

```
let mut a = 1;
let b = &mut a; // borrows the value a
let c = &mut *b; // borrows the value previously borrowed by b
// drop c (ends the region {*b} to access b)
*b += 2;
// drop b (ends the region {a} to access a)
assert!(a == 3);
```

In Rust function signatures, borrows must be attached to explicit regions parameters. That allows to describe the loans (i.e. possible origins) of the function output borrows in terms of the loans of the input borrows. All information about borrow dependencies is thus available in the signature, allowing the borrow checker to ignore the function implementation at its call sites. In the example below, after calling `pick_first` we have two regions: `{v}`, attached to `r` and `{u, *r}`, attached to `t`.

```
// The region 'a of the output groups together loans from x and y.
fn pick_first<'a, T>(x: &'a mut T, y: &'a mut T) -> &'a mut T {
    // drop y (goes out of scope)
    x
}
let (mut u, mut v) = (10, 20);
let r = &mut v;
let t = pick_first(&mut u, &mut *r);
*t += 1;
// drop t (ends the region {u, *r} to access r)
*r += 2;
// drop r (ends the region {v} to access v)
assert!(u == 11 && v == 22);
```

To translate such a program, dependencies between overlapping borrows are reified into functions that take the values under the recent borrows and give back the values under the older borrows. Thus, dropping a borrow results in explicitly returning its updated underlying value. But, to do that, contrarily to e.g. Creusot, Aeneas doesn't get enough information from the drops inserted by the borrow checker. So, Aeneas also plays the borrow checker role, carrying additional information to know how to return the dropped values to their origin.

It first translates the Rust source program into the Low-Level Borrow Calculus (abbreviated LLBC), a new formalism equipped with a functional operational semantics that explicits the loans for each borrow at run-time. Then, following its symbolic execution which approximates at compile-time the borrow corresponding loans, we can generate the functional translation of the source program in continuation-passing style. Aeneas translates each Rust function into:

- A forward function, which translates borrows by their underlying values.
- One backward function per region, which translates it as a function taking the same arguments as the forward function plus the values of the output borrows under the region, and returning the actualized values of the input borrows under the region.

The Aeneas translation of the previous program is shown below, as a reference before exposing our translation. A backward function is called when the corresponding region is ended (and so after the borrows attached to it are all dropped), to get the actualized values of the region loans before accessing them. Below, we see that because `pick_first_fwd` returns `x`, `pick_first_back` returns `ret` (the actualized value of `x`) and `y` (which didn't change since the call to `pick_first_fwd` as it's loaned).

```
let pick_first_fwd T (x y: T): T = x
let pick_first_back T (x y ret: T): T * T = (ret, y)

let (u, v) = (10, 20) in
let r = v in
let t = pick_first_fwd u r in
let t = t + 1 in
let (u, r) = pick_first_back u r t (* drop t *)
let r = r + 2 in in
let v = r in (* drop r *)
assert(u == 11 && v == 22)
```

Our translation comes closer to Polonius: we build linear functions $(A_i \times \dots) \multimap (B_i \times \dots)$ (i.e. functions which must be called exactly once and whose arguments cannot be duplicated or discarded) that play the role of regions enriched with information about how borrows are related to their loans, replacing backward functions. It consumes the tuple of values underlying the borrows attached to the region and returns the tuple of loaned values.

Given all the information about borrows and loans dependencies, we are now able to translate basic borrow operations:

- Creating a new borrow `&mut x` of type T is translated by the underlying value x and a new identity route $\lambda(x').(x') : (T) \multimap (T)$ which will transmit the updated value of x .
- Dropping the k th borrow `drop x` from the route $r : (A_0 \times \dots \times A_n) \multimap B$ is translated by passing x in r with $r = \lambda(a_0 \dots a_{k-1}, a_{k+1} \dots a_n).r(a_0 \dots a_{k-1}, x, a_{k+1} \dots a_n)$.
- Ending the route $r : () \multimap (B_0 \times \dots \times B_n)$ loaning the values of the variables b_0, \dots, b_n is translated to `let (b0, ..., bn) = r()`.

- Similar route manipulations are defined to permute their arguments or to compose them on a single input/output.

Each region parameter of a function is translated to a type parameter and in-&-out route arguments. Both route arguments output the type parameter, which is always instantiated with the tuple made from the types of the loaned values for the corresponding region. The input route arguments are the types of the function input borrows underlying values attached to the translated region, and similarly, the output route arguments are the types of the function output borrows underlying values attached to the region. In the translation below, the `pick_first` region `'a` is thus translated by

- The type parameter `A`.
- The input route `a` expecting the updated values under `x` and `y`.
- The output route built from `a` expecting the updated value under the returned borrow `x`.

To call a function, we must pass it new routes that group borrows under the same regions, such as $\lambda(u2, r2).(u2, r2)$ in the translation below.

The last missing ingredient is to adapt a subset of LLBC annotating Rust function signatures, variables and regions to connect borrowed values to matching route arguments. While we don't expose formal translation rules in this paper as they are still being worked out, we give our minimal borrow calculus syntax in the table below.

Sort	Variable	Grammar
VarId	v	
FunId	f	
RegionId	α	
BorrowId	B_i	
LoanId	L_i	
Value	a	$\star \mid (a, a') \mid B_i a \mid L_i$
Route	r	$\vec{L}_i \multimap (\vec{B}_i \mid \alpha)$
FunSignature	s	$[\vec{\alpha}] (\vec{a}, \vec{r}') \rightarrow (a, \vec{r})$
Environment	Ω	$f : s, \vec{\alpha} \vdash v : (a \mid r)$

Such environments annotate Rust variables and routes with a graph of borrow dependencies:

- Edges correspond to borrow and loan identifiers matching pairs B_i, L_i : each borrowed value is annotated with a B_i and must eventually fill the corresponding hole L_i . Note that this can lead to confusion as the route inputs are tagged with loans identifiers and their outputs with borrow identifiers: this is because the route acts like an indirection in which we give back borrowed values.
- Vertices correspond to either values or routes, which have more structure to retain information about compound values or regions. Star (\star) values are values without loans or borrows.

We can then define the dependencies of a given loan identifier $D(\Omega, L_i)$ as the set of variables reachable from it, by fetching the corresponding borrow identifier:

- $B_i \in a$ with $v : a \in \Omega$ a variable annotation, in which case $D(\Omega, L_i) := \{v\}$,
- $B_i \in \vec{B}$ with $r : \vec{L} \multimap \vec{B} \in \Omega$ a route annotation, in which case $D(\Omega, L_i) := \cup \{D(\Omega, L_j) \mid L_j \in \vec{L}\}$.

Except when dropping variables, those dependencies can only grow, but we must avoid making them grow more than necessary: that would prevent us from accessing some variables without a good reason, making the borrow checker too strict. This could happen by merging too many routes while joining different environments, an operation detailed in the next section.

Finally, we can define our main borrow calculus operations on the environment:

- When accessing a variable $v : L_i$, we fetch all the dependencies of loan identifiers in a and call their corresponding routes, removing them from the environment. The environment thus shrinks until there are no loan identifiers left in a .
- When borrowing a variable value $v : a$, we create a route $r : L_i \multimap B_j$ with fresh identifiers and a borrow $b : B_i a$. The variable value is replaced by the route output corresponding loan L_j .

In our translation below, we comment most lines with the borrow calculus environment.

```
let pick_first T A (x y: T) (a: (T * T)→A): T * (T→A) =
  (* α ⊢ x : B_x*, y : B_y*, a : L_x × L_y → α *)
  let a = λret. a (ret, y) in (* drop y *)
  (* α ⊢ x : B_x*, a : L_x → α *)
  (x, a)

let (u, v) = (10, 20) in
  (* ⊢ u : *, v : * *)
  let (r, dr) = (v, λv2. v2) in
    (* ⊢ u : *, v : L_1, r : B_0 *, dr : L_0 → B_1 *)
    let (t, dt) = pick_first u r (λ(u2, r2). (u2, r2)) in
      (* ⊢ u : L_2, v : L_1, r : B_0 L_3, dr : L_0 → B_1, t : B_4 *, dt : L_4 → B_2 × B_3 *)
      let t = t + 1 in
        let (u, r) = dt t (* drop t *)
          (* ⊢ u : *, v : L_1, r : B_0 *, dr : L_0 → B_1 *)
          let r = r + 2 in in
            let v = dr r in (* drop r *)
              (* ⊢ u : *, v : * *)
              assert(u == 11 && v == 22)
```

Identity routes (that we'll now create with "id") could be inlined to get closer to Aeneas forward and backward functions: above, `dr` could be left implicit and function routes such as `pick_first` third argument could be instantiated with `id`. This yields an equivalent, often more concise translation. We'll keep the style presented above to recognize more easily the translated region parameters in our translations, but we could use this inlined version for any example in this article.

```
(* obtained by inlining a := id in the previous "pick_first" definition *)
let pick_first_inl T (x y: T): T * (T→(T * T)) =
  let a = λret. (ret, y) in (* drop y *)
  (x, a)
```

3 Lifetime bounds

By default, regions mentioned in a function signature are disjoint. This is sometimes too restrictive, as illustrated by the second example below: there, we return either the first two or the last two borrows of a function. So, we want to explain to the borrow checker at the signature level that `ret.0` (the first output) may only overlap `x` and `y`, and similarly that

`ret.1` may only overlap `y` and `z`. This is used in the example to avoid dropping `ret.1` while accessing the value under `x`.

For that, we want to group `ret.0`, `x`, `y` under a region and `ret.1`, `y`, `z` under another one. However, borrows (in particular `y`) can only be placed in one region. So instead, we'll put it in a third region with *bounds* to the two other regions, meaning that both `ret.0` and `ret.1` may overlap `y`.

```
fn three_of_two<'a, 'c, 'b: 'a+'c, T>(i: bool, x: &'a mut T, y: &'b mut T,
  z: &'c mut T) -> (&'a mut T, &'c mut T) {
  if i { (x, y) }
  else { (y, z) }
}
let (mut u, mut v, mut w) = (10, 20, 30);
let (r, s) = three_of_two(true, &mut u, &mut v, &mut w);
*r += 1;
*s += 2;
// drop r
assert!(u == 11);
*s += 3;
// drop s
assert!(v == 25 && w == 30);
```

Each lifetime bound `'a: 'b` in a signature is translated as a dependency from the route `'b` to the route `'a`, i.e. as an argument between them. Its type is unknown from the caller's point of view and is therefore masked by an existential type.

Our translation handles the if/else construct by computing the join (a known problem in abstract interpretation) of the two branches, taking the union of their borrow dependencies: $\forall L_i. D(\text{join}(\Omega, \Omega'), L_i) = D(\Omega, L_i) \cup D(\Omega', L_i)$ (the environments must have the same loan & borrow identifiers). While the obtained routes will match the function return type in this example, we cannot guarantee this in general: see the discussion in future work.

```
let three_of_two T A C B (i: bool) (x y z: T) (a: T→A) (c: T→C) (b: T→B):
  ((T * T) * ∃R. ∃S. (T→(A * R)) * (T→(C * S)) * ((R * S)→B)) =
  if i { ((x, y), λx2. (a x2, ()), λy2. (c z, y2), λ((), y2). b y2) }
  else { ((y, z), λy2. (a x, y2), λz2. (c z2, ()), λ(y2, ()). b y2) }

let (u, v, w) = (10, 20, 30) in
let ((r, s), dr, ds, dt) = three_of_two(true, u, v, w, id, id, id) in
let r = r + 1 in
let s = s + 2 in
let (u, t0) = dr r in (* drop r *)
assert(u == 11) in
let s = s + 3 in
let (w, t1) = ds s in (* drop s *)
let v = dt (t0, t1) in
assert(v == 25 && w == 30)
```

To implement this, we define new operations to manipulate routes:

- Adding a dependency between two routes $a \rightsquigarrow b$ is done by adding a unit output to a and a unit input to b , bound by a new pair of identifiers L_i/B_i .
- Masking a dependency between two routes is done by introducing a new existential type for their corresponding output and input.

- Composing routes along a dependency $a \rightsquigarrow b$, if a has a single output or b has a single input, is done by composing them as expected. The condition ensures that all loan dependencies $D(L)$ are preserved so that this rule can be applied whenever wanted, as a normalization rule. This is not the case in general, if the single input or output condition is not verified.

Then, we can translate the if/else clauses with the following steps. They are interleaved with the environment annotations and the generated code for our example in if/else branches, where new expressions fill the previous hole \diamond to detail the expression translating `three_of_two` before inlining sub-expressions.

```

if:  let ret = (x, y) in  $\diamond$ 
      ret : (Bx★, By★), z : Bz★, a : Lx  $\multimap$   $\alpha$ , b : Ly  $\multimap$   $\beta$ , c : Lz  $\multimap$   $\gamma$ 
else: let ret = (y, z) in  $\diamond$ 
      ret : (By'★, Bz'★), x : Bx★, a : Lx  $\multimap$   $\alpha$ , b : Ly'  $\multimap$   $\beta$ , c : Lz'  $\multimap$   $\gamma$ 

```

1. Before joining the two environments, we drop values which are not shared: in our example, `z` in the first branch and `x` in the second branch.

```

if:  let c = c z in  $\diamond$ 
      ret : (Bx★, By★), a : Lx  $\multimap$   $\alpha$ , b : Ly  $\multimap$   $\beta$ , c :  $\gamma$ 
else: let a = a x in  $\diamond$ 
      ret : (By'★, Bz'★), a :  $\alpha$ , b : Ly'  $\multimap$   $\beta$ , c : Lz'  $\multimap$   $\gamma$ 

```

2. We insert identity routes serving as indirections for the new loans and borrows coming from each branch, to allow many-to-many dependencies between loans and borrows.

```

if:  let (r0, r1) = (id, id) in  $\diamond$ 
      ret : (B0★, B1★), a : Lx  $\multimap$   $\alpha$ , b : Ly  $\multimap$   $\beta$ , c :  $\gamma$ , r0 : L0  $\multimap$  Bx, r1 : L1  $\multimap$  By
else: let (r0, r1) = (id, id) in  $\diamond$ 
      ret : (B0★, B1★), a :  $\alpha$ , b : Ly'  $\multimap$   $\beta$ , c : Lz'  $\multimap$   $\gamma$ , r0 : L0  $\multimap$  By', r1 : L1  $\multimap$  Bz'

```

3. We add on those identity routes the dependencies coming from both branches. At this point, the annotations have the same shape in both branches, so we can rename matching pairs of loan/borrow identifiers to equate them (the renaming is not needed for our example).

```

if:  let r0 =  $\lambda$ x. (r0 x, ()) in let b =  $\lambda$ ((), y). b y in
      let r1 =  $\lambda$ y. (r1 y, ()) in let c =  $\lambda$ () . c in  $\diamond$ 
else: let r0 =  $\lambda$ y. ((), r0 y) in let a =  $\lambda$ () . a in
      let r1 =  $\lambda$ z. ((), r1 z) in let b =  $\lambda$ (y, ()) . b y in  $\diamond$ 
ret : (B0★, B1★), a : Lx  $\multimap$   $\alpha$ , b : Ly  $\times$  Ly'  $\multimap$   $\beta$ , c : Lz'  $\multimap$   $\gamma$ ,
r0 : L0  $\multimap$  Bx  $\times$  By', r1 : L1  $\multimap$  By  $\times$  Bz'

```

4. We normalize all routes with the composition rule, which generates the same code for both branches as their environments are the same.

```

let r0 = ( $\lambda$ u. let (v, w) = r0 u in (a v, w)) in
let r1 = ( $\lambda$ u. let (v, w) = r1 u in (v, c w)) in  $\diamond$ 
ret : (B0★, B1★), b : Ly  $\times$  Ly'  $\multimap$   $\beta$ , r0 : L0  $\multimap$   $\alpha$   $\times$  By', r1 : L1  $\multimap$  By  $\times$   $\gamma$ 

```

5. We mask dependencies between routes with new existential types. At this point, the routes have the same type in both branches and can be returned along each if/else result. In the example, some re-ordering of routes and their outputs is needed before returning the if/else result from the function, which is guided by its signature (and inlined in the translation for simplicity).

```
(ret, b, r0, r1)
```


4 Work in progress and future perspectives

4.1 Translation soundness and expressivity

To ensure that the translation is correct, we should aim for a theorem ensuring that the source program behaves like its translated program. For that, the source program should be interpreted with an imperative operational semantics (like the one given by RustBelt or tree borrows [Vil23]), and the translated program with a standard functional semantics. If the theorem is too complicated, we can aim for an operational semantics closer to a dynamic borrow checker instead (like LLBC or stacked borrows). To see that the translation captures closely Rust programs, we may also investigate denotational completeness for some Rust function types T , i.e. that every element of type $\llbracket T \rrbracket$ is the denotation of a translated source program of type T .

We also want to characterize the translation expressivity: as it is currently, it is determined empirically by looking at which programs the translation accepts. This is very similar to what happens with Rust borrow checker. A first step would be to study the differences between Polonius and our translation. The main difficulty seems to accommodate the lost equivalences when enriching regions/relations to routes/functions.

With regions, any two different ways to present the same dependencies between some borrows and loans are equivalent: we obtain different function signatures that can accept the same programs. In particular, we can call the first function inside the second and vice-versa. With routes, two different presentations of the same dependencies may not be equivalent, preventing the translation of the second function call in the first one or vice-versa.

So either we settle on a translation more restrictive than Polonius, which can be acceptable given that the counter-examples require complex signatures that we rarely find in Rust programs, or we try to rewrite the signatures before translating them. To make further progress, it will be necessary to understand better the structure underlying routes.

4.2 Destructors and effects

Rust sees its values as resources: they are linear values equipped with a destructor, i.e. a function that can consume them and may have an effect to clean up the program state (e.g. a dropped file handle closes its underlying file). The call to the destructor is inserted by the compiler for variables going out of scope, making the type system affine. They are also called when an exception (named “panic” in Rust) is raised.

Like Aeneas, this translation was focused on the notion of borrows. However, Rust also has effectful destructors and exceptions (named “panics”) which play a major role in Rust programs:

- Some use emergent patterns, such as tpestates [SY86], which exploit Rust linearity and destructors to maintain guarantees about the program state. This allows to implement state machines at the type level.
- Some must be fault-tolerant, by being in a suitable state after encountering an exception. This can happen at FFI boundaries or for programs with multiple threads. This is especially important for the concurrency model of Rust which allows to share mutable data with a mutex: when an exception happens in a subroutine, it poisons the mutex to emit an error at subsequent accesses to the mutex. We thus prevent deadlocks or accesses to an invalid state. This error handling is similar to the mechanism from this paper on affine sessions [MV14], where errors do not prevent progress properties of the whole program.

Those kinds of guarantees aren’t preserved in the previously mentioned works, notably Aeneas and RustBelt (where it is an explicit limitation).

Some recent works ([CFMM16], [CMM18]) study the interactions between destructors, effects, and exceptions with denotational semantics for linear logic where a suitable notion of resources can be modeled. Our long-term objective is to leverage our translation linearity to reconcile those works with our handling of borrows, to have a renewed understanding of Rust programming paradigm, and to be able to improve the verification of Rust programs.

4.3 Additional features

Other features from Rust are not present in Aeneas paper. Among those, we investigate nested borrows, i.e. borrows of another borrow, and polymorphism (which was restricted to values without borrows).

Nested borrows are treated similarly to simple borrows, except that their value is consumed by the outer borrow route, and then by the inner borrow route. They are thus carrying those two destinations around, like in the example below. With this representation of nested borrows, `&'a mut T` and `&'a mut (&'a mut T)` become equivalent: the inner borrow is dropped at the same time as the outer borrow. The example below shows how a function with nested borrows can be translated and used.

```
fn nested_swaps<'a, 'b, 'c, T>(x: &'a mut (&'c mut T), y: &'b mut (&'c mut T))
{
    swap(x, y); // Exchanges the value *x with *y.
    swap(*x, *y); // Exchanges the value **x with **y.
}
let (mut u, mut v) = (10, 20);
let mut r = &mut u;
let mut s = &mut v;
nested_swap(&mut r, &mut s);
assert!(*r == 10 && *s == 20);
// drop r, s
assert!(u == 20 && v == 10);

let nested_swaps T A B C (x y: T) (a: T→A) (b: T→B) (c: (T*T)→C):
    A * B * ((T*T)→C) =
    let (x, y, c) = (y, x, λ(x2, y2). c (y2, x2)) in (* swap (x, y) *)
    let (x, y) = (y, x) in (* swap ( *x, *y) *)
    let a = a x in (* drop x *)
    let b = b y in (* drop y *)
    (a, b, c)

let (u, v) = (10, 20) in
let (r, dr) = (u, id) in
let (s, ds) = (v, id) in
let (r, s, drs) = nested_swaps r s id id (λ(r3, s3). (dr r3, ds s3)) in
assert(r == 10 && s == 20) in
let (u, v) = drs (r, s) in (* drop r, s *)
assert(u == 20 && v == 10)
```

The issue with polymorphism is that, when a type parameter is instantiated with a type T containing borrows, we must track the commutations and drops of values of type T . Our solution is thus to add a route consuming values of type T as if it was a borrow. That way, the caller gets back the route having consumed the dropped values, and expecting the returned values of type T .

It now remains to establish proper rules and exploit the translation linearity.

References

- [AMPS19] Vytautas ASTRAUSKAS, Peter MÜLLER, Federico POLI et Alexander J. SUMMERS : Leveraging rust types for modular specification and verification. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019.
- [CFMM16] Pierre-Louis CURIEN, Marcelo FIORE et Guillaume MUNCH-MACCAGNONI : A Theory of Effects and Resources: Adjunction Models and Polarised Calculi. *In Proc. POPL*, 2016.
- [CMM18] Guillaume COMBETTE et Guillaume MUNCH-MACCAGNONI : A resource modality for raii (abstract). Rapport technique, avril 2018.
- [DJM22] Xavier DENIS, Jacques-Henri JOURDAN et Claude MARCHÉ : Creusot: a Foundry for the Deductive Verification of Rust Programs. *In ICFEM 2022 - 23th International Conference on Formal Engineering Methods*, Lecture Notes in Computer Science, Madrid, Spain, octobre 2022. Springer Verlag.
- [HP22] Son HO et Jonathan PROTZENKO : Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022.
- [JDKD19] Ralf JUNG, Hoang-Hai DANG, Jeehoon KANG et Derek DREYER : Stacked borrows: An aliasing model for rust. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.
- [JJKD18] Ralf JUNG, Jacques-Henri JOURDAN, Robbert KREBBERS et Derek DREYER : Rustbelt: securing the foundations of the rust programming language. *PACMPL*, 2(POPL):66:1–66:34, 2018.
- [LHC⁺23] Andrea LATTUADA, Travis HANCE, Chanhee CHO, Matthias BRUN, Isitha SUBASINGHE, Yi ZHOU, Jon HOWELL, Bryan PARNO et Chris HAWBLITZEL : Verus: Verifying rust programs using linear ghost types. *Proc. ACM Program. Lang.*, 7(OOPSLA1), apr 2023.
- [Mat18] Nicholas MATSAKIS : An alias-based formulation of the borrow checker, 2018.
- [MV14] Dimitris MOSTROUS et Vasco Thudichum VASCONCELOS : Affine Sessions. *In* David HUTCHISO, Takeo KANADE, ernhard STEFFÉ, Demetri TERZOPOULOS, Doug TYGA, Gerhard WEIKUM, Eva KÜH, Rosario PUGLIESE, Josef KITTLE, Jon M. KLEINBERG, Alfred KOBSA, Friedemann MATTE, John C. MITCHELL, Moni NAO, Oscar NIERSTRASZ et C. Pandu RANGA, éditeurs : *16th International Conference on Coordination Models and Languages (COORDINATION)*, volume LNCS-8459 de *Coordination Models and Languages*, pages 115–130, Berlin, Germany, juin 2014. Springer.
- [SY86] Robert E. STROM et Shaula YEMINI : Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Software Eng.*, 12(1):157–171, 1986.
- [Vil23] Neven VILLANI : Tree borrows, a new aliasing model for rust, 2023.

LANGAGES DE PROGRAMMATION

Type Inference of Polymorphic and Overloaded Functions

Towards Static Typing of Dynamic Languages

Mickaël Laurent¹

¹Université Paris Cité, IRIF, Paris, 75013, France

We present a type-checker prototype that aims to be used with dynamic languages featuring type-cases, such as JavaScript or Elixir. This type system is able to refine the environment when typing the branches of a type-case (occurrence typing), supports both parametric polymorphism (essential for modularity) and ad-hoc polymorphism through intersection types (essential for capturing overloaded behaviors), and features type inference.

1 Introduction

Typing dynamic languages is a challenging endeavour even for very simple pieces of code. For instance, JavaScript’s logical or operator “`||`” behaves like the following function ¹:

```
1 function lOr (x, y) {  
2   if (toBoolean(x)) { return x; } else { return y; }  
3 }
```

where `toBoolean` is a function which simply checks whether its argument is one of the 8 values that are considered as falsy in JavaScript (i.e., `false`, `""`, `0`, `-0`, `0n`, `undefined`, `null`, and `NaN`) and in this case returns `false`, otherwise it returns `true`.

A naive type for the `lOr` function would be the type $(\text{Any}, \text{Any}) \rightarrow \text{Any}$ (where `Any` is the type of all values), which is a rather useless type since it essentially states that `lOr` is a binary function. To give `lOr` a more informative type, we need union and intersection types : we define the type `Falsy` as the union type `false` \vee `""` \vee `0` \vee `-0` \vee `0n` \vee `undefined` \vee `null` \vee `NaN`, where each value denotes here the *singleton type* containing that value, and the type `Truthy` to be its complement, that is, the type of all values that are not of type `Falsy`. This allows `toBoolean` to be typed $(\text{Truthy} \rightarrow \text{True}) \wedge (\text{Falsy} \rightarrow \text{False})$, where \wedge is a type combinator denoting intersection: if the argument of a function of this type is a `Truthy`, then the function returns `true`, while if the argument is a `Falsy`, then the result is `false`. Then, we can deduce for `lOr` the following more precise type:

$$((\text{Truthy}, \text{Any}) \rightarrow \text{Truthy}) \wedge ((\text{Falsy}, \text{Truthy}) \rightarrow \text{Truthy}) \wedge ((\text{Falsy}, \text{Falsy}) \rightarrow \text{Falsy}) \quad (1)$$

Notice how the use of an intersection of arrow types corresponds to the typing of an “overloading” behavior (also known as *ad-hoc* polymorphism), insofar as the result of an application depends on the type of the input. In order to derive such a type, the type system must deduce that whenever the condition tested by the `if` holds, then `x` is of type `Truthy`

¹This definition does not capture the short-circuit evaluation of “`||`”.

and, therefore, (i) that all occurrences of x in the “then” branch (here just one) have type `Truthy` and (ii) that all the occurrences of the same variable x in the branch “else” (here none) have thus type `Falsy`. This kind of deduction is usually referred as *occurrence typing* [THF10] since it requires to “narrow” the type of a variable x differently for its different occurrences. Our prototype is able to perform such type narrowing even when the tested expression is an application. We can be even more precise by inferring intersections of *polymorphic* function types. Indeed, for the `10r` function defined above, the prototype infers the following first order polymorphic type (where α and β are type variables):

$$\forall \alpha, \beta. ((\alpha \wedge \text{Truthy}, \text{Any}) \rightarrow \alpha \wedge \text{Truthy}) \wedge ((\text{Falsy}, \beta) \rightarrow \beta) \quad (2)$$

This type completely specifies the semantics of the function `10r`: it states that if the first argument is a `Truthy`, then the application of the function returns the first argument, otherwise it returns the second argument. This type is more precise than the one in (1), since it allows the system to deduce that, say, if the first argument of `10r` is an object, then the result will be an object of the same type (rather than just a `truthy` value).

2 Features

For any constant of the language, our prototype features the corresponding singleton type (`Nil`, `True`, `False`, etc.). Types can be combined using union `|`, intersection `&`, difference `\`, and negation `~`. Recursive and parametric types are also supported. A built-in syntax for the types of lists is also available, supporting regular expressions. For instance, `[Int ; Bool*]` denotes the lists starting with an integer followed by any number of Boolean values.

```

4 atom null (* Defines a new value null and a singleton type Null *)
5 type Falsy = False | "" | 0 | Null
6 type Truthy = ~Falsy
7 type Tree 'a = ('a\[Any*]) | [(Tree 'a)*] (* Recursive & parametric *)

```

The example from the introduction can be written as follows (the types inferred are written as comments):

```

8 (* (Falsy -> False) & (Truthy -> True) *)
9 let toBoolean x = if x is Truthy then true else false
10 (* ((Falsy, 'b) -> 'b) & (('a\Falsy, Any) -> 'a\Falsy) *)
11 let 10r (x,y) = if toBoolean x then x else y
12 (* 'a -> 'a *)
13 let (id : 'a -> 'a) x = 10r (x,x)

```

Recursive functions such as `map` can be implemented (and typed) by first defining Curry’s fixpoint combinator:

```

14 (* (('a -> 'b) -> X1) -> X1 where X1 = ('a -> 'b) & 'c *)
15 let fixpoint = fun f ->
16   let delta = fun x ->
17     f ( fun v -> ( x x v ))
18   in delta delta
19 (* ... (type omitted for concision) *)
20 let map_stub map f lst =
21   if lst is Nil then nil else (f (fst lst), map f (snd lst))
22 (* (Any -> [ ] -> [ ]) &
23   (('a -> 'b) -> (('a -> ['b]) & (('a+] -> ['b+]) & ([ ] -> [ ])) *)
24 let map = fixpoint map_stub

```

For convenience, and to allow for some optimisations, a syntax for recursive functions is available. Pattern matching is also implemented:

```

25 (* Tree 'a -> ['a*] *)
26 let rec flatten (x:Tree 'a) = match x with
27   | :[] -> []
28   | (h, t) & :List -> concat (flatten h) (flatten t)
29   | _ -> [x]
30 end

```

Note the presence of a user type annotation for the parameter `x`. Still, the type of `flatten` can be inferred without type annotation, in which case the prototype infers the type `(Tree 'a -> ['a*]) & ('b\List -> ['b\List])` in about 300ms (the second arrow captures the fact that, when given a value that is not a list, `flatten` return a singleton list of this value).

This prototype is implemented in OCaml and uses the CDuce library for deciding subtyping. The source code is available in the supplemental material and on Zenodo: [CLN23]. An online version, compiled using `Js_of_ocaml`, can be tested at the following address: <https://www.cduce.org/dynlang/> (it is about 8 times slower than the native version).

3 Limitations and future work

This prototype is an implementation of a type system formalized in an unpublished manuscript [CLN24] currently under submission, based on prior work on set-theoretic types [Fri04, CNXA15, CLNL22]. The formal type system is safe (*well-typed programs cannot go wrong*). However, as we could expect in a type system featuring unrestricted intersections, the type inference is not complete: our inference algorithm is terminating but may fail to type an expression typeable in the formal system.

In particular, the two main weaknesses of this prototype are: (i) the inference time, in particular for recursive functions, and (ii) the lack of principal types.

The first point is due to the backtracking nature of the inference algorithm and the subtyping constraints it has to solve. The inference time can be significantly reduced by providing type annotations, thus narrowing the search space and the amount of backtracking performed by the inference algorithm. Note that the type system is modular: top-level definitions are only typed once (backtracking only occurs inside of a top-level definition).

The second point can be illustrated with the function `map` presented in the previous section. The type inferred for `map` has several intersections. On the one hand, we might prefer to give it its simple ML type, `('a -> 'b) -> ['a*] -> ['b*]`, even though it is less precise. On the other hand, in some specific contexts, we might need an even more precise type, for instance we might need to capture the fact that lists of length 2 are transformed into lists of length 2 (`('a -> 'b) -> ['a;'a] -> ['b;'b]`), and so on. In the limit, we might want to capture the fact that any list of length n is transformed into a list of length n , but this would need either infinite intersections or dependent types, which are not supported by our type system. As the type system is modular, the lack of principal types makes it impossible to systematically guess a suitable type (when typing the top-level definition `map`, we do not know what uses will be made of it in later top-level definitions). Thus, we do not expect the type inference to be completely autonomous, but rather to assist the user by proposing relevant types that they may or may not choose to use.

Lastly, some future work is necessary before considering application to *industrial* languages such as JavaScript. In particular, the current prototype (and the formal type system it is based on) is only sound for pure expressions. Work is being made for supporting side-effects, as well as row-polymorphism for records (which is essential for typing oriented object languages, as records can be used to encode objects). In parallel, set-theoretic type systems are starting to be adopted outside of the academic sphere, for instance for the language Elixir [CDV23] or LUAU [Jef22].

References

- [CDV23] Giuseppe CASTAGNA, Guillaume DUBOC et José VALIM : The design principles of the Elixir type system, 2023.
- [CLN23] Giuseppe CASTAGNA, Mickaël LAURENT et Kim NGUYEN : Prototype: Polymorphic Type Inference for Dynamic Languages, novembre 2023. Available at: <https://doi.org/10.5281/zenodo.10155221>.
- [CLN24] Giuseppe CASTAGNA, Mickaël LAURENT et Kim NGUYEN : Polymorphic Type Inference for Dynamic Languages. *Proceedings of the ACM on Programming Languages*, 8(POPL):40, 2024.
- [CLNL22] Giuseppe CASTAGNA, Mickaël LAURENT, Kim NGUYEN et Matthew LUTZE : On type-cases, union elimination, and occurrence typing. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.
- [CNXA15] Giuseppe CASTAGNA, Kim NGUYEN, Zhiwu XU et Pietro ABATE : Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '15, pages 289–302, janvier 2015.
- [Fri04] Alain FRISCH : *Théorie, conception et réalisation d'un langage de programmation adapté à XML*. Thèse de doctorat, Université Paris Diderot, December 2004.
- [Jef22] Alan JEFFREY : Semantic subtyping in lua. Blog post, novembre 2022. Accessed on May 6th 2023.
- [THF10] Sam TOBIN-HOCHSTADT et Matthias FELLEISEN : Logical types for untyped languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 117–128, New York, NY, USA, 2010. ACM.

STIMULUS: un langage synchrone à contraintes

B. Jeannet¹, E. Closse¹, F. Gaucher¹ et D. Weil¹

¹Dassault Systèmes, 18, chemin de Malacher, 38240 Grenoble

STIMULUS est un langage synchrone à contraintes, dans lequel les équations flots de données définissant la valeur de signaux sont généralisées par des contraintes liant plusieurs signaux. STIMULUS se révèle facile à utiliser et s'applique à la mise au point des exigences fonctionnelles de systèmes de contrôle-commande et au test de ces systèmes. Il est utilisé dans l'industrie des transports et de l'énergie pour les exigences d'applications critiques.

1 Introduction

La vogue des langages à contrainte tels PROLOG et ses descendants MERCURY et OZ [SHC96, RBD⁺03] est quelque peu retombée, peut-être parce que les applications dans lesquelles ils excellent ont été surestimées. STIMULUS est un langage synchrone à contrainte plus récent qui se distingue de ces langages par les aspects suivants : (i) il s'agit d'un langage *synchrone* et non *logique*, dont les contraintes se réduisent toujours à des contraintes sur des variables de type scalaire (booléens, entiers, etc...), plus intuitives que des contraintes d'unification sur des termes *à la* PROLOG ; (ii) il s'agit d'un langage spécifique à un domaine, dont les restrictions autorisent en contre-partie des analyses précises pour garantir la cohérence d'un programme et le compiler efficacement.

La motivation pour ajouter des contraintes à un langage synchrone est la modélisation de propriétés fonctionnelles et temporelles de programmes synchrones. En effet, tandis qu'un programme synchrone émet des sorties déterministes en fonction de ses entrées et de son état interne, une propriété sur de tels programme peut se ramener à des contraintes liant ses sorties à ses entrées et à un état interne propre à la propriété [HLR93].

La conception de STIMULUS s'est faite en ayant à l'esprit deux écueils à éviter :

- un langage trop déclaratif est difficile à déboguer et souvent coûteux à exécuter, comme le montre l'exemple de PROLOG ;
- le mécanisme de *backtrack*, plus ou moins inhérent à tout langage à contrainte, est à exploiter avec la plus grande parcimonie pour les mêmes raisons.

Deux critères ont également guidé nos choix : (i) s'appliquer au mieux à la mise au point d'exigences et au test de systèmes de contrôle-commande, et (ii) éviter à l'utilisateur toute annotation dont le but est de guider l'exécution, comme le *cut* de PROLOG.

2 STIMULUS en quelques mots

STIMULUS est un langage synchrone flots de données dans la lignée des langages LUSTRE et LUCID SYNCHRONE [C PHP87, CPP05] : les variables sont des signaux discrets, qui possèdent une valeur unique par instant logique, *cf.* Fig. 3, les valeurs des signaux sont définies par des

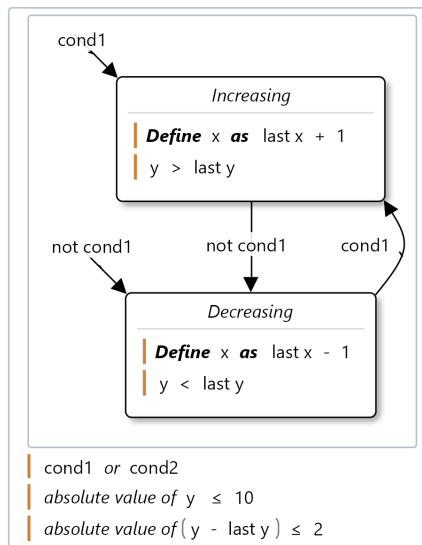


Figure 1. Automate, équations et contraintes

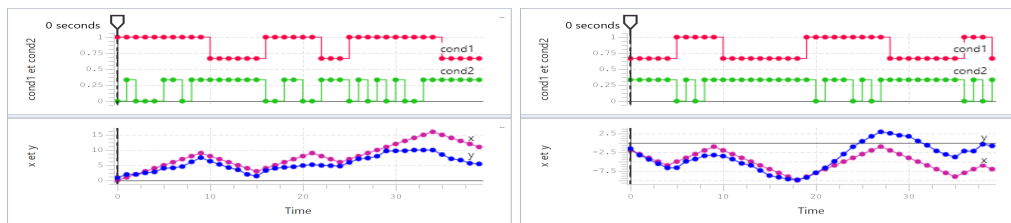


Figure 3. Deux exécutions du programme de la Fig. 2

équations avec *affectation unique*, et le contrôle est spécifié à l'aide d'automates synchrones hiérarchiques [CPP05], cf. les équations sur x introduites par le mot-clé **define** dans chaque état de l'automate de la Fig. 1, où **last** x désigne la valeur de x à l'instant précédent.

Le qualificatif *synchrone* signifie que du point de vue du programmeur, pour tout composant, la lecture des entrées, le calcul et l'émission des sorties se fait dans le même instant logique, et qu'il en est de même pour la propagation des entrées/sorties entre composants en parallèle; ceci rend la composition parallèle synchrone déterministe et commutative. C'est au compilateur que revient la tâche de trouver un ordre d'exécution séquentiel qui donne l'illusion de l'instantanéité des calculs décrite ci-dessus.

Dans ce contexte STIMULUS ajoute les concepts suivants provenant de LUTIN [RRJ08] :

- le concept de *contrainte*, qui introduit du non-déterminisme sur les données, illustré par les contraintes sur $cond1$, $cond2$, et y de la Fig. 1 et les simulations de la Fig. 3;
- le choix non-déterministe dans les automates, qui introduit du non-déterminisme sur le contrôle et du backtrack, en cas de contraintes insatisfaisables;
- la construction *observateur* qui transforme en reconnaisseur de signaux les instructions qu'elle contient, normalement interprétées comme des générateurs de signaux.

STIMULUS possède aussi un système de macros typées, illustrées par le **When** de la Fig. 2, qui rend les programmes plus lisibles en remplaçant les automates par des phrases à trous.

3 Compilation et exécution

La tâche du compilateur est de générer un bytecode dans lequel les instructions sont ordonnées, comme pour tout compilateur synchrone, et où les variables à résoudre ensemble

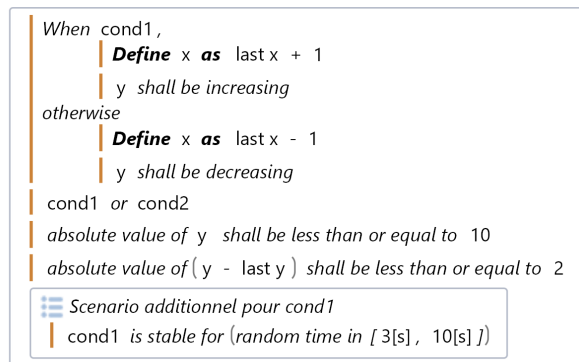


Figure 2. Programme équivalent à celui de la Fig. 1 mais utilisant les composants de la librairie standard de STIMULUS, affichés sous forme de phrases à trous, et auquel on a rajouté un scénario pour le signal $cond1$

et les appels aux solveurs sont explicites. À l'exécution, l'interprète enchaîne calculs explicites, recueil des contraintes actives dépendant des états des automates, et appels au solveur pour tirer des solutions. Le tirage de solutions est aléatoire et paramétré par une graine, ce qui permet d'explorer de nombreuses exécutions possibles et de reproduire l'une d'entre elles.

Un choix sémantique essentiel du langage STIMULUS est que les automates *lisent* les conditions de leurs transitions plutôt qu'ils ne les contraignent, ce qui implique que les automates sont exécutés *après* le calcul de ces conditions. Par exemple, sur la Fig. 1, les signaux *cond1* et *cond2* sont résolus avant *y* et non en même temps. Cette sémantique répond à plusieurs préoccupations mentionnées dans l'introduction :

- elle permet au compilateur de partitionner les contraintes en petits paquets résolus séquentiellement, ce qui rend l'exécution à la fois plus facile à comprendre par l'utilisateur et plus efficace ; les *modes* de MERCURY ont un effet similaire [SHC96] ;
- les transitions conditionnelles restent déterministes et n'introduisent pas de backtrack ;
- tout ceci se fait sans annotation explicite de l'utilisateur.

Cette sémantique pourrait s'avérer trop contraignante en pratique, mais ce n'est pas du tout le cas, d'après notre expérience et le retour de nos utilisateurs.

4 Application aux exigences fonctionnelles

L'application initialement visée par STIMULUS est le test automatisé de systèmes de contrôle-commande. Un banc de test de tels systèmes est constitué (i) d'un environnement qui stimule le système avec des entrées réalistes, (ii) du système sous test, et (iii) un observateur des propriétés attendues.

L'expérience montre que (1) il est très fastidieux de spécifier « à la main » les entrées réalistes d'un tel système, et que (2) lorsque l'observateur est violé, l'analyse de l'erreur mène plus souvent à la correction des propriétés attendues qu'à la correction du système sous test.

Le point (1) est la motivation initiale des langages LUTIN [RRJ08] et STIMULUS, avec l'idée de modéliser les entrées réalistes par des contraintes résolues à l'exécution.

L'observation (2) faite par [JHR13] a mené au cas d'usage de la *mise au point interactive d'exigences*, traitées par STIMULUS comme des générateurs de signaux. Cet usage répond à un besoin bien identifié par nos clients et permet de détecter :

- par inspection visuelle des simulations, les exigences manquantes ou erronées (« ce n'est pas ce que j'ai voulu dire ! ») ;
- et de manière automatique, les exigences contradictoires.

L'objectif est de détecter les erreurs du cahier des charges au moment où on les introduit, et non beaucoup plus tard lors de la confrontation avec implantation. Ce cas d'usage est détaillé dans [JG16, Gau22].

Une fois que les exigences sont mises au point, le second cas d'usage est celui du test automatisé : il consiste à placer les exigences dans le bloc *observateur* du banc de test, qui les transforme en oracle, et de tester si du code ou des exigences de plus bas niveau, constituant le système sous test, les satisfont.

Pour conclure, voici un ordre d'idée des métriques d'un modèle un peu conséquent d'un client dans l'industrie automobile : 1000 automates, exprimés au travers de composants de la librairie standard, et essentiellement composés en parallèle, 2000 variables utilisateurs, et 5000 variables au total générées par le compilateur. À l'exécution, les nombres moyens de contraintes et de variables à résoudre ensemble, par appel au solveur, sont tous deux inférieurs à . . . 2 ! Ceci s'explique d'une part par les choix sémantiques décrits au §3, d'autre part par la constatation que les exigences écrites par les utilisateurs se ramènent généralement à des contraintes élémentaires simples, mais qui dépendent de conditions d'activation compliquées. Enfin, le nombre de cycles synchrones exécutés par seconde se situe entre 100 et 500, ce qui permet l'obtention de simulations pertinentes en quelques secondes.

Références

- [CPHP87] P. CASPI, D. PILAUD, N. HALBWACHS et J. PLAICE : Lustre : a declarative language for programming synchronous systems. *In Symp. on Principles of Programming Languages, POPL'87*. ACM, 1987.
- [CPP05] J-L. COLAÇO, B. PAGANO et M. POUZET : A conservative extension of synchronous data-flow with state machines. *In EMSOFT'05*. ACM, 2005.
- [Gau22] F. GAUCHER : Extending Model-Based Systems Engineering with Requirements-In-the-Loop Simulation : The Landing Gears Case-Study. https://cesam.community/wp-content/uploads/2023/01/14h30-15_1430_CSDM2022_Gaucher.pdf, 2022.
- [HLR93] N. HALBWACHS, F. LAGNIER et P. RAYMOND : Synchronous observers and the verification of reactive systems. *In Algebraic Methodology and Software Technology, AMAST'93*. Springer, 1993.
- [JG16] B. JEANNET et F. GAUCHER : Debugging Embedded Systems Requirements with STIMULUS : an Automotive Case-Study. *In European Congress on Embedded Real Time Software and Systems, ERTS'16*, 2016.
- [JHR13] E. JAHIER, N. HALBWACHS et P. RAYMOND : Engineering functional requirements of reactive systems using synchronous languages. *In Int. Symp. on Industrial Embedded Systems, SIES'13*. IEEE, 2013.
- [RBD⁺03] P. Van ROY, P. BRAND, D. DUCHIER, S. HARIDI, M. HENZ et C. SCHULTE : Logic programming in the context of multiparadigm programming : the oz experience. *Theory and Practice of Logic Programming*, 3(6), 2003.
- [RRJ08] P. RAYMOND, Y. ROUX et E. JAHIER : Lutin : A language for specifying and executing reactive scenarios. *EURASIP J. of Embedded Systems*, 2008.
- [SHC96] Z. SOMOGYI, F. HENDERSON et T. C. CONWAY : The execution algorithm of mercury, an efficient purely declarative logic programming language. *J. of Logic Programming*, 29(1-3), 1996.

Un prototype de système de types graduels ensemblistes pour Elixir

Guillaume Duboc

IRIF, Remote Technology

Elixir est un langage de programmation fonctionnel exécuté sur la machine virtuelle Erlang. Il a récemment fait ses preuves dans divers domaines, notamment les applications Web à large trafic (Discord, Pinterest). La communauté Elixir est en plein essor, et cette recherche répond à une demande croissante d'intégration d'un système de types statiques dans Elixir, similaire à l'approche de Typescript [BAT14] qui a prouvé son efficacité et sa popularité. Sous la direction de Giuseppe Castagna et José Valim, le créateur d'Elixir, ce projet de thèse vise à développer et implémenter un système de types statiques pour le langage. Un prototype d'implémentation de ce système en Elixir est actuellement accessible via <https://typex.fly.dev/> (et sur Zenodo [DCV23]). Cette présentation illustre les difficultés spécifiques au typage d'Elixir, ainsi que les solutions apportées, tout en offrant une perspective sur la conception d'un système de types pratique dont la théorie n'est pas encore complètement établie.

1 Introduction : Elixir et les types ensemblistes

Les fonctions Elixir sont définies par des clauses de filtrage sur les arguments. Une manière typique de définir une fonction en Elixir est donc la suivante :

```
def negate(x) when is_integer(x), do: -x
def negate(x) when is_boolean(x), do: not x
```

La première clause définit la fonction `negate` pour les entiers, et la seconde pour les booléens. Introduire un système de typage statique dans Elixir signifie, dans notre cas, de permettre aux programmeurs d'apposer sur la fonction `negate` une annotation de type, par exemple

```
$ (integer() or boolean()) -> (integer() or boolean())
```

qui indique que la fonction prend en argument un entier ou un booléen, et renvoie un entier ou un booléen. Le connecteur de type `or` est *ensembliste*, c'est-à-dire que si deux types sont vus comme représentant l'ensemble des valeurs qu'ils typent (e.g. 0, 1, 2 ... pour `integer()`), alors le type `(integer() or boolean())` est l'union de ces deux ensembles. Cet opérateur ensembliste n'est pas le seul : en effet, on peut se rendre compte que si `negate` est appelée par une fonction annotée comme `subtract`

```
$ (integer(), integer()) -> integer()
def subtract(x, y) when is_integer(x) and is_integer(y) do
  x + negate(y)
end
```

alors une erreur de type serait levée, déclarant que l'opérateur `+` est appelé avec un terme de type `integer() or boolean()`. Ici, la notion de type *intersection* est utilisée pour donner un type fonctionnel plus précis, polymorphe ad-hoc, à la fonction `negate` :

```
$ (integer() -> integer()) and (boolean() -> boolean())
```

Ce type capture le fait que, lorsque `negate` reçoit un entier en argument, elle renvoie un entier, et lorsque c'est un booléen, un booléen. Ces deux connecteurs font partie de la syntaxe du langage et peuvent être utilisés librement dans les annotations de types, avec également la négation de type, les types singletons, et les variables de types, tous trois utilisés par l'exemple suivant :

```
$ ((false or nil, a) -> a) and
  ((b, term()) -> b) when a: term(), b: not(false or nil)
def logical_or(x, y) when x == false or x == nil, do: y
def logical_or(x, _), do: x
```

La fonction `logical_or` prend deux arguments en entrée ; si le premier est un atome `false` ou `nil` (ceux-ci étant représentés par les types singletons du même nom), alors le second argument est renvoyé, dont le type est capturé par la variable `a` (quantifiée après le type, avec une borne supérieure). Cette variable n'a pas de contrainte particulière puisque sa borne supérieure est `term()` soit le type de toutes les valeurs. Sinon, c'est le premier argument qui est renvoyé, dont le type est capturé par la variable `b` ; celle-ci a pour borne supérieure le type négation `not(false or nil)`, qui est le type de toutes les valeurs qui ne sont ni `false` ni `nil`.

2 Spécificités du typage d'Elixir

Si le cœur de ce typeur réside dans son adoption d'un système de types basé sur le sous-typage sémantique [CF05], une approche enrichie au fil des années avec du polymorphisme [CX11] et du typage graduel [CLPS19], l'extension de ce système à Elixir [CDV23] nécessite des efforts conséquents que nous présentons ici.

2.1 Analyse typée du filtrage avec gardes

Typing Elixir nécessite d'offrir une analyse typée précise des gardes, la difficulté venant du fait que les gardes sont des expressions complexes : ce sont des combinaisons booléennes (avec `and`, `or`, `not`), des tests d'égalité (`==`, `!=`), des comparaisons (`<`, `<=`, `>`, `>=`), des tests de type (`is_atom`, `is_integer`, etc.), des tests de présence dans une liste (`in`), des sélections dans des tuples, listes ou dictionnaires (`elem`, `hd`, `tl`, `map.key`), des opérateurs sur les structures de données (`tuple_size`, `map_size`, etc.). Cette analyse permet à notre typeur de détecter des erreurs de non exhaustivité (ou des branches redondantes) dans le filtrage par motifs, comme dans cet exemple de code qui manipule un type de résultat encodé dans un dictionnaire avec des champs `:output`, `:socket` ou bien `:output`, `:message`.

```
$ type result() =
  %{output: :ok, socket: socket()} or
  %{output: :error, message: :timeout or {:delay, integer()}}

$ result() -> string()
def handle(r) when r.output == :ok, do: "Msg received"
def handle(r) when r.message == :timeout, do: "Timeout"
#=> Type Warning: non-exhaustive pattern matching

$ result() -> string()
def hand(r) when r.output == :ok, do: "Msg received"
def hand(r) when r.output == :error, do: "Error raised"
def hand(%{socket: _}), do: "Socket found"
#=> Type Warning: unused branch
```

Comme les fonctions sont définies par clauses de filtrage successives, cela nous aussi permet d'inférer des types intersections précis qui capturent la relation exacte entre types d'entrées et de sorties de chaque clause. Par exemple, dans cet exemple

```
def handle(r) when r.output == :ok, do: {:accepted, r.socket}
def handle(r) when is_atom(r.message), do: r.message
def handle(r), do: {:retry, elem(r.message, 1)}

$ %{output: :ok, socket: socket()} -> {:accept,socket()} and
  %{output: :error, message: :timeout} -> :timeout and
  %{output: :error, message: {:delay,integer()}} -> {:retry,integer()}
```

Ces différents exemples sont traités dans l'interface Typex sous le nom (`result/handle`).

2.2 Typage graduel : une approche hybride via les types fonctionnels forts

Le typage graduel [ST06] (c.à.d. introduire un type `dynamic()` qui peut devenir n'importe quel type à l'exécution) en pratique donne lieu à des compromis entre sûreté du typage et quantité d'information fournie par le typeur. Typiquement, le typeur de Typescript est non-sûr car, par défaut Javascript ne vérifie pas les types, et le type dynamique, ne peut ainsi pas être contrôlé même lorsqu'il est utilisée par des programmes entièrement annotés. Récupérer cette sûreté nécessite d'activer un mode de compilation spécifique [RSF⁺15].

Notre approche ici est de proposer un système de types sûr par défaut, sans modifier l'exécution. Cet objectif est atteint en prenant en compte les vérifications de types effectuées par la machine virtuelle Erlang. Celle-ci vérifie de manière systématiques les arguments d'opérateurs comme l'addition ; nous parlons d'opérations fortes, car elles sont sûres par défaut. Les gardes écrites par les programmeurs font également l'objet de telles vérifications, ce qui nous mène à ajouter au système de types des types fonctionnels forts, qui désignent une fonction qui vérifie concrètement le type de ses arguments. Ainsi l'identité avec ou sans garde explicite est typée avec le même type, mais si ces fonctions sont appelées avec une expression de type dynamique, elles renverront des types différents :

<pre>\$ integer() -> integer() def id_weak(x), do: x id_weak(x_dyn) #=> dynamic()</pre>	<pre>\$ integer() -> integer() def id_strong(x) when is_integer(x), do: x id_strong(x_dyn) #=> integer()</pre>
--	---

3 Intérêt du logiciel

Le prototype Typex est une implémentation en Elixir d'un futur système de types permettant d'ajouter des annotations de types aux programmes Elixir, et de les vérifier. Il a été présenté lors de conférences majeures comme ElixirConf 2023 et BEAM Code 2023, recevant un accueil très positif. La question du typage dans les communautés Erlang et Elixir est très présente, avec divers efforts et théories [LS06, eqw, Jos19, CTPV20, VH18, SWB23], dont l'adoption n'est pas complète, en premier lieu car résoudre les problèmes spécifiques à Elixir de manière assez satisfaisante nécessite des techniques en partie développées dans ce projet. José Valim, le créateur d'Elixir, joue un rôle actif et soutient fortement ce projet de recherche, ce qui augmente considérablement sa visibilité et son acceptabilité au sein de la communauté Elixir. Le typeur développé dans le cadre de ce projet, dont Typex représente une façade de démonstration, est prévu pour être intégré progressivement au compilateur, ce qui ferait à terme d'Elixir un langage statiquement typé par défaut.

Références

- [BAT14] G. BIERMAN, M. ABADI et M. TORGERSEN : Understanding TypeScript. *In European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.
- [CDV23] Giuseppe CASTAGNA, Guillaume DUBOC et José VALIM : The Design Principles of the Elixir Type System, 2023.
- [CF05] Giuseppe CASTAGNA et Alain FRISCH : A gentle introduction to semantic subtyping. *In Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 198–208, 2005.
- [CLPS19] Giuseppe CASTAGNA, Victor LANVIN, Tommaso PETRUCCIANI et Jeremy G SIEK : Gradual typing : a new perspective. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, 2019.
- [CTPV20] Mauricio CASSOLA, Agustín TALAGORRIA, Alberto PARDO et Marcos VIERA : A gradual type system for Elixir. *In Proceedings of the 24th Brazilian Symposium on Context-oriented Programming and Advanced Modularity*, pages 17–24, 2020.
- [CX11] Giuseppe CASTAGNA et Zhiwu XU : Set-theoretic foundation of parametric polymorphism and subtyping. *In Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, pages 94–106, 2011.
- [DCV23] Guillaume DUBOC, Giuseppe CASTAGNA et Jose VALIM : Accepted artifact for 'the design principles of the elixir type system'. <https://doi.org/10.5281/zenodo.8425534>, 2023.
- [eqw] eqWAlizer. <https://github.com/WhatsApp/eqwalizer>.
- [Jos19] Svenningsson JOSEF : Gradualizer. <https://github.com/josefs/Gradualizer>, 2019.
- [LS06] Tobias LINDAHL et Konstantinos SAGONAS : Practical type inference based on success typings. *In ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming*, 2006.
- [RSF⁺15] A. RASTOGI, N. SWAMY, C. FOURNET, G. BIERMAN et P. VEKRIS : Safe & efficient gradual typing for TypeScript. *In POPL '15*, pages 167–180. ACM, 2015.
- [ST06] J. G. SIEK et W. TAHA : Gradual typing for functional languages. *In Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.
- [SWB23] Albert SCHIMPF, Stefan WEHR et Annette BIENIUSA : Set-theoretic types for erlang. *arXiv preprint arXiv :2302.12783*, 2023.
- [VH18] Nachiappan VALLIAPPAN et John HUGHES : Typing the wild in Erlang. *In Proceedings of the 17th ACM SIGPLAN International Workshop on Erlang*, pages 49–60, 2018.

Destination-passing style programming: a Haskell implementation

Thomas Bagrel¹

¹INRIA/LORIA, Vandœuvre-lès-Nancy, 54500, France

¹Tweag, Paris, 75012, France

Destination-passing style programming introduces destinations, which represent the address of a write-once memory cell. Those destinations can be passed as function parameters, and thus enable the caller of a function to keep control over memory management: the body of the called function will just be responsible of filling that memory cell. This is especially useful in functional programming languages, in which the body of a function is typically responsible for allocation of the result value.

Programming with destination in Haskell is an interesting way to improve performance of critical parts of some programs, without sacrificing memory guarantees. Indeed, thanks to a linearly-typed API I present, a write-once memory cell cannot be left uninitialized before being read, and is still disposed of by the garbage collector when it is not in use anymore, eliminating the risk of uninitialized read, memory leak, or double-free errors that can arise when memory is managed manually.

In this article, I present an implementation of destinations for Haskell, which relies on so-called compact regions. I demonstrate, in particular, a simple parser example for which the destination-based version uses 35% less memory and time than its naive counterpart for large inputs.

1 Introduction

Destination-passing style (DPS) programming takes its source in the early days of imperative languages with manual memory management. In the C programming language, it's quite common for a function not to allocate memory itself for its result, but rather to receive a reference to a memory location where to write its result (often named *out parameter*). In that scheme, the caller of the function has control over allocation and disposal of memory for the function result, and thus gets to choose where the latter will be written.

DPS programming is an adaptation of this idea for functional languages, based on two core concepts: having arbitrary data structures with *holes* — that is to say, memory cells that haven't been filled yet — and *destinations*, which are pointers to those holes. A destination can be passed around, as a first-class object of the language (unlike holes), and it allows remote action on its associated hole: when one *fills* the destination with a value, that value is in fact written in the hole. As structures are allowed to have holes, they can be built from the root down, rather than from the leaves up. Indeed, children of a parent

node no longer have to be specified when the parent node is created; they can be left empty (which leaves holes in the parent node), and added later through destinations to those holes. It is thus possible to write very natural solutions to problems for which the usual functional bottom-up building approach is ill-fitting. On top of better expressiveness, DPS programming can lead to better time or space performance for critical parts of a program, allowing for example tail-recursive map, or efficient difference lists.

That being said, DPS programming is not about giving unlimited manual control over memory or using mutations without restrictions. The existence of a destination is directly linked to the existence of an accompanying hole: we say that a destination is *consumed* when it has already been used to write something in its associated hole. It must not be reused after that point, to ensure immutability and prevent a range of memory errors.

In this paper, I design a destination API whose memory safety (write-once model) is ensured through a linear type discipline. Linear type systems are based on Girard’s Linear logic [Gir95], and introduce the concept of *linearity*: one can express through types that a function will *consume* its argument exactly once given the function result is *consumed* exactly once. Linearity helps to manage resources — such as destinations — that one should not forget to *consume* (e.g. forgetting to fill a hole before reading a structure), but also that shouldn’t be reused several times.

The Haskell programming language is equipped with support for linear types through its main compiler, *GHC*, since version 9.0.1 [BBN⁺18]. But Haskell is also a *pure* functional language, which means that side effects are usually not safe to produce outside of monadic contexts. This led me to set a slightly more refined goal: I wanted to hide impure memory effects related to destinations behind a *pure* Haskell API, and make the whole safe through the linear type discipline. Although the proofs of type safety haven’t been made yet, the early practical results seem to indicate that my API is safe, and its purity makes it more convenient to adopt DPS in a codebase compared to a monadic approach that would be more “contaminating”.

The main contributions of this paper are

- a linearly-typed API for destinations that let us build and manipulate data structures with holes while exposing a pure interface (Section 4);
- a first implementation of destinations for Haskell relying on so-called compact regions (Section 5), together with a performance evaluation (Section 6). Implementation code is available in [Bag23a] and [Bag23b] (specifically `src/Compact/Pure/Internal.hs` and `bench/Bench`).

2 A short primer on linear types

Linear Haskell [BBN⁺18] introduces the linear function arrow, $\mathbf{a} \multimap \mathbf{b}$, that guarantees that the argument of the function will be consumed exactly once when the result of the function is consumed exactly once. On the other hand, the regular function arrow $\mathbf{a} \rightarrow \mathbf{b}$ doesn’t guarantee how many times its argument will be consumed when its result is consumed once.

A value is said to be *consumed once* (or *consumed linearly*) when it is pattern-matched on and its sub-components are consumed once; or when it is passed as an argument to a linear function whose result is consumed once. A function is said to be *consumed once* when it is applied to an argument and when the result is consumed exactly once. We say that a variable x is *used linearly* in an expression u when consuming u once implies consuming x exactly once. Linearity on function arrows thus creates a chain of requirements about consumption of values, which is usually bootstrapped by using the *scope function* trick, as detailed in Section 4.2.

Unrestricted values Linear Haskell introduces a wrapper named `Ur` which is used to indicate that a value in a linear context doesn't have to be used linearly. `Ur a` is equivalent to `!a` in linear logic, and there is an equivalence between `Ur a -> b` and `a -> b`.

The value `(x, y)` is said to be consumed linearly only when both `x` and `y` are consumed exactly once; whereas `Ur x` is considered to be consumed once as long as one pattern-matches on it, even if `x` is not consumed exactly once after (it can be consumed several times or not at all). Conversely, both `x` and `y` are used linearly in `(x, y)`, whereas `x` is not used linearly in `Ur x`. As a result, only values already wrapped in `Ur` or coming from the left of a non-linear arrow can be put in another `Ur` without breaking linearity. The only exceptions are values of types that implement the `Movable` typeclass such as `Int` or `()`. `Movable` provides `move :: a -> Ur a` so a value can escape linearity restrictions.

Operators Some Haskell operators are often used in the rest of this article:

`(<&>)` :: `Functor f => f a -> (a -> b) -> f b` is the same as `fmap` with the order of the arguments flipped: `x <&> f = fmap f x`;

`(;)` :: `() -> b -> b` is used to chain a linear operation returning `()` with one returning a value of type `b` without breaking linearity;

`Class => ...` is notation for typeclass constraints (resolved implicitly by the compiler).

3 Motivating examples for DPS programming

The following subsections present three typical settings in which DPS programming brings expressiveness or performance benefits over a more traditional functional implementation.

3.1 Efficient difference lists

Linked lists are a staple of functional programming, but they aren't efficient for concatenation, especially when the concatenation calls are nested to the left.

In an imperative context, it would be quite easy to concatenate linked lists efficiently. One just has to keep both a pointer to the root and to the last *cons* cell of each list. Then, to concatenate two lists, one just has to mutate the last *cons* cell of the first one to point to the root of the second list.

It isn't possible to do so in an immutable functional context though. Instead, *difference lists* can be used: they are very fast to concatenate, and then to convert back into a list. They tend to emulate the idea of having a mutable (here, write-once) last *cons* cell. Usually, a difference list `x1 : ... : xn : □` is encoded by function `\ys -> x1 : ... : xn : ys` taking a last element `ys :: [a]` and returning a value of type `[a]` too.

With such a representation, concatenation is function composition: `f1 <> f2 = f1 . f2`, and we have `empty = id1`, `toList f = f []` and `fromList xs = \ys -> xs ++ ys`.

In DPS, instead of encoding the concept of a write-once hole with a function, we can represent the hole of type `[a]` as a first-class object with a *destination* of type `Dest [a]`. A difference list now becomes an actual data structure in memory — not just a pending computation — that has two handles: one to the root of the list of type `[a]`, and one to the yet-to-be-filled hole in the last *cons* cell, represented by the destination of type `Dest [a]`.

With the function encoding, it isn't possible to read the list until a last element of type `[a]` has been supplied to complete it. With the destination representation, this constraint must persist: the actual list `[a]` shouldn't be readable until the accompanying destination is filled, as pattern-matching on the hole would lead to a dreaded *segmentation fault*. This constraint is embodied by the `Incomplete a b` type of our destination API: `b` is what needs to be linearly consumed to make the `a` readable. The `b` side often carries the destinations of

¹`empty` and `<>` are the usual notations for neutral element and binary operation of a monoid in Haskell.

a structure. A difference list is then `type DList a = Incomplete [a] (Dest [a])`: the `Dest [a]` must be filled (with a `[a]`) to get a readable `[a]`.

The implementation of destination-backed difference lists is presented in Table 1. More details about the API primitives used by this implementation are given in Section 4. For now, it's important to note that `fill` is a function taking a constructor as a type parameter (often used with `@` for type parameter application, and `'` to lift a constructor to a type).

- `alloc` (Figure 1) returns a `DList a` which is exactly an `Incomplete [a] (Dest [a])` structure. There is no data there yet and the list that will be fed in `Dest [a]` is exactly the list that the resulting `Incomplete` will hold. This is similar to the function encoding where $\backslash x \rightarrow x$ represents the empty difference list;
- `append` (Figure 3) adds an element at the tail position of a difference list. For this, it first uses `fill @'(:)` to fill the hole at the end of the list represented by `d :: Dest [a]` with a hollow `cons` cell with two new holes pointed by `dh :: Dest a` and `dt :: Dest [a]`. Then, `fillLeaf` fills the hole represented by `dh` with the value of type `a` to append. The hole of the resulting difference list is the one pointed by `dt :: Dest [a]` which hasn't been filled yet.
- `concat` (Figure 4) concatenates two difference lists, `i1` and `i2`. It uses `fillComp` to fill the destination `dt1` of the first difference list with the root of the second difference list `i2`. The resulting `Incomplete` object hence has the same root as the first list, holds the elements of both lists, and inherits the hole of the second list. Memory-wise, `concat` just writes an address into a memory cell; no move is required.
- `toList` (Figure 2) completes the incomplete structure by plugging `nil` into its hole with `fill @' []` (whose individual behavior is presented in Figure 6) and removes the `Incomplete` wrapper as the structure is now complete, using `fromIncomplete_'`.

To use this API safely, it is imperative that values of type `Incomplete` are used linearly. Otherwise we could first complete a difference list with `l = toList i`, then add a new `cons` cell with a hole to `i` with `append i x` (actually reusing the destination inside `i` for the second time). Doing that creates a hole inside `l`, although it is of type `[a]` so we are allowed to pattern-match on it (so we might get a segfault)! The simplified API of Table 1 doesn't actually enforce the required linearity properties, I'll address that in Section 4.2.

This implementation of difference list matches closely the intended memory behaviour, we can expect it to be more efficient than the functional encoding. We'll see in Section 6 that the prototype implementation presented in Section 5 cannot yet demonstrate these performance improvements.

3.2 Breadth-first tree traversal

Consider the problem, which Okasaki attributes to Launchbury [Oka00]

Given a tree T , create a new tree of the same shape, but with the values at the nodes replaced by the numbers $1 \dots |T|$ in breadth-first order.

This problem admits a straightforward implementation if we're allowed to mutate trees. Nevertheless, a pure implementation is quite tricky [Oka00, Gib93] and a very elegant, albeit very clever, solution was proposed recently [GKSW23].

With destinations as first-class objects in our toolbox, we can implement a solution that is both easy to come up with and efficient, doing only a single breadth-first traversal pass on the original tree. The main idea is to keep a queue of pairs of a tree to be relabeled and of the destination where the relabeled result is expected (as destinations can be stored in arbitrary containers!) and process each of them when their turn comes. The implementation provided in Table 2, implements the slightly more general `mapAccumBFS` which applies on each node of the tree a relabeling function that can depend on a state.

```

1  data [a] = {- nil constructor -} [] | {- cons constructor -} (:) a [a]
2
3  type DList a = Incomplete [a] (Dest [a])
4
5  alloc :: DList a -- API primitive (simplified signature w.r.t. Section 4)
6
7  append :: DList a -> a -> DList a
8  append i x =
9    i <&> \d -> case fill @'(:) d of
10     (dh, dt) -> fillLeaf x dh ; dt
11
12  concat :: DList a -> DList a -> DList a
13  concat i1 i2 = i1 <&> \dt1 -> fillComp i2 dt1
14
15  toList :: DList a -> [a]
16  toList i = fromIncomplete_' (i <&> \dt -> fill @'[] dt)

```

Table 1. Implementation of difference lists with destinations

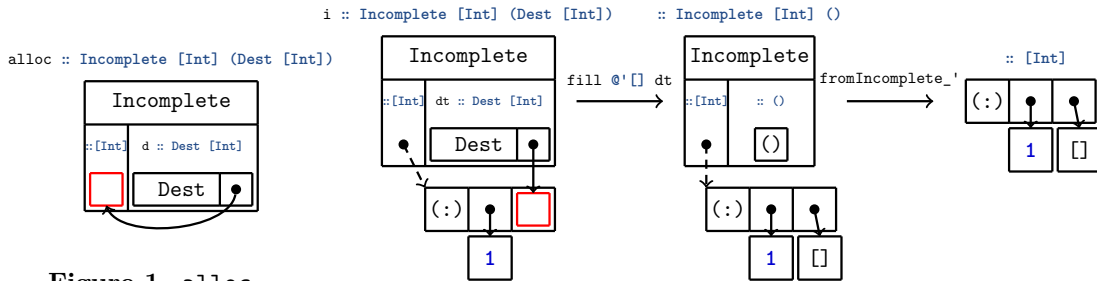


Figure 1. alloc

Figure 2. Memory behavior of toList i

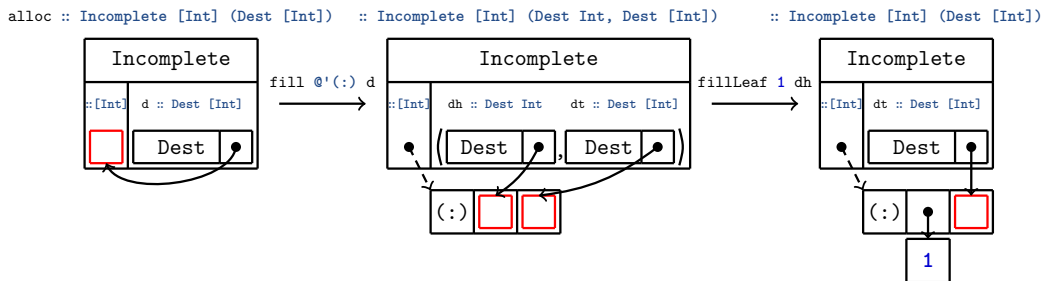


Figure 3. Memory behavior of append alloc 1

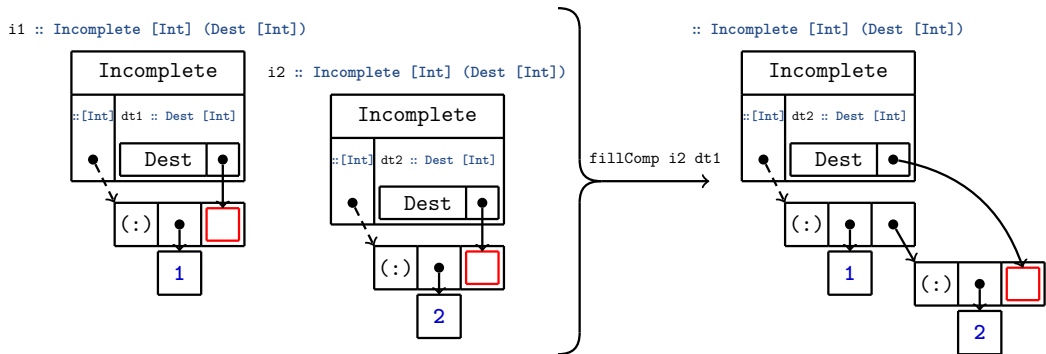


Figure 4. Memory behavior of concat i1 i2 (based on fillComp)

```

1 data Tree a = Nil | Node a (Tree a) (Tree a)
2
3 relabelDPS :: Tree a → Tree Int
4 relabelDPS tree = fst (mapAccumBFS (\st _ → (st + 1, st)) 1 tree)
5
6 mapAccumBFS :: ∀ a b s. (s → a → (s, b)) → s → Tree a → (Tree b, s)
7 mapAccumBFS f s0 tree =
8   fromIncomplete' ( -- simplified alloc signature w.r.t. Section 4
9     alloc <&> \dtree → go s0 (singleton (Ur tree, dtree)))
10  where
11    go :: s → Queue (Ur (Tree a), Dest (Tree b)) → Ur s
12    go st q = case dequeue q of
13      Nothing → Ur st
14      Just ((utree, dtree), q') → case utree of
15        Ur Nil → fill @'Nil dtree ; go st q'
16        Ur (Node x tl tr) → case fill @'Node dtree of
17          (dy, dtl, dtr) →
18            let q'' = q' `enqueue` (Ur tl, dtl) `enqueue` (Ur tr, dtr)
19                (st', y) = f st x
20            in fillLeaf y dy ; go st' q''

```

Table 2. Implementation of breadth-first tree traversal with destinations

Note that the signatures of `mapAccumBFS` and `relabelDPS` don't involve linear types. Linear types only appear in the inner loop `go`, which manipulates destinations. Linearity enforces the fact that every destination ever put in the queue is eventually filled at some point, which guarantees that the output tree is complete after the function has run.

As the state-transforming function $s \rightarrow a \rightarrow (s, b)$ is non-linear, the nodes of the original tree won't be used in a linear fashion. However, we want to store these nodes together with their accompanying destinations in a queue of pairs, and destinations used to construct the queue have to be used linearly (because of `<&>` signature, which initially gives the root destination). That forces the queue to be used linearly, so the pairs too, so pairs' components too. Thus, we wrap the nodes of the input tree in the `Ur` wrapper, whose linear consumption allows for unrestricted use of its inner value, as detailed in Section 2.

This example shows how destinations can be used even in a non-linear setting in order to improve the expressiveness of the language. This more natural and less convoluted implementation of breadth-first traversal also presents great performance gains compared to the fancy functional implementation from [GKSW23], as detailed in Section 6.

3.3 Deserializing, lifetime, and garbage collection

In client-server applications, the following pattern is very frequent: the server receives a request from a client with a serialized payload, the server then deserializes the payload, runs some code, and respond to the request. Most often, the deserialized payload is kept alive for the entirety of the request handling. In a garbage collected language, there's a real cost to this: the garbage collector (GC) will traverse the deserialized payload again and again, although we know that all its internal pointers are live for the duration of the request.

Instead, we'd rather consider the deserialized payload as a single heap object, which doesn't need to be traversed, and is freed as a block. GHC supports this use-case with a feature named *compact regions* [YCA⁺15]. Compact regions contain normal heap objects, but the GC never follows pointers into a compact region. The flipside is that a compact region can only be collected when all of the objects it contains are dead.

With compact regions, we would first deserialize the payload normally, in the GC heap, then copy it into a compact region and only keep a reference to the copy. That way, internal

```

1 parseSList :: ByteString → Int → [SEExpr] → Either Error SEExpr
2 parseSList bs i acc = case bs !? i of
3   Nothing → Left (UnexpectedEOFList i)
4   Just x → if
5     | x == ')' → Right (SList i (reverse acc))
6     | isSpace x → parseSList bs (i + 1) acc
7     | otherwise → case parseSEExpr bs i of
8       Left err → Left err
9       Right child → parseSList bs (endPos child + 1) (child : acc)

```

Table 3. Implementation of the S-expression parser without destinations

```

1 parseSListDPS :: ByteString → Int → Dest [SEExpr] → Either Error Int
2 parseSListDPS bs i d = case bs !? i of
3   Nothing → fill @'[] d ; Left (UnexpectedEOFList i)
4   Just x → if
5     | x == ')' → fill @'[] d ; Right i
6     | isSpace x → parseSListDPS bs (i + 1) d
7     | otherwise →
8       case fill @'(:) d of
9         (dh, dt) → case parseSEExprDPS bs i dh of
10           Left err → fill @'[] dt ; Left err
11           Right endPos → parseSListDPS bs (endPos + 1) dt

```

Table 4. Implementation of the S-expression parser with destinations

pointers of the region copy will never be followed by the GC, and that copy will be collected as a whole later on, whereas the original in the GC heap will be collected immediately.

However, we are still allocating two copies of the deserialized payload. This is wasteful, it would be much better to allocate directly in the region, but this isn't part of the original compact region API. Destinations are one way to accomplish this. In fact, as I'll explain in Section 5, my implementation of DPS for Haskell is backed by compact regions because they provide more freedom to do low-level memory operations without interfering with GC.

Given a payload serialized as S-expressions, let's see how using destinations and compact regions for the parser can lead to greater performance. S-expressions are parenthesized lists whose elements are separated by spaces. These elements can be of several types: int, string, symbol (a textual token with no quotes around it), or a list of other S-expressions.

Parsing an S-expression can be done naively with mutually recursive functions:

- `parseSEExpr` scans the next character, and either dispatches to `parseSList` if it encounters an opening parenthesis, or to `parseSString` if it encounters an opening quote, or eventually parses the string into a number or symbol;
- `parseSList` calls `parseSEExpr` to parse the next token, and then calls itself again until reaching a closing parenthesis, accumulating the parsed elements along the way.

Only the implementation of `parseSList` will be presented here as it is enough for our purpose, but the full implementation of both the naive and destination-based versions of the whole parser can be found in `src/Compact/Pure/SEExpr.hs` of [Bag23b].

The implementation presented in Table 3 is quite standard: the accumulator `acc` collects the nodes that are returned by `parseSEExpr` in the reverse order (because it's the natural building order for a linked list without destinations). When the end of the list is reached (line 5), the accumulator is reversed, wrapped in the `SList` constructor, and returned.

We will see that destinations can bring very significant performance gains with only very little stylistic changes in the code. Accumulators of tail-recursive functions just have to be changed into destinations. Instead of writing elements into a list that will be reversed


```

1 data Token
2 consume  ::      Token -> ()
3 dup2     ::      Token -> (Token, Token)
4 withToken :: ∀ a. (Token -> Ur a) -> Ur a
5
6 data Incomplete a b
7 fmap     :: ∀ a b c. (b -> c) -> Incomplete a b -> Incomplete b c
8 alloc    :: ∀ a.      Token -> Incomplete a (Dest a)
9 intoIncomplete :: ∀ a.      Token -> a -> Incomplete a ()
10 fromIncomplete_ :: ∀ a.      Incomplete a () -> Ur a
11 fromIncomplete  :: ∀ a b.     Incomplete a (Ur c) -> Ur (a, c)
12
13 data Dest a
14 type family Destsof lCtor a -- returns dests associated to fields of constructor
15 fill      :: ∀ lCtor a. Dest a -> Destsof lCtor a
16 fillComp  :: ∀ a b.     Incomplete a b -> Dest a -> b
17 fillLeaf  :: ∀ a.       a -> Dest a -> ()

```

Table 5. Destination API for Haskell

at the end as we did before, the program in the destination style will directly write the elements into their final location.

Code for `parseSListDPS` is presented in Table 4. Let’s see what changed compared to the naive implementation:

- even for error cases, we are forced to consume the destination that we receive as an argument (to stay linear), hence we write some sensible default data to it (see line 3);
- the `SExpr` value resulting from `parseSExprDPS` is not collected by `parseSListDPS` but instead written directly into its final location by `parseSExprDPS` through the passing and filling of destination `dh` (see line 9);
- adding an element of type `SExpr` to the accumulator `[SExpr]` is replaced with adding a new cons cell with `fill @'(:)` into the hole represented by `Dest [SExpr]`, writing an element to the *head* destination, and then doing a recursive call with the *tail* destination passed as an argument (which has type `Dest [SExpr]` again);
- instead of reversing and returning the accumulator at the end of the processing, it is enough to complete the list by writing a nil element to the tail destination (with `fill @'[]`, see line 5), as the list has been built in a top-down approach;
- DPS functions return the offset of the next character to read instead of a parsed value.

Thanks to that new implementation which is barely longer (in terms of lines of code) than the naive one, the program runs almost twice as fast, mostly because garbage-collection time goes to almost zero. The detailed benchmark is available in Section 6.

4 API Design

Table 5 presents my pure API for functional DPS programming. This API is sufficient to implement all the examples of Section 3. This section explains its various parts in detail.

4.1 The Incomplete type

The main design principle behind DPS structure building is that no structure can be read before all its destinations have been filled. That way, incomplete data structures can be freely passed around and stored, but need to be completed before any pattern-matching can be made on them.

Hence we introduce a new data type `Incomplete a b` where `a` stands for the type of the structure being built, and `b` is the type of what needs to be linearly consumed before the structure can be read. The idea is that one can map over the `b` side, which will contain destinations or containers with destinations inside, until there is no destination left but just a non-linear value that can safely escape (e.g. `()`, `Int`, or something wrapped in `Ur`). When destinations from the `b` side are consumed, the structure on the `a` side is built little by little in a top-down fashion, as we showed in Figures 3 and 4. And when no destination remains on the `b` side, the value of type `a` no longer has holes, thus is ready to be released/read.

It can be released in two ways: with `fromIncomplete_`, the value on the `b` side must be unit `()`, and just the complete `a` is returned, wrapped in `Ur`. With `fromIncomplete`, the type on the `b` side must be of the form `Ur c`, and then a pair `Ur (a, c)` is returned.

It is actually safe to wrap the structure that has been built in `Ur` because its leaves either come from non-linear sources (as `fillLeaf :: a → Dest a → ()` consumes its first argument non-linearly) or are made of 0-ary constructors added with `fill`, both of which can be used in an unrestricted fashion safely. Variants `fromIncomplete_` and `fromIncomplete` from the beginning of this article just drop the `Ur` wrapper.

Conversely, the function `intoIncomplete` takes a non-linear argument of type `a` and wraps it into an `Incomplete` with no destinations left to be consumed.

4.2 Ensuring write-once model for holes with linear types

Types aren't linear by themselves in Linear Haskell. Instead, functions can be made to use their arguments linearly or not. So in direct style, where the consumer of a resource isn't tied to the resource creation site, there is no way to state that the resource must be used exactly once:

```
1 createR      :: Resource -- no way to force the result to be used exactly once
2 consumerR   :: Resource → ()
3 exampleShouldFail :: () =
4   let x = createR in consumerR x ; consumerR x -- valid even if x is consumed twice
```

The solution is to force the consumer of a resource to become explicit at the creation site of the resource, and to check through its signature that it is indeed a linear continuation:

```
1 withR       :: (Resource → a) → a
2 consumerR   :: Resource → ()
3 exampleFail :: () = withR (\x → consumerR x ; consumerR x) -- not linear
```

The `Resource` type is in positive position in the signature of `withR`, so that the function should somehow know how to produce a `Resource`, but this is opaque for the user. What matters is that a resource can only be accessed by providing a linear continuation to `withR`.

Still, this is not enough; because `\x → x` is indeed a linear continuation, one could use `withR (\x → x)` to leak a `Resource`, and then use it in a non-linear fashion in the outside world. Hence we must forbid the resource from appearing anywhere in the return type of the continuation. To do that, we ask the return type to be wrapped in `Ur`: because the resource comes from the left of a linear arrow, and doesn't implement `Movable`, it cannot be wrapped in `Ur` without breaking linearity (see Section 2). On the other hand, a `Movable` value of type `()` or `Int` can be returned:

```
1 withR'      :: (Resource → Ur a) → Ur a
2 consumerR   :: Resource → ()
3 exampleOk'   :: Ur ()           = withR' (\x → let u :: () = consumerR x' in move u)
4 exampleFail' :: Ur Resource = withR' (\x → Ur x) -- not linear
```

This explicit *scope function* trick will no longer be necessary when linear constraints will land in GHC (see [SKB⁺22]). In the meantime, this principle has been used to ensure safety of the DPS implementation in Haskell.

Ensuring linear use of Incomplete objects If an `Incomplete` object is used linearly, then its destinations will be written to exactly once; this is ensured by the signature of `fmap` for `Incompletes`. So we need to ensure that `Incomplete` objects are used linearly. For that, we introduce a new type `Token`. A token can be linearly exchanged one-for-one with an `Incomplete` of any type with `alloc`, linearly duplicated with `dup2`, or linearly deleted with `consume`. However, it cannot be linearly stored in `Ur` as it doesn't implement `Movable`.

As in the example above, we just ensure that `withToken :: (Token \multimap Ur a) \multimap Ur a` is the only source of `Tokens` around. Now, to produce an `Incomplete`, one must get a token first, so has to be in the scope of a continuation passed to `withToken`. Putting either a `Token` or `Incomplete` in `Ur` inside the continuation would make it non-linear. So none of them can escape the scope as is, but a structure built from an `Incomplete` and finalized with `fromIncomplete` would be automatically wrapped in `Ur`, thus could safely escape².

4.3 Filling functions for destinations

The last part of the API is the one in charge of actually building the structures in a top-down fashion. To fill a hole represented by `Dest a`, three functions are available:

`fillLeaf :: \forall a. a \rightarrow Dest a \multimap ()` uses a value of type `a` to fill the hole represented by the destination. The destination is consumed linearly, but the value to fill the hole isn't (as indicated by the first non-linear arrow). Memory-wise, the address of the object `a` is written into the memory cell pointed to by the destination (see Figure 7).

`fillComp :: \forall a b. Incomplete a b \multimap Dest a \multimap b` is used to plug two `Incomplete` objects together. The target `Incomplete` isn't represented in the signature of the function. Instead, only the target hole that will receive the address of the child is represented by `Dest a`; and `Incomplete a b` in the signature refers to the child object. A call to `fillComp` always takes place in the scope of `fmap/⟨&⟩` over the parent object:

```

1 parent :: Incomplete BigStruct (Dest SmallStruct, Dest OtherStruct)
2 child :: Incomplete SmallStruct (Dest Int)
3 comp = parent ⟨&⟩ \ds extra  $\rightarrow$  fillComp child ds
4      :: Incomplete BigStruct (Dest Int, Dest OtherStruct)

```

The resulting structure `comp` is morally a `BigStruct` like `parent`, that inherited the hole from the child structure (`Dest Int`) and still has its other hole (`Dest OtherStruct`) to be filled. An example of memory behavior of `fillComp` in action can be seen in Figure 4.

`fill :: \forall lCtor a. Dest a \multimap DestsOf lCtor a` lets us build structures using layers of hollow constructors. It takes a constructor as a type parameter (`lCtor`) and allocates a hollow heap object that has the same header/tag as the specified constructor but unspecified fields. The address of the allocated hollow constructor is written in the destination that is passed to `fill`. As a result, one hole is now filled, but there is one new hole in the structure for each field left unspecified in the hollow constructor that is now part of the bigger structure. So `fill` returns one destination of matching type for each of the fields of the constructor. An example of the memory behavior of `fill @'(:) :: Dest [a] \multimap (Dest a, Dest [a])` is given in Figure 5 and the one of `fill @'[] :: Dest [a] \multimap ()` is given in Figure 6.

`DestsOf` is a type family (i.e. a function operating on types) whose role is to map a constructor to the type of destinations for its fields. For example, `DestsOf '[] [a] = ()` and `DestsOf '(:) [a] = (Dest a, Dest [a])`. More generally, there is a duality between the type of a constructor `Ctor :: (f1...fn) \rightarrow a` and of the associated destination-filling functions `fill @'Ctor :: Dest a \multimap (Dest f1...Dest fn)`. Destination-based data building can be seen as more general than the usual bottom-up constructor approach, as we can recover `Ctor` from the associated function `fill @'Ctor`, but not the reverse:

²This is why the `fromIncomplete'` and `fromIncomplete_'` variants aren't that useful in the actual memory-safe API (which differs slightly from the simplified examples of Section 3): here the built structure would be stuck in the scope function without its `Ur` escape pass.

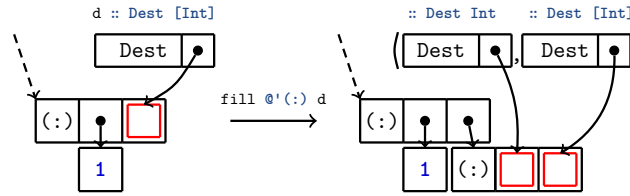


Figure 5. Memory behavior of `fill @'(:) :: Dest [a] -> (Dest a, Dest [a])`

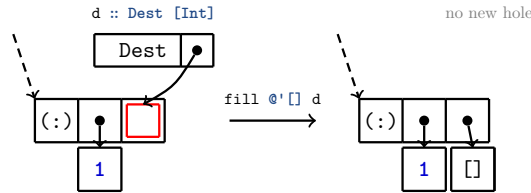


Figure 6. Memory behavior of `fill @'[] :: Dest [a] -> ()`

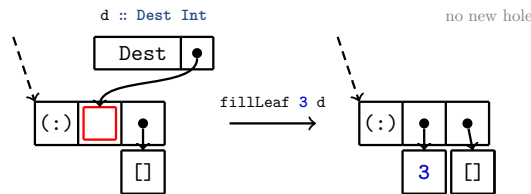


Figure 7. Memory behavior of `fillLeaf :: a -> Dest [a] -> ()`

```

1 Ctor :: (f1...fn) -> a
2 Ctor (x1...xn) = fromIncomplete_' (
3   alloc <&> \ (d :: Dest a) -> case fill @'Ctor d of
4   (dx1...dxn) -> fillLeaf x1 dx1 ; ... ; fillLeaf xn dxn)

```

5 Implementing destinations in Haskell

Having incomplete structures in the memory inherently introduces a lot of tension with both the garbage collector and compiler. Indeed, the GC assumes that every heap object it traverses is well-formed, whereas incomplete structures are absolutely ill-formed: they contain uninitialized pointers, which the GC should absolutely not follow.

The tension with the compiler is of lesser extent. The compiler can make some optimizations because it assumes that every object is immutable, while DPS programming breaks that guarantee by mutating constructors after they have been allocated (albeit only one update can happen). Fortunately, these errors are easily detected when implementing the API, and fixed by asking GHC not to inline specific parts of the code (with pragmas).

5.1 Compact Regions

As I teased in Section 3.3, *compact regions* from [YCA⁺15] make it very convenient to implement DPS programming in Haskell. A compact region represents a memory area in the Haskell heap that is almost fully independent from the GC and the rest of the garbage-collected heap. For the GC, each compact region is seen as a single heap object with a single lifetime. The GC can efficiently check whether there is at least one pointer in the garbage-collected heap that points into the region, and while this is the case, the region is kept alive. When this condition is no longer matched, the whole region is discarded. The

result is that the GC won't traverse any node from the region: it is treated as one opaque block (even though it is actually implemented as a chain of blocks of the same size, that doesn't change the principle). Also, compact regions are immobile in memory; the GC won't move them, so a destination can just be implemented as a raw pointer (type `Addr#` in Haskell): `data Dest r a = Dest Addr#`

By using compact regions to implement DPS programming, we completely elude the concerns of tension between the garbage collector and incomplete structures we want to build. Instead, we get two extra restrictions. First, every structure in a region must be in a fully-evaluated form. Regions are strict, and a heap object that is copied to a region is first forced into normal form. This might not always be a win; sometimes laziness, which is the default *modus operandi* of the garbage-collected heap, might be preferable.

Secondly, data in a region cannot contain pointers to the garbage-collected heap, or pointers to other regions: it must be self-contained. That forces us to slightly modify the API, to add a phantom type parameter `r` which tags each object with the identifier of the region it belongs to. There are two related consequences: `fillLeaf` has to copy each *leaf* value from the garbage-collected heap into the region in which it will be used as a leaf; and `fillComp` can only plug together two `Incompletes` that come from the same region.

A typeclass `Region r` is also needed to carry around the details about a region that are required for the implementation. This typeclass has a single method `reflect`, not available to the user, that returns the `RegionInfo` structure associated to identifier `r`.

The `withRegion` function is the new addition to the modified API presented in Table 6 (the `Token` type and its associated functions `dup2` and `consume` are unchanged). `withRegion` is mostly a refinement over the `withToken` function from Table 5. It receives a continuation in which `r` must be a free type variable. It then spawns both a new compact region and a fresh type `r` (not a variable), and uses the `reflection` library to provide an instance of `Region r` on-the-fly that links `r` and the `RegionInfo` for the new region, and calls the continuation at type `r`. This is fairly standard practice since [LPJ94].

5.2 Representation of Incomplete objects

Ideally, as we detailed in the API, we want `Incomplete r a b` to contain an `a` and a `b`, and let the `a` free when the `b` is fully consumed (or linearly transformed into `Ur c`). So the most straightforward implementation for `Incomplete` would be a pair `(a, b)`, where `a` in the pair is only partially complete.

It is also natural for `alloc` to return an `Incomplete r a (Dest a)`: there is nothing more here than an empty memory cell (named *root receiver*) of type `a` which the associated destination of type `Dest a` points to, as presented in Figure 1. A bit like the identity function, whatever goes in the hole is exactly what will be retrieved in the `a` side.

If `Incomplete r a b` is represented by a pair `(a, b)`, then the root receiver should be the first field of the pair. However, the root receiver must be in the region, otherwise the GC might follow the garbage pointer that lives inside; whereas the `Incomplete` wrapper must be in the garbage-collected heap so that it can sometimes be optimized away by the compiler, and always deallocated as soon as possible.

One potential solution is to represent `Incomplete r a b` by a pair `(Ur a, b)` where `Ur` is allocated inside the region and its field `a` serves as the root receiver. With this approach, the issue of `alloc` representation is solved, but every `Incomplete` will now allocate a few words in the region (to host the `Ur` constructor) that won't be collected by the GC for a long time even if the parent `Incomplete` is collected. This makes `intoIncomplete` quite inefficient memory-wise too, as the `Ur` wrapper is useless for already complete structures.

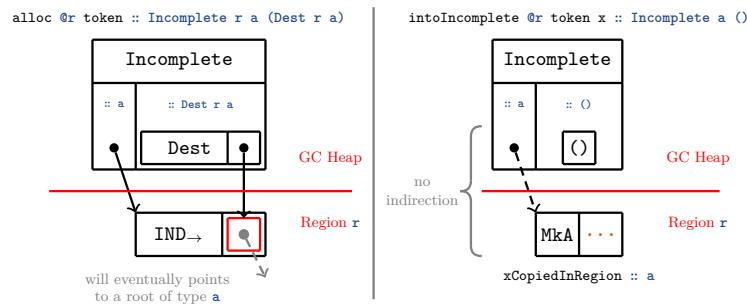
The desired outcome is to only allocate a root receiver in the region for actual incomplete structures, and skip that allocation for already complete structures that are turned into an `Incomplete` object, while preserving a same type for both use-cases. This is made possible by replacing the `Ur` wrapper inside the `Incomplete` by an indirection object (`stg_IND` label)

```

1 type Region r :: Constraint
2 withRegion :: ∀ a. (∀ r. Region r ⇒ Token → Ur a) → Ur a
3
4 data Incomplete r a b
5 fmap      :: ∀ r a b c. (b → c) → Incomplete r a b → Incomplete r b c
6 alloc     :: ∀ r a.   Region r ⇒ Token → Incomplete r a (Dest r a)
7 intoIncomplete :: ∀ r a.   Region r ⇒ Token → a → Incomplete r a ()
8 fromIncomplete_ :: ∀ r a.   Region r ⇒ Incomplete r a () → Ur a
9 fromIncomplete  :: ∀ r a b.  Region r ⇒ Incomplete r a (Ur c) → Ur (a, c)
10
11 data Dest r a
12 type family DestsOf lCtor r a
13 fill      :: ∀ lCtor r a. Region r ⇒ Dest r a → DestsOf lCtor r a
14 fillComp :: ∀ r a b.   Region r ⇒ Incomplete r a b → Dest r a → b
15 fillLeaf :: ∀ r a.     Region r ⇒ a → Dest r a → ()

```

Table 6. Destination API using compact regions

Figure 8. Memory behaviour of `alloc` and `intoIncomplete` in the region implementation

for the actually-incomplete case. `Incomplete r a b` will be represented by a pair (a, b) allocated in the garbage-collected heap, with slight variations as illustrated in Figure 8:

- in the pair (a, b) returned by `alloc`, the `a` side points to an indirection object (a sort of constructor with one field, whose resulting type `a` is the same as the field type `a`), that is allocated in the region, and serves as the root receiver;
- in the pair (a, b) returned by `intoIncomplete`, the `a` side directly points to the object of type `a` that has been copied to the region.

The implementation of `fromIncomplete_` is then relatively straightforward. It allocates a hollow `Ur` \square in the region, writes the address of the complete structure into it, and returns the `Ur` (an alternative would have been to use a regular `Ur` allocated in the GC heap).

5.3 Deriving `fill` for all constructors with Generics

The `fill @lCtor @r @a` function should plug a new hollow constructor `Ctor \square :: a` into the hole of an existing incomplete structure, and return one destination object per new hole in the structure (corresponding to the unspecified fields of the new hollow constructor). Naively, we would need one `fill` function per constructor, but that cannot be realistically implemented. Instead, we have to generalize all `fill` functions into a typeclass `Fill lCtor a`, and derive an instance of the typeclass (i.e. implement `fill`) generically for any constructor, based only on statically-known information about that constructor.

In Section 5.4, we will see how to allocate a hollow heap object for a specified constructor (which is known at compile-time). The only other information we need to implement `fill` generically is the shape of the constructor, and more precisely the number and type of its

fields. So we will leverage `GHC.Generics` to find the required information.

`GHC.Generics` is a built-in Haskell library that provides compile-time inspection of a type metadata through the `Generic` typeclass: list of constructors, their fields, memory representation, etc. And that typeclass can be derived automatically for any type! Here's, for example, the `Generic` representation of `Maybe a`:

```
1 repl> :k! Rep (Maybe a) () -- display the Generic representation of Maybe a
2 M1 D (MetaData "Maybe" "GHC.Maybe" "base" False) (
3   M1 C (MetaCons "Nothing" PrefixI False) U1
4   :+: M1 C (MetaCons "Just" PrefixI False) (M1 S [...] (K1 R a)))
```

We see that there are two different constructors (indicated by `M1 C ...` lines): `Nothing` has zero fields (indicated by `U1`) and `Just` has one field of type `a` (indicated by `K1 R a`).

With a bit of type-level programming³, we can extract the parts of that representation which are related to the constructor `lCtor` and use them inside the instance head of `Fill lCtor a` so the implementation of `fill` can depend on them. That's how we can give the proper types to the destinations returned by that function for a specified constructor. The `DestsOf lCtor a :: Type` type family also uses the generic representation of `a` to extract what it needs to know about `lCtor` and its fields.

5.4 Changes to GHC internals and RTS

We will see here how to allocate a hollow heap object for a given constructor, but let's first take a detour to give more context about the internals of the compiler.

Haskell's runtime system (RTS) is written in a mix of C and C--. The RTS has many roles, among which managing threads, organizing garbage collection or managing compact regions. It also defines various primitive operations, named *external primops*, that expose the RTS capabilities as normal functions. Despite all its responsibilities, however, the RTS is not responsible for the allocation of normal constructors (built in the garbage-collected heap). One reason is that it doesn't have all the information needed to build a constructor heap object, namely, the info table associated to the constructor.

The info table is what defines both the layout and behavior of a heap object. All heap objects representing a same constructor (let's say `Just`) have the same info table, even when the associated types are different (e.g. `Maybe Int` and `Maybe Bool`). Heap objects representing this constructor point to a label `<ctor>_con_info` that will be later resolved by the linker into an actual pointer to the shared info table.

The RTS is in fact a static piece of code that is compiled once when GHC is built. So the RTS has no direct way to access the information emitted during the compilation of a program. In other words, when the RTS runs, it has no way to inspect the program that it runs and info table labels have long been replaced by actual pointers so it cannot find them itself. But it is the one which knows how to allocate space inside a compact region.

As a result, I need to add two new primitives to GHC to allocate a hollow constructor:

- one *external primop* to allocate space inside a compact region for a hollow constructor. This primop has to be implemented inside the RTS for the aforementioned reasons;
- one *internal primop* (internal primops are macros which generates C-- code) that will be resolved into a normal albeit static value representing the info table pointer of a given constructor. This value will be passed as an argument to the external primop.

All the alterations to GHC that will be showed here are available in full form in [Bag23a].

External primop: allocate a hollow constructor in a region The implementation of the external primop is presented in Table 7. The `stg_compactAddHollowzh` function (whose equivalent on the Haskell side is `compactAddHollow#`) is mostly a glorified call to

³see `src/Compact/Pure/Internal.hs:418` in [Bag23b]

the `ALLOCATE` macro defined in the `Compact.cmm` file, which tries to do a pointer-bumping allocation in the current block of the compact region if there is enough space, and otherwise add a new block to the region.

As announced, this primop takes the info table pointer of the constructor to allocate as its second parameter (`w_info`) because it cannot access that information itself. The info table pointer is then written to the first word of the heap object in the call to `SET_HDR`.

Internal primop: reify an info table label into a runtime value The only way, in Haskell, to pass a constructor to a primop so that the primop can inspect it, is to lift the constructor into a type-level literal. It's common practice to use a `Proxy a` (the unit type with a phantom type parameter) to pass the type `a` as an input to a function. Unfortunately, due to a quirk of the compiler, primops don't have access to the type of their arguments. They can, however, access their return type. So I'm using a phantom type `InfoPtrPlaceholder# a` as the return type, to pass the constructor as an input!

The gist of this implementation is presented in Table 8. The primop `reifyInfoPtr#` pattern-matches on the type `resTy` of its return value. In the case it reads a string literal, it resolves the primop call into the label `stg_<name>` (this is used in particular to retrieve `stg_IND` to allocate indirection heap objects). In the case it reads a lifted data constructor, it resolves the primop call into the label which corresponds to the info table pointer of that constructor. The returned `InfoPtrPlaceholder# a` can later be converted back to an `Addr#` using the `unsafeCoerceAddr` function.

As an example, here is how to allocate a hollow `Just` constructor in a compact region:

```
1 hollowJust :: Maybe a = compactAddHollow#
2   compactRegion#
3   (unsafeCoerceAddr (reifyInfoPtr# (# #) :: InfoPtrPlaceholder# 'Just ))
```

Built-in type family to go from a lifted constructor to the associated symbol The internal primop `reifyInfoPtr#` that we introduced above takes as input a constructor lifted into a type-level literal, so this is also what `fill` will use to know which constructor it should operate with. But `DestsOf` have to find the metadata of a constructor in the `Generic` representation of a type, in which only the constructor name appears.

So we added a new type family `LCtorToSymbol` inside GHC that inspects its (type-level) parameter representing a constructor, fetches its associated `DataCon` structure, and returns a type-level string (kind `Symbol`) carrying the constructor name, as presented in Table 9.

6 Evaluating the performance of DPS programming

Benchmarking methodology All over this article, I talked about programs in both naive style and DPS style. With DPS programs, the result is stored in a compact region, which also forces strictness i.e. the structure is automatically in fully evaluated form.

For naive versions, we have a choice to make on how to fully evaluate the result: either force each chunk of the result inside the GC heap (using `Control.DeepSeq.force`), or copy the result in a compact region that is strict by default (using `Data.Compact.compact`).

In programs where there is no particular long-lived piece of data, having the result of the function copied into a compact region isn't particularly desirable since it will generally inflate memory allocations. So we use `force` to benchmark the naive version of those programs (the associated benchmark names are denoted with a "*" suffix).

Concatenating lists and difference lists We compared three implementations. `foldr (++)*` has calls to `(++)` nested to the right, giving the most optimal context for list concatenation (it should run in $\mathcal{O}(n)$ time). `foldl' concatλ*` uses function-backed


```

1 // compactAddHollow#
2 // :: Compact# → Addr# → State# RealWorld → (# State# RealWorld, a #)
3 stg_compactAddHollowzh(P_ compact, W_ info) {
4     W_pp, ptrs, nptrs, size, tag, hp;
5     P_to, p; p = NULL; // p isn't actually used by ALLOCATE macro
6     again: MAYBE_GC(again); STK_CHK_GEN();
7
8     pp = compact + SIZEOF_StgHeader + OFFSET_StgCompactNFData_result;
9     ptrs = TO_W_(%INFO_PTRS(%STD_INFO(info)));
10    nptrs = TO_W_(%INFO_NPTRS(%STD_INFO(info)));
11    size = BYTES_TO_WDS(SIZEOF_StgHeader) + ptrs + nptrs;
12
13    ALLOCATE(compact, size, p, to, tag);
14    P_[pp] = to;
15    SET_HDR(to, info, CCS_SYSTEM);
16    #if defined(DEBUG)
17    ccall verifyCompact(compact);
18    #endif
19    return (P_[pp]);
20 }

```

Table 7. compactAddHollow# implementation in rts/Compact.cmm

```

1 case primop of
2   [...]
3   ReifyStgInfoPtrOp → \_ → -- we don't care about the function argument (# #)
4     opIntoRegsTy $ \[res] resTy → emitAssign (CmmLocal res) $ case resTy of
5       -- when 'a' is a Symbol, and extracts the symbol value in 'sym'
6       TyConApp_addrLikeTyCon [_typeParamKind, LitTy (StrTyLit sym)] →
7         CmmLit (CmmLabel (
8           mkCmmInfoLabel rtsUnitId (fsLit "stg_" `appendFS` sym)))
9       -- when 'a' is a lifted data constructor, extracts it as a DataCon
10      TyConApp_addrLikeTyCon [_typeParamKind, TyConApp tyCon _]
11      | Just dataCon ← isPromotedDataCon_maybe tyCon →
12        CmmLit (CmmLabel (
13          mkConInfoTableLabel (dataConName dataCon) DefinitionSite))
14      _ → [...] -- error when no pattern matches

```

Table 8. reifyInfoPtr# implementation in compiler/GHC/StgToCmm/Prim.hs

```

1 matchFamLctorToSymbol :: [Type] → Maybe (CoAxiomRule, [Type], Type)
2 matchFamLctorToSymbol [kind, ty]
3   | TyConApp tyCon _ ← ty, Just dataCon ← isPromotedDataCon_maybe tyCon =
4     let symbolLit = (mkStrLitTy . occNameFS . occName . getName $ dataCon)
5         in Just (axLctorToSymbolDef, [kind, ty], symbolLit)
6 matchFamLctorToSymbol tys = Nothing
7
8 axLctorToSymbolDef =
9   mkBinAxiom "LctorToSymbolDef" typeLctorToSymbolTyCon Just
10  (\case { TyConApp tyCon _ → isPromotedDataCon_maybe tyCon ; _ → Nothing })
11  (\_ dataCon → Just (mkStrLitTy . occNameFS . occName . getName $ dataCon))

```

Table 9. LctorToSymbol implementation in compiler/GHC/Builtin/Types/Literal.hs

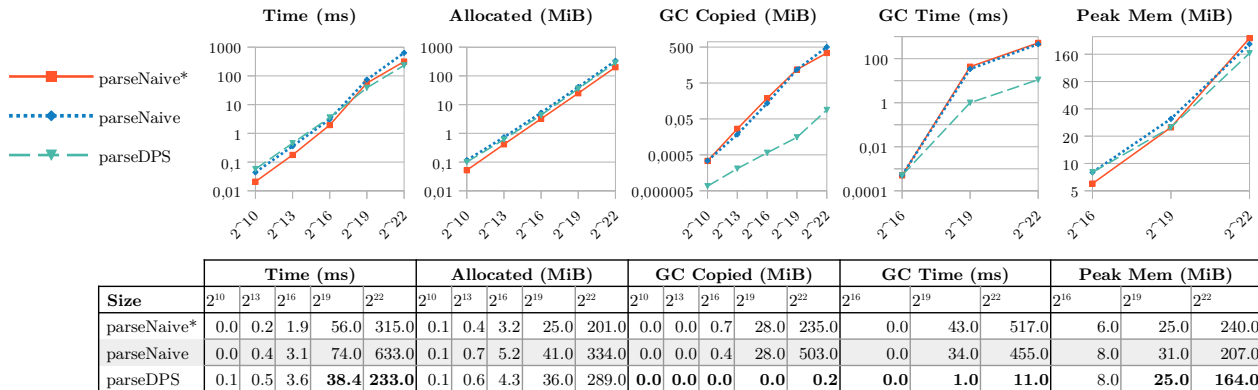
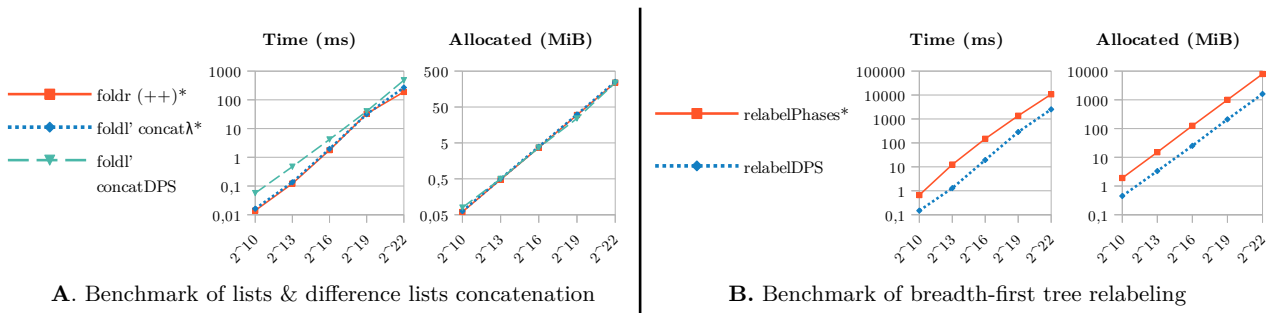


Figure 9. Benchmarks performed on AMD EPYC 7401P @ 2.0 GHz (single core, -N1 -O2)

difference lists, and `foldl' concatDPS` uses destination-backed ones, so both should run in $\mathcal{O}(n)$ even if calls to `concat` are nested to the left.

We see in part **A** of Figure 9 that the destination-backed difference lists have a comparable memory use as the two other linear implementations, while being quite slower (by a factor 2-4) on all datasets. We would expect better results though for a DPS implementation outside of compact regions because those cause extra copying.

Breadth-first relabeling We see in part **B** of Figure 9 that the destination-based tree traversal is almost one order of magnitude more efficient, both time-wise and memory-wise, compared to the implementation based on *Phases* applicatives presented in [GKSW23].

Parsing S-expressions In part **C** of Figure 9, we compare the naive implementation of the S-expression parser and the DPS one (see Section 3.3). For this particular program, where using compact regions might reduce the future GC load of the application, it is relevant to benchmark the naive version twice: once with `force` and once with `compact`.

The DPS version starts by being less efficient than the naive versions for small inputs, but gets an edge as soon as garbage collection kicks in (on datasets of size $\leq 2^{16}$, no garbage collection cycle is required as the heap size stays small).

On the largest dataset ($2^{22} \simeq 4\text{MiB}$ file), the DPS version still makes about 45% more allocations than the starred naive version, but uses 35% less memory at its peak, and more importantly, spends $47\times$ less time in garbage collection. As a result, the DPS version only takes $0.55\text{-}0.65\times$ the time spent by the naive versions, thanks to garbage collection savings. All of this also indicates that most of the data allocated in the GC heap by the DPS version just lasts one generation and thus can be discarded very early by the GC, without needing to be copied into the next generation, unlike most nodes allocated by the naive versions.

Finally, copying the result of the naive version to a compact region (for future GC savings) incurs a significant time and memory penalty, that the DPS version offers to avoid.

7 Related work

The idea of functional data structures with write-once holes is not new. Minamide already proposed in [Min98] a variant of λ -calculus with support for *hole abstractions*, which can be represented in memory by an incomplete structure with one hole and can be composed efficiently with each other (as with `fillComp` in Figure 4). With such a framework, it is fully possible to implement destination-backed difference lists for example.

However, in Minamide’s work, there is no concept of destination: the hole in a structure can only be filled if one has the structure itself at hand. On the other hand, our approach introduces destinations, as a way to interact with a hole remotely, even when one doesn’t have a handle to the associated structure. Because destinations are first-class objects, they can be passed around or stored in collections or other structure, while preserving memory safety. This is the major step forward that our paper presents.

More recently, [PP13] introduced the Mezzo programming language, in which mutable data structures can be freed into immutable ones after having been completed. This principle is used to some extent in their list standard library module, to mimic a form of DPS programming. An earlier appearance of DPS programming as a mean to achieve better performance in a mutable language can also be seen in [Lar89].

Finally, both [SFPJV17] and [BCS21] use DPS programming to make list or array processing algorithms more efficient in a functional, immutable context, by turning non tail-recursive functions into tail-recursive DPS ones. More importantly, they present an automated way to go from a naive program to its tail-recursive version. However, holes/destinations are only supported at an intermediary language level, while both [Min98] and our present work support safe DPS programming in user-land. In a broader context, [LLS23] presents a system in which linearity is used to identify where destructive updates can be made, so as to reuse the same constructor instead of deallocating and reallocating one; but this optimization technique is still mostly invisible for the user, unlike ours which is made explicit.

8 Conclusion and future work

Programming with destinations definitely has a place in the realm of functional programming, as the recent adoption of *Tail Modulo Cons* [BCS21] in the OCaml compiler shows. In this paper, we have shown how destination-passing style programming can be used in user-land in Haskell safely, thanks to a linear type discipline. Adopting DPS programming opens the way for more natural and efficient programs in a variety of contexts, where the major points are being able to build structures in a top-down fashion, manipulating and composing incomplete structures, and managing holes in these structures through first-class objects (destinations). Our DPS implementation relies only on a few alterations to the compiler, thanks to *compact regions* that are already available as part of GHC. Simultaneously, it allows to build structures in those regions without copying, which wasn’t possible before.

There are two limitations that we would like to lift in the future. First, DPS programming could be useful outside of compact regions: destinations could probably be used to manipulate the garbage-collected heap (with proper read barriers in place), or other forms of secluded memory areas that aren’t traveled by the GC (RDMA, network serialized buffers, etc.). Secondly, at the moment, the type of `fillLeaf` implies that we can’t store destinations (which are always linear) in a difference list implemented as in Section 3.1, whereas we can store them in a regular list or queue (like we do, for instance, in Section 3.2). This unwelcome restriction ensures memory safety but it’s quite coarse grain. In the future we’ll be trying to have a more fine-grained approach that would still ensure safety.

References

- [Bag23a] BAGREL, THOMAS: GHC with support for hollow constructor allocation. Software Heritage, swh:1:dir:84c7e717fd5f189c6b6222e0fc92d0a82d755e7c; origin=https://github.com/tweag/ghc; visit=swh:1:snp:141fa3c28e01574deebb6cc91693c75f49717c32; anchor=swh:1:rev:184f838b352a0d546e574bdeb83c8c190e9dfdc2, 2023. Accessed: 2023-10-19.
- [Bag23b] BAGREL, THOMAS: `linear-dest`, a Haskell library that adds supports for DPS programming. Software Heritage, swh:1:rev:0e7db2e6b24aad348837ac78d8137712c1d8d12a; origin=https://github.com/tweag/linear-dest; visit=swh:1:snp:c0eb2661963bb176204b46788f4edd26f72ac83c, 2023. Accessed: 2023-10-19.
- [BBN⁺18] Jean-Philippe BERNARDY, Mathieu BOESPFLUG, Ryan R. NEWTON, Simon Peyton JONES and Arnaud SPIWACK: Linear Haskell: practical linearity in a higher-order polymorphic language. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–29, January 2018. arXiv:1710.09756 [cs].
- [BCS21] Frédéric BOUR, Basile CLÉMENT and Gabriel SCHERER: Tail Modulo Cons. *arXiv:2102.09823 [cs]*, February 2021. arXiv: 2102.09823.
- [Gib93] Jeremy GIBBONS: Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips. (No. 71), 1993. Number: No. 71.
- [Gir95] J.-Y. GIRARD: Linear Logic: its syntax and semantics. In Jean-Yves GIRARD, Yves LAFONT and Laurent REGNIER, éditeurs: *Advances in Linear Logic*, pages 1–42. Cambridge University Press, Cambridge, 1995.
- [GKSW23] Jeremy GIBBONS, Donnacha Oisín KIDNEY, Tom SCHRIJVERS and Nicolas WU: Phases in Software Architecture. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Functional Software Architecture*, pages 29–33, Seattle WA USA, August 2023. ACM.
- [Lar89] James Richard LARUS: *Restructuring symbolic programs for concurrent execution on multiprocessors*. phd, University of California, Berkeley, 1989. AAI9006407.
- [LLS23] Anton LORENZEN, Daan LEIJEN and Wouter SWIERSTRA: FP²: Fully in-Place Functional Programming. *Proceedings of the ACM on Programming Languages*, 7(ICFP):275–304, August 2023.
- [LPJ94] John LAUNCHBURY and Simon L. PEYTON JONES: Lazy functional state threads. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation, PLDI '94*, pages 24–35, New York, NY, USA, June 1994. Association for Computing Machinery.
- [Min98] Yasuhiko MINAMIDE: A functional representation of data structures with a hole. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, POPL '98*, pages 75–84, New York, NY, USA, January 1998. Association for Computing Machinery.
- [Oka00] Chris OKASAKI: Breadth-first numbering: lessons from a small exercise in algorithm design. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, ICFP '00*, pages 131–136, New York, NY, USA, September 2000. Association for Computing Machinery.

- [PP13] Jonathan PROTZENKO and François POTTIER: Programming with Permissions in Mezzo. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, pages 173–184, September 2013. arXiv:1311.7242 [cs].
- [SFPJV17] Amir SHAIKHHA, Andrew FITZGIBBON, Simon PEYTON JONES and Dimitrios VYTINIOTIS: Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*, pages 12–23, Oxford UK, September 2017. ACM.
- [SKB⁺22] Arnaud SPIWACK, Csongor KISS, Jean-Philippe BERNARDY, Nicolas WU and Richard A. EISENBERG: Linearly qualified types: generic inference for capabilities and uniqueness. *Proceedings of the ACM on Programming Languages*, 6(ICFP): 95:137–95:164, August 2022.
- [YCA⁺15] Edward Z. YANG, Giovanni CAMPAGNA, Ömer S. AACAN, Ahmed EL-HASSANY, Abhishek KULKARNI and Ryan R. NEWTON: Efficient communication and collection with compact normal forms. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, pages 362–374, Vancouver BC Canada, August 2015. ACM.

Modular efficient deconstruction with typed pointer reversal

Jean Caspar¹ and Guillaume Munch-Maccagnoni²

¹École normale supérieure – PSL university

²INRIA, LS2N, Nantes

Abstract

Destructors, responsible for releasing memory and other resources in languages such as C++ and Rust, can lead to stack overflows when releasing a recursive structure that is too deep. In certain cases, it is possible to generate an efficient destructor (non-allocating and tail recursive) using a typed variant of pointer reversal. We extend this technique by making it more modular, in order to handle abstract types, separate compilation, and unboxed types.

1 Introduction

In some programming languages, such as Rust and C++, memory management is performed by the compiler via the insertion of some code at the end of the scope of values, responsible for releasing the memory allocated on the heap. Such code is called a destructor. Since they run predictably and reliably when variables get out of scope, destructors can be used to reason about resources other than memory, such as files, shared data protected by a mutex, network connections, or transactions. The use of destructors as a resource-handling mechanism is called RAII (*Resource Acquisition Is Initialization*) [KS90].

The stack overflow problem In RAII-based languages, destructors recursively call the destructors of each sub-field; therefore, if the type they operate on is recursive, then the destructor should be recursive as well. In the evolution of this model, as incarnated by the programming languages C++11 and Rust, compiler-generated destructors themselves are potentially recursive (via “smart pointers”), and their naive implementations by contemporary compilers potentially cause stack overflows. It was believed that one cannot implement destructors in general without allocating on the stack or on the heap, or without changing the order and time of destruction, as Herb Sutter explained [Sut16]. The fact that generated destructors themselves can overflow the stack makes it a problem of compiler correctness.

Typed pointer reversal Given that the programmer can supply an arbitrary non-raising function acting as a destructor on a per-type basis, generated destructors must take programmer-provided destructors into account, and are specific to a given type.

Continuing previous work by Douence and the second author [MMD19], the subject of this paper is to derive efficient and correct destructors for recursive types, starting from their specifications (that is, their naive implementations) and applying provably-correct

transformations. As shown previously, it is indeed possible to obtain implementations which are tail-recursive and which do not allocate, by starting from the naive implementation and applying standard transformations: namely a continuation-passing style (CPS) transformation followed by defunctionalization [Rey72, Wan80, DN01], and a transformation to re-use the memory of the (unaliased) value as a way to store the defunctionalized continuations [Laf88, Bak92]. A dual transformation, about recycling continuations into data, had also been observed earlier by Sobel and Friedman [SF98]. In effect, this transformation amounts to a typed generalization of pointer reversal [SW67], a graph traversal algorithm developed for garbage collection, notorious for being hard to get right and to reason about.

In particular, the resulting implementation is equivalent to the naive implementation, but it does not overflow the stack. (Another benefit which we can expect is slightly improved performance due to better cache locality, but we do not propose benchmarks in this paper.)

This paper assumes familiarity with neither CPS nor defunctionalization. We directly give the general shape of the resulting implementation below in Section 2, which we use as a starting point.

Limitations to its applicability & goals The typed pointer reversal method suffers from limitations if we want to use it in the construction of compilers:

- It does not handle abstract types: if a custom destructor supplied for a type is recursive, then it is not clear how to apply the transformation. Therefore, this method cannot be used by the compiler to automatically derive efficient destructors.
- It does not handle separate compilation: if the specification of a destructor is unknown, even if its efficient implementation is provided, then we cannot derive another efficient destructor that calls this destructor, because the method is not compositional.
- It does not handle unboxed types: currently, it assumes that all values are boxed behind a pointer (Lisp-like representation of values), whereas unboxed values (C-like representation of values, as used in Rust and C++) are not handled.

In this paper, we show that it is possible to extend typed pointer reversal by making it modular, in order to handle abstract types and separate compilation (Section 3), and we sketch an application to unboxed types (Section 5).

Since pointer reversal is tricky to implement by hand and to prove correct, we ideally want to prove that the new transformations themselves are correct, and maybe relate them to ones that have already been studied; as far as the present work is concerned this is work in progress.

Throughout the paper, we propose examples and descriptions in an ML-like syntax. There are two reasons for this choice:

1. the Lisp-like memory representation is simple, so is easy to start with, and there are additional issues with the C-like memory representation of Rust and C++ that we will not touch upon before the later parts of the paper;
2. it is possible to imagine extensions of ML that incorporate RAII-based resource-management [MM23]. In this case, the challenges go beyond C++ and Rust, as witnessed by examples from Section 4 involving elaborate forms of polymorphism.

Example: B-trees The B-tree data structure constitutes a good example of the limitations of the approach from the previous work. A B-tree is a tree where each node can have a non-bounded number of children, and holds a non-bounded number of key-value pairs. It can be defined in ML as follows:

```
type 'a btree = Leaf of (int * 'a) array | Node of (int * 'a) array * 'a btree array
```

This type uses arrays—values whose size is not known during compilation. This is not

supported by typed pointer reversal as we described so far, since `array` is not an algebraic data type. In addition, an efficient destructor for `array` would be of no help in letting us deal with a recursion occurring via its parameter as in its third occurrence above. In fact, the naive destructor for arrays, which iterates on the elements, is already an efficient destructor (since `array` is not recursive on its own).

Furthermore, we want to be able to derive an efficient destructor for B-trees, even when the implementations of arrays and of its destructor are opaque (for instance opaque to the compiler, when arrays are defined in another compilation unit). This might actually be of a lesser concern in languages that are not so strict about separate compilation, such as C++ and Rust. Lastly, in languages such as C++ and Rust, where we would substitute vectors for arrays, the vectors would be unboxed (as triplets consisting of a size, a capacity, and a pointer to a backing array).

All in all, if we want to apply typed pointer reversal to the implementation of programming languages, we need to determine how to handle abstract types, separate compilation, and unboxed types.

2 On efficient destructors for algebraic data types

2.1 Generic shape for typed pointer reversal

To start from where Douence and the second author [MMD19] left off, we first describe a generic form of the efficient destructors obtained with their method, for algebraic data types of the following generic shape:

$$\begin{aligned} A_i &= B_{i,1}(A_1, \dots, A_n) + \dots + B_{i,m_i}(A_1, \dots, A_n) & 1 \leq i \leq n \\ B_{i,j}(A_1, \dots, A_n) &= C_{i,j,1} \times \dots \times C_{i,j,l_{i,j}} & 1 \leq i \leq n, 1 \leq j \leq m_i \end{aligned}$$

where $C_{i,j,k}$ is either of the form $A_{i'}$, or some other type which does not contain any $A_{i'}$. In the latter case, the destructors are assumed given (as `drop_C_{i,j,k}`), and possibly arbitrary.

The specification of the destructor is to traverse the structure depth-first, from left to right, calling the destructors of the $C_{i,j,k}$, and freeing the memory of each node it exits—as follows:

```
(* for all i: *)
let rec drop_A_i t = match t with
(* for all j: *)
| B_{i,j}(c_1, ..., c_l) -> drop_C_{i,j,1} c_1; ...; drop_C_{i,j,l} c_l; free t
```

Above, recursion takes place when $C_{i,j,k}$ is of the form $A_{i'}$.

Applying the CPS transformation yields tail-recursive implementation of destructors, with an accumulator (the continuation), so as to avoid consuming stack space. Then, through defunctionalization, the continuation becomes a first-order data structure. We call these implementations of destructors `drop_iter_A_i`. Then, `drop_A_i` is equivalent to `drop_iter_A_i` called with initial continuation.

The type of defunctionalized continuations has the form $\text{cont} = 1 + \sum_{i,j,k} K_{i,j,1}^k \times \dots \times K_{i,j,l_{i,j}}^k$ where the sum is indexed by all tuples (i, j, k) where $C_{i,j,k}$ is of the form $A_{i'}$ for some i' , and where $K_{i,j,k'}^k = \text{cont}$ whenever $k' \leq k$ and $C_{i,j,k}$ is of the form $A_{i'}$, and $K_{i,j,k'}^k = C_{i,j,k}$ otherwise.

After the transformation, we obtain the functions from Listing 1 below, written in ML-like pseudo-code plus explicit freeing of memory. We write `K_{i,j,k}` the constructor of the variant corresponding to (i, j, k) in the sum cont , and `Empty` the constructor for 1. In addition, we added the annotation `@` on constructors to denote that a memory cell can be re-used, as explained next.


```

(* for all i: *)
let rec drop_iter_A_i t cont = match t with
(* for all j: *)
| B_{i,j}(c_1, ..., c_l) ->
  (* We statically find the least k such that C_{i,j,k} is of the form A_{i'}. *)
  (* If k exists: *)
  drop_C_{i,j,1} c_1; ... drop_C_{i,j,k-1} c_{k-1};
  drop_iter_A_{i'} c_k (K_{i,j,k}@t (cont, c_{k+1}, ..., c_l))
  (* If k does not exist: *)
  drop_C_{i,j,1} c_1; ... drop_C_{i,j,l} c_l; free t; invoke cont
| ...

and invoke = function
(* for all i, j and k: *)
| K_{i,j,k}(cont, c_{k+1}, ..., c_l) as m ->
  (* We statically find the least k' > k such that C_{i,j,k'} is of the form
  A_{i'}. *)
  (* If k' exists: *)
  drop_C_{i,j,k+1} c_{k+1}; ... drop_C_{i,j,k'-1} c_{k'-1};
  drop_iter_A_{i'} c_{k'} (K_{i,j,k'}@m (cont, c_{k'+1}, ..., c_l))
  (* If k' does not exist: *)
  drop_C_{i,j,k+1} c_{k+1}; ... drop_C_{i,j,l} c_l; free k; invoke cont
| ...
| Empty -> ()

let drop_A_i a = drop_iter_A_i a Empty

```

Listing 1. A generic efficient destructor for algebraic data types.

The last part of the transformation involves re-using memory to store the continuations, so as to avoid allocating on the heap. We use the notation $K@v(a, b, \dots)$ above to mean that $K(a, b, \dots)$ can be allocated by reusing the memory cell of the value v . This involves mutating both the fields and the tag of v .

There are two key observations to make here. First, notice that t is no longer used in the rest of the destructor; it also cannot be used later in the program, since the destructor eventually frees its memory. Second, the new continuation always consists of a strict suffix of the previous value, plus a field to store the previous continuation. In particular, the re-used value is large enough, and at most one field needs to be mutated.

As written down, it is manifest that this implementation is tail-recursive and does not allocate.

2.2 Possible improvements

The previous section assumes that there is no limit on the number of variants. But there can be more variants for the continuation than there is in the original type. In practice, this can introduce additional constraints for the memory representation if values.

In order to bound the number of variants, the first observation is that we can reduce the amount of variants for the type cont to at most two for each pair (i, j) . Indeed, from the second field being destroyed onwards, a second cell is available, which can store an additional integer. The k distinct variants can thus be replaced by two variants plus an auxiliary integer ranging from 2 to k .

We can further extend this solution to recursive types containing arrays (i.e. some of the $C_{i,j,k}$ being arrays possibly containing some A_i s). As above, after the first element has been destroyed, we store an integer in the first field, which indicates how many fields have been destroyed so far.

2.3 Example: B-trees

Having just explained how to extend the method to arrays, let us go back to the example of B-trees. In Listing 2 we provide a destructor for a simplified version of the B-tree. The keyword `cast` is used below to coerce the type of values in an unsafe way. We need two variants in `cont` in order to deal with the array. The first one, `K`, allows us to drop its first element, and the store the continuation at its place. The second one, `K'`, takes advantage of the fact that there is at least two free memory cells and store in the second one the number of cells already dropped. The variants `Leaf` and `Node` contain exactly zero or one element, thus we can further optimize and not adding variants for them inside `cont`.

```

type btree = Leaf | Node of btree array
type cont = Empty | K of cont | K' of cont * int

let rec drop_btree t k = match t with
| Leaf -> invoke k
| Node x -> free t; drop_array x k
and drop_array a k =
  if Array.length a = 0 then invoke k
  else let y = a.(0) in drop_btree y (K@a k)
and invoke = function
| Empty -> ()
| K k' as k ->
  let (x : btree array) = cast k in
  if Array.length x = 1 then invoke k'
  else let y = x.(1) in drop_btree y (K'@x (k', 2))
| K' (k', i) as k ->
  let (x : btree array) = cast k in
  if Array.length x = i then invoke k'
  else let y = x.(i) in drop_btree y (K'@x (k', i + 1))

```

Listing 2. Efficient B-tree destructor.

3 An interface for a modular efficient drop

While we have shown that it is possible to handle B-trees by extending the algorithm to arrays, the goal is to obtain a modular interface to handle types that could be defined as an abstract type by a library (e.g. `vector` in C++/Rust), perhaps in an opaque compilation unit.

3.1 An interface in ML-like languages

In order to handle abstract types, we propose a modular interface that any type can implement in order to provide an efficient destructor. It allows composing such destructors automatically for algebraic data types, and can be implemented by hand for more complex data types. This (unsafe) interface is given in Figure 1b in the syntax of ML modules. It can be seen as both as a description of the interface needed by a compiler to generate an efficient destructor, and a refinement of the `Drop trait` in Rust's terminology (essentially the module type in Figure 1a).

The `Drop` interface allows destructing a type `t`, and must recursively call the destructor of each of the values it contains. As we have seen so far, an implementation of `Drop` can be derived automatically for all mutually-recursive types comprised of algebraic types and arrays.

The `DropIter` interface allows implementing and composing efficient destructors; it is unsafe because the parameter `'a` represents a type of continuations, but it is challenging to describe this type in advance.

```

module type Drop = sig
  type t
  val drop : t -> unit
end

(a) Drop interface.

module type Base = sig val base : int end

module type DropIter = sig
  type t
  val nb_state : int
  module Make(B : Base) : sig
    type cont
    val drop_iter : t -> cont -> unit
    val invoke : cont -> unit
  end
end

(b) DropIter interface.

```

Figure 1. Interfaces for destructors as ML modules.

The key idea behind this interface is that each implementation will declare `nb_state` variants of a `cont` recursive type (which will never be formally declared), with these variants having a tag belonging to the interval $[\text{base}; \text{base} + \text{nb_state}]$. The implementation is parameterized by the `base` value, to be chosen such that the various implementations of `DropIter` involved in the destruction of a data structure do not interfere with each other.

The values `nb_state` and `base` are therefore constant and known at compile-time. The functions `drop_iter` and `invoke` play the same roles as in the previous sections, and can be automatically derived for algebraic data types. Their implementation for any `base` is known at compile-time.

Lastly, the implementation of a `DropIter` is generally parametrised by the `DropIter`s of its type parameters: the modular efficient drop interface for a parametrised type (`'a t`) is a functor from `DropIter` to `DropIter`.

The type `cont` represents a type of continuations, which depends on the values of `nb_state`, `base` and the other instances of `DropIter` involved. We have not yet correctly typed the continuation in this interface in any language in a way that respects the required representation of values, we relied on unsafe features instead.

In `invoke`, implementations should look at the tag of the parameter, and if it is some universal and fixed integer constant, it should return (this means the iteration reached the end). If it belongs to the interval $[\text{base}; \text{base} + \text{nb_state}]$, it should handle it appropriately (either by doing something directly or by dispatching it to a `DropIter` implementation instantiated by the current implementation). Lastly, if it belongs to one of the `DropIter` implementations by which this implementation is parameterized, it must dispatch it accordingly. In correct implementations, other cases do not happen.

Given a `DropIter` instance, one can build a `Drop` instance in the following way:

```

module DropIterToDrop(D: DropIter) : (Drop with type t = D.t) = struct
  type t = D.t
  module Impl = D.Make(struct let base = 0 end)
  let drop x = Impl.drop_iter x Empty
end

```

Here, `Empty` is the constant on which all `invoke` functions must return.

3.2 Example with a generic type

Now let us focus on a concrete example. Suppose we have a type parameterized by a type variable, for which we want to implement `DropIter`, such as the following:

```

type 'a tree = Node of 'a tree * 'a * 'a tree | Leaf

```

The type parameter is required to also have a modular efficient destructor, therefore the implementation of `DropIter` for `'a tree` is parametrised by a `DropIter` for `'a`. In the

pseudo-code below, we use ellipses to denote that the tags of variants are chosen among $[\text{base}; \text{base} + 2[$.

```

module DropIterTree (D: DropIter) : (DropIter with type t = D.t tree) = struct
  type t = D.t tree
  let nb_state = 2
  module Make(B : Base) = struct
    let d_base = B.base + nb_state
    module DImpl = D.Make(struct let base = d_base end)
    type cont = Empty
    | ...
    | KDropVal of cont * D.t * D.t tree (* with tag offset B.base *)
    | KDropRight of cont * D.t tree
    | ...

    let rec drop_iter t k = match t with
    | Leaf -> invoke k
    | Node (l, x, r) -> drop_iter l (KDropVal@t (k, x, r))
    and invoke k = match k with
    | Empty -> ()
    | KDropVal (k',x,r) -> D.drop_iter x (KDropRight@k (k',r))
    | KDropRight (k',r) -> free k; drop_iter r k'
    | K _ when d_base <= tag K && tag K < d_base+D.nb_state -> DImpl.invoke k
    | _ -> assert false
  end
end

```

Listing 3. Efficient and modular drop implementation for 'a tree.

The `KDropVal` variant represents a node whose left child has been dropped. The `KDropRight` variant represents a node whose left child and the value it holds have been dropped, but not the right child.

This functor can be automatically derived with the typed pointer reversal method (e.g. by the compiler) in the case of mutually algebraic data types and arrays, similarly to the version of section 2.

4 Limitations due to polymorphism

Nested or non-regular data types are recursive polymorphic data types that can recursively instantiate themselves with different parameters.

Common examples of such data types, as presented by Matthes [Mat08], include:

- Perfect binary trees : $\text{Tree } \alpha = \alpha + \text{Tree}(\alpha \times \alpha)$
- Bushes : $\text{Bush } \alpha = 1 + \alpha \times \text{Bush}(\text{Bush } \alpha)$
- Lambda calculus with de Bruijn indexes and explicit substitutions:
 $\text{Lam } \alpha = \alpha + \text{Lam } \alpha \times \text{Lam } \alpha + \text{Lam}(1 + \alpha) + \text{Lam}(\text{Lam } \alpha)$

If we try to apply our algorithm to derive an efficient destructor for such data types, we will be stuck because our algorithm relies on the fact that we encounter a finite number of types in the implementation of a single recursive destructor. But if we want to generate a destructor for perfect binary trees $\text{Tree } \alpha$, we will need to generate the destructor for $\text{Tree}(\alpha \times \alpha)$, $\text{Tree}((\alpha \times \alpha) \times (\alpha \times \alpha))$, and so on. This would lead to an infinite number of states, at least with this approach.

But, for such types T we can still define a less efficient destructor which is better than the naive one, by using the *drop* function for β when we try to destruct $T \beta$ with $\alpha \neq \beta$. The

stack consumption will then increase only when, while destructing a value of type $T \alpha$, we destruct a value of type $T \beta$ with $\beta \neq \alpha$; outside of that, it is constant. Therefore for a value of type $T \alpha$, the stack usage is proportional to the maximum number of edges $T \beta \rightarrow T \gamma$ where possibly $\beta \neq \gamma$ that we encounter in a path from the origin to a leaf.

This limitation does not apply to languages like C++ or Rust where all polymorphic arguments are monomorphized at compile-time, because we cannot define recursive functions on such a type. Monomorphization (substituting all generic types by a concrete type) is indeed less expressive than polymorphism à la ML.

5 Deconstructing unboxed types

5.1 Refining the interface for unboxed types

Currently, we have only considered values that are boxed, that is, are accessed through an indirection. But some languages such as C++ and Rust allow someone to manipulate unboxed values, that are laid out consecutively in memory. A problem arises when dealing with unboxed values, as we cannot overwrite the unboxed value with the continuation before we have dropped it entirely. It worked with boxed values because we have been exchanging values of the same size.

We propose in the next section a refinement of `DropIter` which drops a value of type T placed at a specific offset of a type U , rather than accessed by indirection. We deal with transitive subfields in the same way, in terms of offsets of a parent boxed type. Furthermore, a problem arises when dealing with mutually recursive types naively: if we have two mutually recursive types T and U of different unboxed sizes, then it is not clear which should be the size of the continuation.

We can address this problem by using tagged pointers for the continuation, where the tag is used to identify the continuation types before we dereference them. In typical 64-bit architecture, many bits of pointers are non-significant and can be re-used.¹ We call this kind of tagged pointer to a continuation a `ContPointer`.

5.2 Unboxing the efficient destructor

We introduce a refinement of the previous interface, with explicit pointers, in order to drop a type T that is inlined in another type U , at offset o (in bytes):

```
drop_fieldT,o,U : U* → ContPointer → 1
invoke_fieldT,o,U : ContPointer → 1
retrieve : T* → ContPointer
```

where U^* is the type of pointers of U .

We can derive an implementation of this interface called `DropFieldT,o,U` for a type T starting with a `DropIter` interface for T , under a few conditions: we require that all calls to `drop_iter` for an inlined T take a continuation of the form `KRest@u(k, ...)` where u is the pointer to U that has been recycled (that is, we do not optimize the last call of a sequence of destructors as we did in subsection 2.2). We call this transformation *unboxing*.

The function `drop_field` is essentially `drop_iter`, but where right at the beginning we retrieve a T^* by doing pointer arithmetic with the U^* pointer in argument and the offset o . We use this T^* pointer as in `drop_iter`, except that we store the tag in the pointer U^* instead of the pointer T^* ; we still recycle the memory of our T^* pointer (since we do not know the state of the rest of the structure, we can only access a portion of it).

¹For instance, on the `x86_64` architecture, one can re-use at least the lower 3 bits due to alignment, and at least 7 of the upper bits if 5-level paging is used [Int23, Vol. 3A, §4.5].

For `invoke_field`, we cast the `ContPointer` into a `U*` and also do pointer arithmetic to retrieve a `T*`. On it, we execute the same code that `invoke_iter` (the function called `invoke` formerly) executes.

But inside these two functions we change one thing: we do not free the memory used by the pointer `T*`. Furthermore, the last recursive call which calls the continuation passed as a parameter (of the form `k = KRest(k', ...)`) is changed to a call to `invoke_iter_U KEnd_{T,o,U}(k, ...)` where we recycle the `U*` pointer and store the continuation `k` somewhere in the current structure, and where `KEnd_{T,o,U}` is a new state that belongs to `U`.

And that is where the `retrieve` function comes in: it must be able to retrieve this continuation after the the destruction of the nested `T*`. The code to handle a `KEnd_{T,o,U}` is also inserted into the `invoke_iter_U` function:

```
invoke_iter_U (KEnd_T_o_U as k) =
  let u = (cast k : U*) in
  let t = (cast (u + o) : T*) in
  let k' = retrieve t in
  (* this code can be inlined and the variant KRest deleted *)
  invoke_iter_U (KRest@u(k', ...))
```

We need also to change the code calling `drop_iter` and `invoke_iter` for `T` when it is a field of `U`.

Every call to `drop_iter_T t KRest@u(k, ...)` where `t` is inlined in some structure `U` must be replaced with `drop_field_{T,o,U} u k` where `o` is the difference in bytes between `t` and `u`, and every call to `invoke_iter_T k` must be replaced with `invoke_field_{T,o,U} k`. When several types are inlined inside one another, the type `U` and the offset must be taken from the outermost type, that is, from the pointer that currently owns the allocation for the whole structure.

5.3 Example: handling vectors

The vector in C++ and Rust is an unboxed type which consists of a pointer to the start of a backing array, a capacity pointer to its end, and a length pointer that points past the last element of the vector.

If the capacity is greater than the length, then we can store the continuation at the end of the backing array. We can decrement the length pointer and drop the next element through `drop_iter` if there remain elements to destroy, or free the backing array and call the continuation otherwise. If the capacity is equal to the length, then one can first store the continuation by overwriting the capacity (since this information is redundant), while tagging the length pointer to indicate this first step taken. After the first step, the capacity pointer is restored from the length pointer, there is now room at the end, and one proceeds as above.

All this obviously relies on implementation details of vectors. The implementation of the efficient destructor therefore needs to be given by standard library implementors, via the modular efficient drop interface.

Further code and examples, including a detailed implementation for C++ vectors, are available in the public repository <https://gitlab.com/JeanCASPAR/artefacts-jfla-2023>.

6 Further work

More work is needed to formally define and prove the correctness of our transformations, especially the unboxing transformation. We would like to find more conceptual explanations of our transformations, which might simplify proofs of correctness. We would also like to clarify the constraints on the memory representation of variants, and distinguish pertinent memory representation details from irrelevant ones.

References

- [Bak92] Henry G Baker. Lively linear lisp: "look ma, no garbage!". *ACM Sigplan notices*, 27(8):89–98, 1992.
- [DN01] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at Work. Research Report BRICS RS-01-23, Department of Computer Science, Aarhus University, Aarhus, Denmark, July 2001.
- [Int23] Intel. *Intel 64 and IA-32 Architectures Software Developer’s Manual: System Programming Guide*, September 2023. Volume 3.
- [KS90] Andrew Koenig and Bjarne Stroustrup. Exception handling for C++. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 149–176, San Francisco, CA, USA, 1990.
- [Laf88] Yves Lafont. The linear abstract machine. *Theoretical computer science*, 59(1-2):157–180, 1988.
- [Mat08] Ralph Matthes. Recursion on nested datatypes in dependent type theory. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Logic and Theory of Algorithms*, pages 431–446, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [MM23] Guillaume Munch-Maccagnoni. Resource polymorphism: A proposal for integrating first-class resources into ML (abstract). In *ML Family workshop (ML’23)*, 2023.
- [MMD19] Guillaume Munch-Maccagnoni and Rémi Douence. Efficient deconstruction with typed pointer reversal (abstract). In *ML Family workshop (ML’19)*, 2019.
- [Rey72] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.
- [SF98] Jonathan Sobel and Daniel P. Friedman. Recycling continuations. *SIGPLAN Not.*, 34(1):251–260, September 1998.
- [Sut16] Herb Sutter. Leak-Freedom in C++... By Default. <https://www.youtube.com/watch?v=JfmTagWcqoE&t=?t=978>, 2016. CppCon 2016.
- [SW67] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10(8):501–506, August 1967.
- [Wan80] Mitchell Wand. Continuation-Based Program Transformation Strategies. *J. ACM*, 27(1):164–180, 1980.

PREUVE FORMELLE, INGÉNIERIE DE LA
PREUVE

Towards the Fundamental Theorem of Calculus for the Lebesgue Integral in Coq

Reynald Affeldt^{1,2} and Zachary Stone²

¹National Institute of Advanced Industrial Science and Technology (AIST), Tokyo, Japan

²The MathComp-Analysis development team

We report on an ongoing formalization of the Fundamental Theorem of Calculus (FTC) for the Lebesgue integral in the Coq proof assistant. To this aim, we formalize Lebesgue’s differentiation theorem, which has other applications and whose proof can be decomposed in lemmas of independent interest. As a result, we significantly enrich the theories of MathComp-Analysis, an extension of the Mathematical Components library for analysis, and eventually relate its definitions of derivability and of integration.

Motivation and contributions

The formalization of the Fundamental Theorem of Calculus (FTC) for the Lebesgue integral in the COQ proof assistant is an ongoing work part of the development of MATHCOMP-ANALYSIS [ABC⁺23], a library for analysis that extends the Mathematical Components library [MT21] with classical axioms [ACR18, §5]. Besides mathematics, MATHCOMP-ANALYSIS has also been used to formalize programming languages [ACS23, ZBS⁺23, SA23].

The first FTC for Lebesgue integration can be stated as follows: for f integrable on \mathbb{R} , $F(x) \stackrel{\text{def}}{=} \int_{t \in]-\infty, x]} f(t)(\mathbf{d}\mu)$ is derivable and $F'(x) = f(x)$ a.e. (i.e., almost-everywhere) relatively to μ , which is the Lebesgue measure. This is different from the statement for the Riemann integral, where f is usually assumed to be continuous, making for a simple proof. In comparison, connecting derivation and Lebesgue integration under an integrability hypothesis is unwieldy, even more so in MATHCOMP-ANALYSIS whose formalizations of derivation [ACR18, §4.5] and of the Lebesgue integral [AC23, §6.4] have been unrelated so far. They can be bridged thanks to the Lebesgue differentiation theorem and this is appealing for two reasons. First, it is a useful theorem in itself: the FTC is a consequence, as well as other results such as Lebesgue’s density theorem. Second, we can decompose its proof in several results: this provides a nice way to incrementally enrich the theories of MATHCOMP-ANALYSIS. We think that this approach is an instance of a more generic way to tackle formalization of mathematics: find a path through the literature to present many key results as easy consequences of a central, technical lemma with rather weak assumptions.

In terms of formalization in a proof assistant, our contributions are as follows. We provide the first formalization of the first FTC for Lebesgue integration in COQ. We also bring to COQ standard results of measure theory and topology: Vitali’s lemmas and theorem, Hardy-Littlewood’s operator, Urysohn’s lemma, Ergorov’s, Lusin’s, Tietze’s theorems, and the Lebesgue differentiation theorem. We also improve the MATHCOMP-ANALYSIS support

for real functions (extended lim sup / lim inf theory) and for topology (lower semicontinuity, normal spaces, etc.).

Lebesgue differentiation theorem and the FTC

Let us note $[f]_A \stackrel{\text{def}}{=} \frac{1}{\mu(A)} \int_{y \in A} |f(y)| (\mathbf{d}\mu)$ the average of a real-valued function f over the set A . We introduce the notation $\overline{f_{B(x,r)}} \stackrel{\text{def}}{=} [\lambda y. f(y) - f(x)]_{B(x,r)}$ where $B(x,r)$ is a ball centered at x of radius r . Given a real-valued function f , a *Lebesgue point* is a real number x s.t. $\overline{f_{B(x,r)}} \xrightarrow[r \rightarrow 0^+]{}$ 0. Reusing the Lebesgue measure (hereafter μ) formalized in [AC23, §5.2] and notations from MATHCOMP-ANALYSIS, we formally define Lebesgue points:

```

Definition iavg (f : R -> R) (A : set R) :=
  (fine (mu A))^-1%:E * \int[mu]_(y in A) ^| (f y)%:E |.
Definition favg (f : R -> R) (x r : R) := iavg (center (f x) \o f) (ball x r).
Definition lebesgue_pt (f : R -> R) (x : R) := favg f x r @[r --> 0^'+] --> 0.

```

To inject/project between numeric values and their extended versions we use $\%:E/\text{fine}$. The notation for convergence $f x @ [x \rightarrow a] \rightarrow l$ stands for $f(x) \xrightarrow[x \rightarrow a]{}$ l . We use the notation \wedge^+ for the filters of neighborhoods of x intersected with $]x, +\infty[$ to define the fact that r tends to 0^+ . The Lebesgue differentiation theorem states that, for a real-valued, locally-integrable function f (i.e., integrable on compact subsets of its domain), we have Lebesgue points a.e.:

```

Lemma lebesgue_differentiation (f : R -> R) : locally_integrable setT f ->
  {ae mu, forall x, lebesgue_pt f x}.

```

The first step of the proof of the Lebesgue differentiation theorem is to reduce the problem to functions $f_k \stackrel{\text{def}}{=} f \mathbb{1}_{B_k}$ with $B_k \stackrel{\text{def}}{=} B(0, 2(k+1))$:

```

Lemma lebesgue_differentiation_bounded (f : R -> R) :
  let B k := ball 0 k.+1.*2%:R in let f_k := f * \1_(B k) in
  (forall k, mu.-integrable setT (EFin \o f_k)) ->
  forall k, {ae mu, forall x, B k x -> lebesgue_pt (f_k) x}.

```

This problem reduction is often hand-waved in lecture notes (one exception is [Sch97, (5.12.101)]). Second, instead of proving for all k that we have a.e. over B_k Lebesgue points, it is sufficient to prove that the set $A_k(a) \stackrel{\text{def}}{=} B_k \cap \{x \mid a < \limsup_{r \rightarrow 0} \overline{f_{k B(x,r)}}\}$ is negligible for all $a > 0$. The idea is to exhibit a sequence of continuous functions g_i such that $A_k(a) \subseteq \underbrace{\bigcap_n B_k \cap \{x \mid f_k(x) - g_n(x) \geq a/2\}}_{(a)} \cup \underbrace{\{x \mid \text{HL}(f_k(x) - g_n(x)) > a/2\}}_{(b)}$ where

$\text{HL}(f)(x) \stackrel{\text{def}}{=} \sup_{r>0} \{[f]_{B(x,r)}\}$ is the Hardy-Littlewood operator. We can show that the measure of the right-hand side is null. To deal with (a), we use Markov's inequality and the fact that continuous functions are dense in L^1 . To deal with (b), we need the Hardy-Littlewood maximal inequality, which in turns relies on Vitali's covering lemma. Figure 1 shows the main lemmas that we add to MATHCOMP-ANALYSIS to formalize this proof.

As a consequence, we can prove the first FTC for Lebesgue integration:

```

Theorem FTC1 (f : R -> R) : mu.-integrable setT (EFin \o f) ->
  let F x := (\int[mu]_(t in ^)-oo, x) (f t)%R in
  forall x, lebesgue_pt f x ->
  derivable (F : R^o -> R^o) x 1 /\ (F : R -> R^o)^(^)-() x = f x.

```

The predicate `derivable` is for derivability (`1` is the direction) and the notation $\wedge^{\circ}()$ is for derivatives with domain \mathbb{R} [ACR18, §4.5]. The theorem above connects these notions with the Lebesgue integral [AC23, §6.4]. The proof goes through a generalization of the Lebesgue differentiation theorem where balls are replaced with *nicely shrinking* sets [Li22, II.4.1].

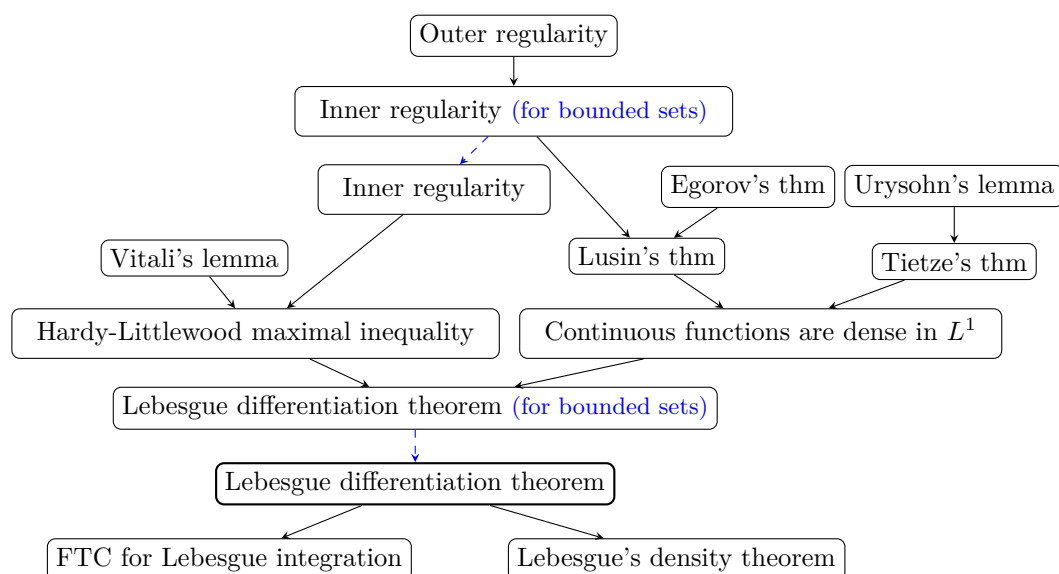


Figure 1. Development overview [AS23]

Related work

We have been using various documents to formalize the Lebesgue differentiation theorem. In particular, the main lines are drawn from [Bow14]. For the proofs of the Hardy-Littlewood maximal inequality and the proof of the Lebesgue differentiation theorem we used books by Li [Li22] and Schwartz [Sch97]. Surely, the same contents can be found elsewhere.

The FTC has already been the target of formalizations in proof assistants. It can be found in COQ but for the Riemann integral in a constructive setting [Cru02, §6]. NASalib does not feature the first FTC for Lebesgue integration but an elementary proof (for a C^1 function) of the second FTC [NAS23a, file `lebesgue_fundamental.pvs`]. Isabelle/HOL features the first FTC for Lebesgue integration but for continuous functions whereas we prove it for integrable functions [AHS17, §3.7]. Mathlib has a number of variants of the first FTC that require integrability and continuity at the endpoints but that establish strict differentiability [Mat23c]. They stem from a lemma analogous to a strengthening of the Lebesgue differentiation theorem with nicely shrinking sets [Mat23d]. Mathlib also contains a formalization of the Lebesgue differentiation theorem [Mat23b] which is used to prove a generic version of Lebesgue's density theorem [Mat23a][Nas23b, §3.2]. In other words, we are able to match our lemmas with Mathlib lemmas but statements are slightly different and proofs are organized in a different way.

Perspectives

MATHCOMP-ANALYSIS is still under development in the sense that not all notions are formalized as we would like them to be. Yet, the proof of the Lebesgue differentiation theorem shows that it is a rich library and already a useful tool to formalize mathematics. Indeed, we could use it to revisit Urysohn's lemma by producing an original proof. We are now working on the second FTC for Lebesgue integration whose most general form deals with *absolutely continuous functions* [Moh21], using as the main ingredient the Radon-Nikodým theorem already available in MATHCOMP-ANALYSIS [IA23].

Acknowledgments The authors are grateful to A. Bruni, C. Cohen, Y. Ishiguro, and T. Saikawa for their inputs. This work has benefited from feedback gained during the MATHCOMP-ANALYSIS development meetings. The first author acknowledges support of the JSPS KAKENHI Grant Number 22H00520.

References

- [ABC⁺23] Reynald Affeldt, Yves Bertot, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, Pierre Roux, Kazuhiko Sakaguchi, Zachary Stone, Pierre-Yves Strub, and Laurent Théry. MathComp-Analysis: Mathematical Components compliant analysis library. <https://github.com/math-comp/analysis>, 2023. Since 2017. Version 0.6.6.
- [AC23] Reynald Affeldt and Cyril Cohen. Measure construction by extension in dependent type theory with application to integration. *J. Autom. Reason.*, 67(3):28:1–28:27, 2023.
- [ACR18] Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization techniques for asymptotic reasoning in classical analysis. *J. Formaliz. Reason.*, 11(1):43–76, 2018.
- [ACS23] Reynald Affeldt, Cyril Cohen, and Ayumu Saito. Semantics of probabilistic programs using s-finite kernels in Coq. In *12th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP 2023) Boston, MA, USA, January 16–17, 2023*, pages 3–16. ACM, 2023.
- [AHS17] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *J. Autom. Reason.*, 59(4):389–423, 2017.
- [AS23] Reynald Affeldt and Zachary Stone. Lebesgue differentiation theorem and applications. Available at <https://github.com/math-comp/analysis/pull/1065>, 2023. Pull request to [ABC⁺23].
- [Bow14] Lewis Bowen. Lecture notes in real analysis. Available at <https://web.ma.utexas.edu/users/lpbowen/m381c/lecture-notes.pdf>, Dec 2014. University of Texas at Austin.
- [Cru02] Luís Cruz-Filipe. A constructive formalization of the fundamental theorem of calculus. In *Selected Papers of the 2nd International Workshop on Types for Proofs and Programs (TYPES 2002), Berg en Dal, The Netherlands, April 24–28, 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 108–126. Springer, 2002.
- [IA23] Yoshihiro Ishiguro and Reynald Affeldt. A progress report on formalization of measure theory with MathComp-Analysis. In *25th Workshop on Programming and Programming Languages (PPL2023), Nagoya University, March 6–8, 2023*. Japan Society for Software Science and Technology, Mar 2023. 15 pages.
- [Li22] Daniel Li. *Notions fondamentales d’analyse réelle et complexe*. Ellipses, 2022.
- [Mat23a] Mathlib 4. File `MeasureTheory/Covering/DensityTheorem.lean`. [url](#), Dec 2023.
- [Mat23b] Mathlib 4. File `MeasureTheory/Covering/Differentiation.lean`. [url](#), Dec 2023.
- [Mat23c] Mathlib 4. File `MeasureTheory/Integral/FundThmCalculus.lean`. [url](#), Dec 2023.
- [Mat23d] Mathlib 4. File `MeasureTheory/Integral/SetIntegral.lean`. [url](#), Dec 2023.
- [Moh21] Maran Mohanarangan. The fundamental theorem of calculus for Lebesgue integration. Technical report, ETH Zürich, 2021. Exercise Class 13, Measure and Integration, Spring 2021, available at https://metaphor.ethz.ch/x/2021/fs/401-2284-00L/sc/notes_exclass13.pdf.
- [MT21] Assia Mahboubi and Enrico Tassi. *Mathematical Components*. Zenodo, Jan 2021.

- [NAS23a] NASALib. NASA PVS library of formal developments. Current version: 7.1.1. Available at <https://github.com/nasa/pvslib>, 2023.
- [Nas23b] Oliver Nash. A Formalisation of Gallagher’s Ergodic Theorem. In *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:16, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.
- [SA23] Ayumu Saito and Reynald Affeldt. Experimenting with an intrinsically-typed probabilistic programming language in Coq. In *21st Asian Symposium on Programming Languages and Systems (APLAS 2023), November 26–29, 2023, Taipei, Taiwan*, volume 14405, pages 182–202. Springer, 2023.
- [Sch97] Laurent Schwartz. *Analyse III: Calcul intégral*. Hermann, 1997.
- [ZBS⁺23] Li Zhou, Gilles Barthe, Pierre-Yves Strub, Junyi Liu, and Mingsheng Ying. CoqQ: Foundational verification of quantum programs. *Proc. ACM Program. Lang.*, 7(POPL):833–865, 2023.

Packaging Proofs with Why3find

Loïc Correnson

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

With the increasing maturity of proof assistants, diving into the development of large theories is appealing, but existing toolchains might not scale. Although standard software engineering methods can be applied to mechanized proof development, specific issues shall be addressed. In this article, we focus on the Why3 platform. We present why3find, an independent tool for supporting the development of large, trustworthy Why3 packages. Why3find is designed to address common issues encountered in real world industrial developments based on formal methods. It proposes Why3-based solutions for configuring projects, managing dependencies, proving and checking proofs, tracking axioms and possible inconsistencies, extracting code, generating documentation and distributing packages.

1 Introduction

Software engineering practices have been profoundly modified by the development of package repositories and package managers. Developers not only need to learn programming languages, they must learn their associated package ecosystems as well. Typically, the OCaml language is now supported by the so-called OCaml Platform¹: the OCaml package repository, OPAM, and the build system, dune smoothly operate with each other. Similarly, the Rust² language comes with its Crates repository of packages and the Cargo package manager and build system; this observation generalizes to all main stream programming languages, such as Java, JavaScript, Python, *etc.*

Packages are shared by developers, so they can benefit from other's previous work and build their own new software upon. Quality of external packages can be an issue, and people are generally looking at some *metrics*, *e.g.* popularity, dependencies, activity, quality of documentation, *etc.*, in order to choose the packages they will rely on.

What about developing with proof assistants? Following the principles of the Curry-Howard correspondence where propositions are types, and proofs are programs [Wad15], crafting theories shall be regarded as software engineering. Many mature programming languages are available for *programming* proofs. Let us simply mention Coq, Isabelle/HOL or Lean. Noticeably, Coq and Isabelle also come with their associated package repositories^{3,4}.

Mechanized proofs are the Graal of Formal Methods practitioners: not only do you have a nice piece a software, not only do you have a proof of its correctness, but this proof has been completely machine checked. However, some questions come up: are there issues when using third-party proof packages? How far shall we trust the development of others? What is actually checked by the compilers? All proof assistants feature the declaration of *axioms*

¹<https://ocaml.org/docs/platform>

²<https://www.rust-lang.org>

³<https://coq.inria.fr/coq-package-index>

⁴<https://www.isa-afp.org>

and unspecified *parameters*. How many of such *hypotheses* are you importing when linking your own proofs with external packages? Does the documentation provide hints regarding such concerns?

An illustration of how important those questions can be found in the guidelines from ANSSI⁵ [ANS21, §4.1, p. 12], where security evaluators are asked to use the `Print Assumptions` command from Coq environment, and to review any missing definition by hand.

In this article, we present an open-source tool, named `why3find`, whose purpose is to provide some solutions to the aforementioned issues in the context of the Why3 platform. We have developed this tool to support our own industrial and academic developments, and we think that it can also benefit to other people in the Formal Method community. Briefly speaking, `why3find` is a package manager dedicated to Why3 development that smoothly integrates with the OCaml platform through OPAM and dune. Moreover, `why3find` is designed to be compliant with security evaluation and certification processes, by generating enhanced documentation decorated with proofs evidences and hypotheses reporting. It also eases proof replay and collaborative, incremental proofs development.

The article is structured as follows: in Section 2 we present Why3 development in general. Section 3 presents the features provided by `why3find` to ease the development of proofs. In Section 4, we introduce the notion of module consistency and how we implemented it in `why3find` to track axioms and logical inconsistencies. Then we illustrate the features proposed by `why3find` for generating documentation (Section 5) and for distributing packages (Section 6). We finally conclude and present some perspectives and future work directions.

2 Developing with Why3

Why3 [BFMP11, FP13] is a platform developed at INRIA and University of Paris-Saclay for Deductive Program Verification [Fil11]. It provides a rich language for writing logic specifications and programming, called WhyML, and relies on external theorem provers, both automated and interactive, to discharge verification conditions. The WhyML language is designed for the development of first-order logical theories. It also offers a programming language largely inspired from OCaml, featuring algebraic data-types, mutable records, exceptions, and contract-based specifications in Hoare logic style. Furthermore, Why3 can extract executable OCaml programs from WhyML code, hence providing a way to develop correct-by-construction OCaml programs.

WhyML sources are organized in files, each file defining collections of theories and modules. Each theory and module consists of logical declarations that can be abstract (assumed) or concrete (defined). The Why3 compiler uses *Weakest Precondition Calculus* to generate verification conditions (VCs) from logical declarations and program specifications. The generated VCs are then submitted to external provers. Why3 also offers a collection of *transformations* that can be applied interactively to decompose complex VCs into smaller, hopefully simpler ones. Interactive sessions can be stored on disk and replayed later from the command-line.

Proof development is modular. When some module A depends on definitions from module B, properties of module B are *assumed* when proving properties of module A. This strategy makes proof development tractable, but makes external dependencies an issue for large development: one shall manually review that everything has been proved along *all* dependencies, which can be tedious.

The Why3 platform offers several command-line and GUI tools for supporting the development of Why3ML theories and programs:

⁵French National Cybersecurity Agency

<code>why3 config</code>	for detecting and configuring provers;
<code>why3 prove</code>	for proving VCs (with a single prover);
<code>why3 ide</code>	for proving and transforming VCs interactively;
<code>why3 replay</code>	for checking and/or replaying saved interactive sessions;
<code>why3 doc</code>	for generating HTML documentation from sources;
<code>why3 extract</code>	for extracting OCaml programs from sources.

A typical development with Why3 consists in repeatedly executing the following tasks: edit Why3ML source files; replay all proof sessions with `why3 replay`; interactively fix broken proofs under `why3 ide`; generate code and documentation with `why3 extract` and `why3 doc`. This process works well out-of-the-box for small or medium size projects, and one can easily automate some steps by using makefiles and shell scripts. However, when facing large projects, say thousands line of Why3 codes, software engineering problems arise.

We now describe the features proposed by `why3find` to leverage the capabilities of `why3` tools and address some issues related to large development and also common issues regarding security evaluation.

3 Automated Proving

Discharging verification conditions (VCs) is actually performed by running SMT solvers or other automated provers from `why3 ide`, possibly after applying transformations. This process is essentially manual although Why3 provides *proof strategies*, which consists of scripts that can automate this process up to a certain extent. Why3 offers predefined proof strategies and users can write their own ones.

Our experience on large academic and industrial projects — several thousands lines of Why3 code — is that proof strategies like `auto level 2` and `auto level 3` are generally able to tackle most proofs. The few residual proof tasks can still be tackled, provided we add some proof hints such as intermediate code assertions, lemma functions or by using `(by)` and `(so)` Why3 operators.

Hence, to simplify code base management, we decided to stick to automated proof strategies only. This method has many advantages from a software engineering perspective: (1) proof hints inside code is much more stable than looking for appropriate transformations and manually compose their parameters under `why3 ide`; (2) we have a simple criterion for whether a property is *provable* or not: it shall be discharged by applying the automated strategy; (3) proofs are easy to build: open `why3 ide`, select root node, click on `auto level 3`, wait for everything to be proved; (4) in case something goes wrong, apply transformations by hand under `why3 ide` to understand why the proof failed, add the necessary proof hints to the code, and replay Step (3).

Still, this iterative process suffers from some pitfalls: (a) it still requires user interaction through a graphical user interface; (b) when this process is replayed on a different machine, the resulting `why3` proof session files differ, essentially because of different solver proof times; (c) checking the proof session files on a different machine is likely to fail because of inappropriate timeouts; (d) when debugging proofs, `why3 ide` still spends too much time on retrying proof branches that are likely to fail from previous runs; (e) listing all transformations actually used for a project is not that easy. Those issues make Why3 code maintenance and proof checking difficult in practice.

Alleviating those difficulties but still following our automating strategy was the primary objective of `why3find`. Our solution builds on several new ingredients: *proof certificates* which are simplified forms of Why3 proof sessions that serve many purposes, *prover calibration* to deal with machine-dependent proof times, and a *configurable* automated proof strategy. Additionally, `why3find` uses a local cache for proof results to speed up iterative proof development, and can manage a cloud of provers spread over multiple machines. We now describe in more details those new features.

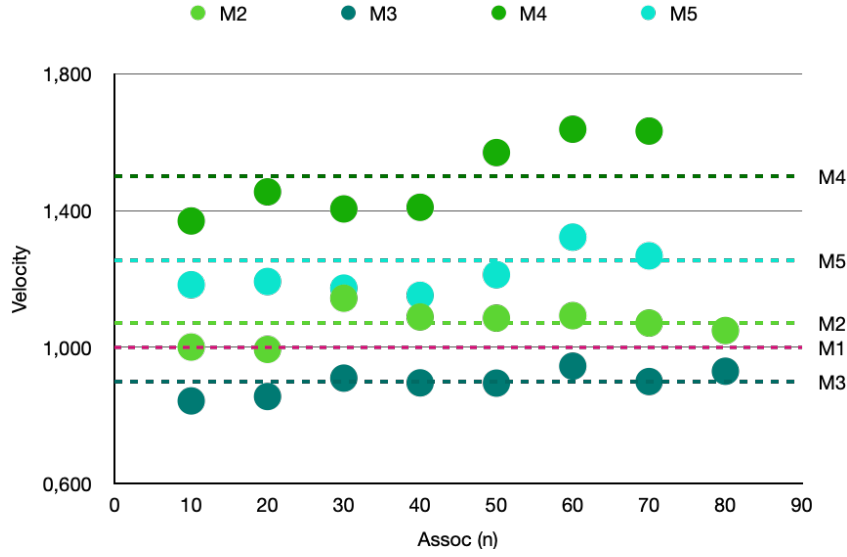


Figure 1. Experimenting the Velocity Law.

3.1 Prover Calibration

As mentioned above, replaying proofs on different machines is difficult because execution times are likely to be different from one computer to another. Of course, one could decide to multiply timeouts by a large factor, say 5 or 10, to guarantee that proofs will succeed on every machine. But when debugging proofs, this is *not* an option: waiting for all provers to reach their timeouts is far too long, and developers really want incomplete proofs to fail *fast*.

After facing too many times this issue, we decided to conduct a short study in order to measure how proof times are different from one machine to another. Our intuition is that, for given formula φ , prover π and machine M , the proof time $T(\varphi, \pi, M)$ will roughly obey a linear law that only depends on the prover π and the machine M . We introduce *velocity*, denoted by $\alpha_{M/M'}^\pi(\varphi)$, to be the ratio between proof time on machine M and machine M' , for problem φ with prover π . Our intuition is that velocity α (approximately) does not depend on φ :

$$\forall \varphi, \quad \frac{T(\varphi, \pi, M)}{T(\varphi, \pi, M')} \equiv \alpha_{M/M'}^\pi(\varphi) \simeq \alpha_{M/M'}^\pi \quad (1)$$

To test this law, we have set up a test bench with a collection of predefined, automatically generated formulas, and we have measured the associated proof times on various machines and various SMT solvers, namely Z3, CVC4 and Alt-Ergo. Choosing an appropriated family of reference proof problems was not so easy. For now, after several tries, we stick to a parameterized problem $\text{Assoc}(n)$ that is formulated as follows: let \oplus be an associative operator, we ask the solvers to prove that left and right parenthesized forms of $x_1 \oplus \dots \oplus x_n$ are equal:

$$\frac{\forall xyz. x \oplus (y \oplus z) = (x \oplus y) \oplus z}{\forall x_1 \dots x_n. x_1 \oplus (x_2 \oplus (\dots \oplus x_n)) = ((x_1 \oplus x_2) \oplus \dots) \oplus x_n} \text{Assoc}(n)$$

Our intuition seems to be roughly correct, as illustrated in Figure 1: the diagram shows measured values of velocity $\alpha_{M_i/M_1}^\pi(\varphi(n))$ on different machines $M_{i \leq 5}$ for a range of formulas $\varphi(n) = \text{Assoc}(n)$ with $10 \leq n \leq 80$, and prover $\pi = \text{CVC4}$. For each machine M_i , the measured proof times are roughly dispatched around a median line that we define to be the value of α_{M_i/M_1}^π . Similar results can be observed with other provers and other families of formulas.

The simple linear law of Equation (1) enjoys nice algebraic rules that allow us to convert proof times between different machines. For instance, we have the following rules:

$$\alpha_{M/M'}^\pi = 1/\alpha_{M'/M}^\pi \quad \alpha_{M_1/M_2}^\pi = \alpha_{M_1/M_0}^\pi \times \alpha_{M_0/M_2}^\pi \quad (2)$$

We introduce the notion of *local proof time*, which is the proof time measured on a given machine, and the notion of *normalized proof time*, which is the proof time measured on one distinguished reference machine.

Each prover π and median time t used by `why3find prove` or configured by `why3find config` will be *calibrated* by looking for a parameter n such that `Assoc(n)` is relevant for the configured median time.⁶

The first time a prover π is calibrated on a machine M , this machine is designated to be the reference for this prover in the project. The proof time $t = T(\text{Assoc}(n), \pi, M)$ is measured and the pair (n, t) is recorded and versioned in the project configuration for prover π . Then, when using the same prover π on a different machine M' , the *velocity* $\alpha_{M'/M}^\pi$ is locally measured and stored in the *local* cache of machine M' . It is then always possible to convert *local proof times* with prover π on machine M' to *normalized proof times* for reference machine M , and conversely, by using Equations (2).

3.2 Proof Certificates

Why3 proof sessions created by `why3 ide` store very precise and complete results for all provers and transformations resulting from user interactions. Despite interesting Why3 developments for reusing past sessions on modified code [BFM⁺], proof sessions are not really robust in practice, especially when replaying a proof on different computers, when debugging failed proofs and when propagating changes along module dependencies.

We introduce a simpler proof format that only records *one single* way to discharge proof objectives, if any, using *normalized proof times* only (See §3.1). These proof certificates are stored and versioned along with other source files of the project, so that they are shared between different machines. This is different from cached proof results and Why3 session files that are both *local* to one machine.

Proof certificates are simple trees with the following abstract syntax, where π identifies a prover, t a *normalized* proof time and τ identifies a Why3 transformation (without parameters):

$$c ::= \perp \mid \pi : t \mid \tau(c_1, \dots, c_n)$$

Certificate \perp designates an unproved goal, $\pi : t$ a proof goal that can be discharged by prover π in normalized time t , and $\tau(c_1 \dots c_n)$ a proof goal that can be transformed by applying transformation τ , resulting in n sub goals respectively associated with proof certificates $c_{i \in 1..n}$. A proof certificate is *complete* when it contains no \perp .

Proof certificates serve many purposes in `why3find`: when complete, they witness successful proofs that can be concisely described in documentation (See §5); they ease the replay of successful proofs on *different* computers, but they also ease proof updates and proof debugging under the standard `why3 ide` when incomplete (See §3.4).

Technically, proof certificates are stored with prover calibration and velocity data with the granularity of Why3 modules and shall be versioned with other files of the project and distributed with proof packages (See §6). On the contrary, Why3 proof sessions are local data that shall *not* be versioned for the project.

3.3 Proof Strategy

Our `why3find prove` tool implements an automated proof strategy similar in spirit to Why3 built-in proof strategies `auto level 2-3` with several improvements. In this section, we describe this proof strategy and its integration with *proof certificates* introduced above.

⁶We use a dichotomic search based on some initial hard-coded guess for a family of SMT solvers.

Consider a new proof goal for which no proof certificate is available yet. The `why3find` proof strategy consists of several steps that are tried in sequence until success:

1. all *configured* provers are first tried sequentially with a low timeout (1/5 of the *local, median* time);
2. then all provers are tried in parallel with the normal timeout (*local, median* time), and interrupted as soon as one succeeds;
3. if no prover succeed, *configured* transformations are tried in sequence; we stick to the first transformation that can be applied and sub-goals are queued so that the proof strategy can be recursively applied on them;
4. finally, if no transformation is applicable, all provers are lastly tried in parallel with twice the normal timeout, like in step 2.

The key differences with built-in Why3 proof strategies are: (a) our proof strategy is easily and globally configurable (provers, median time, strategies); (b) we introduce sub-second timeouts (using 1-2s median time on modern computers *is* definitely an option); (c) we introduce a novel fail-fast prove-fast heuristic in Step 1. Actually, in many projects, we observed that there is *one* distinguished solver which is faster than the other ones at solving *most* of the generated proof goals. In such situations, we configure this particular solver to be the first one to be tried during Step 1 of our proof strategy. This choice is quite efficient: it limits the number of provers scheduled in parallel for most goals, and finally saves a lot of CPU load and time.

In Step 3, we do *not* backtrack on proof transformations for which the generated sub-goals would be incomplete in the end. This is important in order to fail as fast as possible on *incorrect* proofs. We experiment that it is more efficient in practice to spend time on debugging proofs and introducing proof hints inside Why3 code, compared to exploring *all* the possible proof trees that can be generated by applying every transformation on every proof node.

When all proof tasks have been dequeued, proof certificates (complete or not) are generated and stored on disk.

3.4 Incremental Proofs

Consider now a proof goal for which a proof certificate has been stored from a previous run. We have no guarantee that this proof certificate is still applicable: maybe the code for this goal or any of its module dependencies have been modified. Proof certificates are associated to proof goals by names (using module path, module name and declaration name), hence previous proof certificates can still be used as an initial *guess* on how to prove our possibly updated proof goal. At least, possibly some prefix of the proof certificate will still apply to the new goal.

Compared to re-exploring the entire proof tree from scratch, using *some* proof certificate as an initial guess, initially complete or incomplete, is very efficient in practice. We observed that proof certificates are quite stable during incremental proof development.

In case of failure, the `why3find prove -i` option can be used to interactively fix the proof: it generates on-the-fly a local Why3 proof session and launches a pre-configured `why3 ide` so that the user can debug its proof. Moreover, compared with a full Why3 proof session generated by `why3 ide`, every proved sub-goals is already registered with a complete proof tree generated from the certificate, so that the user can quickly focus on the failed goals and `why3 ide` will no longer waste time to re-try proof attempts that are doomed to fail. Finally, `why3 ide` is pre-configured with custom Why3 proof strategies that mimics the different steps of `why3find` proof strategy, with respect to the current project configuration.

This process reveals to be user-friendly and very efficient in practice. Still, we optimized it a bit further. First, each proof replay indeed generates a new proof certificate with different

$$\perp \simeq \perp \qquad \frac{t/2 \leq t' \leq 2.t}{\pi : t \simeq \pi : t'} \qquad \frac{c_i \simeq c'_i}{\tau(\bar{c}) \simeq \tau(\bar{c}')}$$

Figure 2. Similar Proof Certificates.

proof times, even if we use calibration. Second, when a proof fails, we generally obtain a proof tree with only few leaves to be completed. Once fixed, probably some intermediate nodes of the tree can now be solved by provers without requiring a transformation to be applied. However, since we still use the previous proof certificate as an initial guess for the updated goal, all previous intermediate transformations will still be applied.

To avoid those two issues, we proceed as follows. First, we define a relation of *similarity* between two certificates: two certificates are *similar* if they only differ by proof time on their leaves, and all the proof times are within half-to-twice from each others (see Figure 2). After proof replay or completion for a given proof goal, the newly computed proof certificate is stored on disk only when it is *not* similar to the previously existing one for the same goal. This strategy greatly improves the stability of proof certificates when synchronizing projects between different machines.

Second, when storing an incomplete proof certificate, every transformation node with only incomplete children is removed and replaced by \perp . This avoids keeping doomed proof trees that do not really simplify the initial proof goal.

Last, but not least, we implement `why3find prove -m` option that tries to *minimize* proof certificates: when reusing a complete proof certificate as a guess, we try to replace transformation proof nodes $\tau(c_1, \dots, c_n)$ with a single prover call by applying Step (2) of our automated proof strategy instead. This operation can be costly, although it is optional: large proof certificates are still correct and can still be used to replay proofs automatically.

The `why3find prove` command provides options to further tune the proof strategy and how proof certificates are reused and/or updated.

3.5 Speeding Up Proofs

Whatever proof task is performed, for a proof goal or for calibration, all prover results are stored and cached locally by default. We use the hash of proof tasks computed by Why3 to efficiently associate prover results to proof goals. Proof results are automatically upgraded or downgraded with respect to timeouts (*e.g.* a success with proof time 2.5s becomes a timeout for asked time 1s).

Using the cache during development is necessary in practice. It saves a significant amount of time during the process of incrementally developing and fixing proofs. Moreover, it also makes proof minimization not that costly, since proof attempts that are known to fail cost nothing but a disk access. However, the `why3find prove` command provides dedicated options to further tune or disable cache management.

A key principle of `why3find` proof strategy is to *fail fast* on incorrect proofs. For this purpose, when new proof tasks are dequeued, we give priority to tasks with *incomplete* certificates, and `why3find prove` can be asked to stop on the first incomplete proof. This strategy, combined with cache management and launching pre-configured `why3 ide` on-demand, makes proof development quite user-friendly and efficient.

3.6 Proof Server

Both Why3 and `why3find` schedule proof tasks in parallel to benefit from all available cores of the local machine. However, many proof tasks must wait for an available slot from `why3` proof scheduler. Moreover, when several developers want to collaborate on a given

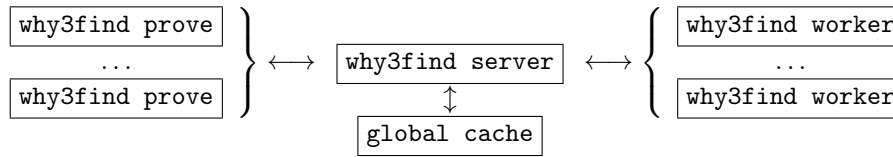


Figure 3. Proofs in the Cloud.

project, they all need to install the same provers, with the same versions, on their respective machines.

To alleviate those issues, **why3find** can establish a proof server and run a cloud of workers that can dynamically offer their cores and their solvers to proof developers. The proposed architecture is illustrated in Figure 3.

Servers and workers are implemented with the Zmq library⁷ which offers built-in support for robust server-worker architecture with automatic load-balancing facilities. The **why3find server** command establishes a server with a centralized on-disk proof cache, and waits for developers and workers to connect.

Proof tasks emitted by developers when running **why3find prove** are automatically submitted to the Server, which will immediately reply when the proof task is already in the centralized proof cache. This first query is fast since only the hash of the proof task is exchanged with the server. In case the proof task is unknown, the Server asks for the proof task input file and stores it in its global cache.

On the other side, workers can be created with command **why3find worker** to offer their cores and provers to the Server. Pending proof tasks received by the Server are submitted to available workers. Proof results are sent back to the Server, which stores them in the cache and sends them back to the emitters, provided they are still connected and waiting for them.

Developer processes can also play the role of workers. Actually, when **why3find prove** emits a proof task, it is sent to the Server *and* locally scheduled on its own cores. Depending on the respective machine loads, any of the working processes may reply first. The mission of the Server is to orchestrate and synchronize all these tasks.

The Server's global cache is more sophisticated than the **why3find prove** local caches. Where local caches on each developer's computer can be easily erased to save space, we want a more clever cache management on Server side. We currently use a multi-layered cache. New entries are always inserted in top-layer 0. When some entry is not found in layer n it is looked up in layer $n + 1$ until it is found. Once an entry is retrieved from layer $n > 0$, it is immediately promoted to layer 0. Periodically, Server maintainers might ask **why3find server** to prune the global cache up to layer N : each layer $k < N$ is renamed into layer $k + 1$ and layers $k \geq N$ are simply erased. This simple heuristic makes live queries to remain in layer 0, and layers $n > 0$ to store all queries that were not claimed since the last n maintenance periods. The **why3find server** command can be used to obtain statistics on cache layers in order to ease cache maintenance.

4 Checking Proofs Consistency

The Why3 platform implements and promotes a modular proof strategy: when asked to prove a property from module A , all other results from module B that A depends on are assumed as hypotheses. This is a mandatory feature to have in order to scale and to develop large theories, otherwise proof development would be exponential in the number of proved properties. However, this modular proof strategy implies that proofs for module A are

⁷<https://zeromq.org>

complete if, and only if, all proofs from module B that A depends on are also complete, and so on. Off the shelf, Why3 provides no tool to check such a completion.

Moreover, the Why3 language allows the introduction of declarations with *no* associated definition: one can introduce abstract types, undefined logic functions and predicates, *axioms*, and undefined functions with *assumed* post-conditions and effects. It is even possible to *admit* a code annotation within a function definition.

To fully trust a Why3 development, we should be able to answer the following questions:

- (a) Given a module A , are all of its properties proved?
- (b) If module A depends on module B , are all properties of B also proved?
- (c) Does module A have assumed hypotheses? Is it possible to fulfill or justify them in some way?
- (d) What about B hypotheses when A depends on B ?

There are some problematic code patterns that an evaluator shall be aware of. For instance, the following abstract declaration introduces an inconsistency whenever function f is invoked, since it is admitted that its returned value would be strictly both positive and negative:

```
val f () : int
  ensures { result > 0 }
  ensures { result < 0 }
```

To avoid such inconsistencies in admitted properties, our idea is to ask developers to witness *some* possible definition for abstract declarations and to prove them to be valid. Hence, one can be definitely sure that abstract definitions are free of inconsistencies.

Fortunately, Why3 provides a feature for refining modules, called module cloning, that we will investigate in more details soon.

But even *defined* and *proved* declarations might drive us into dire straits. Consider for instance the following definition for abstract function f above:

```
let f () : int
  ensures { result > 0 }
  ensures { result < 0 }
  = assume { false } ; any int
```

This function definition is trivially proved by Why3, although the logical inconsistency is still there. The problem comes from the assumed clause *inside* the definition of the value, which is not required by Why3 to be proved, contrarily to regular `assert` clauses. Unfortunately, as far as we know, Why3 does not offer any way to refine further such an assumed clause. Hence, we shall pay special attention to them.

As a matter of fact, providing evidences of *proofs completeness* — Questions (a) and (b) — and *proofs consistency* — Questions (c) and (d) — is not so easy when dealing with large code bases. Moreover, for developers to rely on properties provided by third party libraries, they will expect convincing and concise answers to such questions.

Since the main purpose of `why3find` is to provide packaging and evaluation-oriented features for Why3 developments, we shall address those issues. Our proposal is to systematically look for proof completeness and track possible sources of inconsistencies and to report on them, typically when generating documentation.

The remaining of the section briefly introduce Why3 module cloning and how it can be used to provide witness of absence of logical inconsistencies.

4.1 Why3 Module Cloning

From the definition of the WhyML language, a Why3 module consists of a collection of types, logic constants, functions and predicates, axioms and lemmas, and functions with contracts.

Declarations can be abstract, which means that the module is generic and that it can be refined by assigning concrete definitions to abstract declarations. Refinement is introduced by using module cloning declarations, with the following syntax:

```

module M
  ...
  clone A with  $x_1 = a_1, \dots, x_n = a_n$ 
  ...
end

```

Such a declaration inserts all the declarations from module A into module M after substituting abstract declarations $A.x_i$ with a_i . Each a_i can be either abstract or concrete depending on the context, and unless specified, axioms from A are turned into lemmas when inserted in M . On the contrary, lemmas from module A are treated as axioms from M 's point of view, since Why3 already issued appropriate proof obligations for them when proving module A , like module dependencies. When refining abstract functions with contracts, the new target function contract is proved to be consistent with the abstract one.

4.2 Proof Consistency

The problem with abstract module declarations is that they can introduce inconsistencies, as illustrated in the introduction. Without diving here into the complete semantics of the WhyML language [Fil13], logical inconsistencies might occur when there is no way to refine an abstract declaration with a concrete definition such that associated hypotheses (axioms and contracts) can be fulfilled.

Hence, if module A has been cloned inside some module M in such a way that all abstract definitions from A are assigned to concrete definitions, then proving M entails that A is free from logical inconsistencies.

Of course, if module M or module A have dependencies on other modules, they can also be sources of inconsistencies. Partially cloning a module also introduces abstract declarations that shall be taken into accounts.

To avoid this problem, we introduce the notion of module consistency. Module A is defined to be consistent if and only if there exists a module M that clones A such that after taking cloning into account:

- (a) Module M has no residual axiom and no residual abstract function contract.
- (b) The definitions of M — and A — are free of assumed clauses.
- (c) All dependencies of M — and A — are consistent modules.

Notice that we do not require abstract types, constants, pure logical functions and predicates to be assigned a concrete definition. Indeed, since every type is *inhabited* by construction in Why3, provided module M has no more axioms, any residual abstract declarations may be assigned any concrete value of the expected type without introducing any logical inconsistency.

4.3 Checking Module Consistency

Proof consistency is checked by `why3find` when proving a module and when generating package documentation. For every module A with abstract declarations, it checks for local

consistency inside the modules of the package only. This allows for reporting consistency at the package level, without taking into account future module instances from package clients.

Moreover, since modules of Why3 standard library are fully realized in Coq, they can be considered to be fully consistent as well, even if their Why3 declarations contain bundles of axioms and abstract values.

Local proof completeness and consistency is reported by the `why3find doc` tool inside the generated documentation (See §5), and by `why3find prove` on demand.

For proof developers, witnessing evidence of soundness for a module with abstract definitions is relatively easy: one “just” needs to create, somewhere in the package, a clone instance with enough concrete definitions to be fully proved. In practice, we always succeed in providing a concrete witness of consistency for every module of our largest projects. Moreover, during this process, we have sometimes observed that some hypotheses were missing from our initial design in order to make the abstract module realistically implementable.

Computing and consolidating statistics from proof certificates and local module consistency actually provides a concise and relevant feedback regarding the trust base of Why3 developments. Moreover, it can be smoothly integrated to the generated documentation, see Section 5 for more details.

5 Providing Documentation

Generating a good documentation for formal developments is not an easy task. As an illustration, for documenting the Coq development of CompCert [Ler09], X. Leroy developed a dedicated tool, `coq2html`⁸ to overcome the limitations of Coq built-in documentation generator.

The built-in documentation generator distributed with Why3 also has its own limitations. In particular, it does not offer simple typographic facilities, and it provides no feedback on proved goals and hypotheses. Moreover, cross-referencing documentation generated from different Why3 developments is not possible (but for the standard library) and this is an issue when distributing package documentation.

Hence, we provide an enhanced documentation generator, `why3find doc` that addresses those issues. The main additional features of our documentation generator are: (a) limited support for Markdown styling; (b) foldable code parts, paragraphs or entire sections (*e.g.* to hide proofs); (c) proof certificates and feedback regarding completeness and consistency; (d) inlined clone definitions (useful for nested and exported clones); (e) inter-package cross-references; (f) additional documentation-only chapters.

Regarding proof completeness, the documentation generated by `why3find doc` outputs, for each Why3 declaration, the associated proof certificate and its completeness. A summary is also consolidated by module, by source file and by package: each identifier in the generated HTML documentation is accompanied by small colored icons and detailed tooltips to summarize whether the proof certificates are complete or not.

Similarly, local proof consistency is reported in the generated documentation with small colored icons, and tooltips with detailed statistics on proof consistency. Additional information is provided for consistency, namely the list of sound instances found (when consistent) or the list of incomplete instances (when not fully consistent).

Such facilities ease the work of evaluators when they examine the trust base of a development with Why3. Since all packages and modules have concise summaries of their dependencies, abstract parameters and axioms, proof completeness and proof consistency, an evaluator can easily navigate to the residual missing parts and forge its opinion on robustness with respect to the documentation provided by the proof developers.

⁸<https://github.com/xavierleroy/coq2html>

6 Packaging & Distributing

The development of a (large) project with Why3 requires managing some configuration data, including source files, dependencies, proof strategies, *etc.* All those data are generally materialized in shell scripts or in Makefiles together with additional data and commands to package the development and make it available for distribution.

With `why3find`, we propose a collection of command line tools to manage the project configuration data and ease its packaging and installation process.

More precisely, `why3find config` will manage the following data for a given project: (a) the other packages the project depends on; (b) the provers and transformations to be used (See 3.3); (c) the *median time* used to define normalized timeouts (See 3.1); (d) the number of parallel proof jobs; (e) the proof server to connect, if any (See §3.6). The `why3find config` provides many options to print, update or reset the global project configuration. Configuration data is stored in a configuration file that shall be versioned with the source files of the project.

When a Why3 development has been properly proved and documented with `why3find`, one may want to distribute it and make it available for other projects. To distribute a Why3 package, we decide to rely on the OPAM and dune infrastructures of OCaml Platform. Actually, recent versions of dune offer a nice feature, named *installation sites* for centralizing a local repository of files from different packages. Hence, `why3find` has its own dune installation site and the `why3find install` tool will automatically generate a dedicated dune file to install all the required artifacts for a Why3 package: configuration, sources files, proof certificates, prover calibration *and* documentation.

Moreover, `why3find` also provides features to ease the extraction of OCaml code from Why3 sources. In particular, `why3find extract` produces a dune file for compiling the extracted code, so that it can also be linked or installed with other OCaml source files in the project.

All tools provided by `why3find` actually know how to deal with installed packages, for proving, extracting and generating documentation. Altogether, we hope that those integrated features would bring the packaging and distribution of Why3 packages to real life.

7 Conclusion

We have presented a prototype tool, `why3find`, that provides facilities upon the standard Why3 tool box. From our experience on large academic and industrial projects, `why3find` really facilitates the work of both proof developers and proof evaluators and offers an infrastructure based on the OCaml Platform for distributing Why3 packages. We plan to release `why3find` in open source via OPAM within a very near future.

Although already functional, `why3find` can still be improved in many directions. Proof automation can still be improved, and our experimental model for prover calibration shall be further tested and explored: the level of automation currently provided by the tool opens the route for experimenting new heuristics and systematically evaluating their benefits. The documentation generator can also be improved in many ways. Finally, proof consistency would deserve to be refined and formalized in a proof assistant.

We also expect feedback from the community and possible integration with other proof platforms such as Coq, Isabelle or PVS. For now, we have only proposed our own, opinionated, proposals to overcome some common issues in large development with formal methods and Why3 in particular, although we are opened to external contributions. Our intention is really to contribute to make Why3 a full-featured platform for the collaborative development of mechanized proofs and proved libraries.

Acknowledgments. I would like to warmly thank the JFLA reviewers for their kind suggestions and insights. I would also like to thank Benjamin Jorge for its fruitful contributions regarding the proof strategy of Why3find. Finally, I would like to thank all the developers and security evaluators I've met so far for their instructive feedback regarding the usage and the deployment of formal method tools in their respective settings.

References

- [ANS21] Requirements on the use of coq in the context of common criteria evaluations. Rapport technique v1.1, ANSSI & INRIA, décembre 2021.
- [BFM⁺] François BOBOT, Jean-Christophe FILLIÂTRE, Claude MARCHÉ, Guillaume MELQUIOND et Andrei PASKEVICH : Preserving user proofs across specification changes. pages 191–201.
- [BFMP11] François BOBOT, Jean-Christophe FILLIÂTRE, Claude MARCHÉ et Andrei PASKEVICH : Why3: Shepherd your herd of provers. *In Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. <https://hal.inria.fr/hal-00790310>.
- [Fil11] Jean-Christophe FILLIÂTRE : Deductive Software Verification. *International Journal on Software Tools for Technology Transfer*, 13(5):397–403, 2011.
- [Fil13] Jean-Christophe FILLIÂTRE : One logic to use them all. *In Maria Paola BONACINA, éditeur : Automated Deduction – CADE-24*, pages 1–20, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [FP13] Jean-Christophe FILLIÂTRE et Andrei PASKEVICH : Why3 — where programs meet provers. *In Matthias FELLEISEN et Philippa GARDNER, éditeurs : Proceedings of the 22nd European Symposium on Programming*, volume 7792 de *Lecture Notes in Computer Science*, pages 125–128. Springer, mars 2013.
- [Ler09] Xavier LEROY : Formal Verification of a Realistic Compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [Wad15] Philip WADLER : Propositions as types. *Commun. ACM*, 58(12):75–84, nov 2015.

A diagram editor to mechanise categorical proofs

Démonstration de logiciel

Ambroise Lafont¹

¹École Polytechnique, Palaiseau, France

Diagrammatic proofs are ubiquitous in certain areas of mathematics, especially in category theory. Mechanising such proofs is a tedious task because proof assistants (such as Coq) are text based. We present a prototypical diagram editor to make this process easier, building upon the vscode extension coq-lsp for the Coq proof assistant and a web application available on the author's personal website. It currently targets the UniMath mathematical library for the Coq proof assistant, but could in principle easily be adapted to other targets.

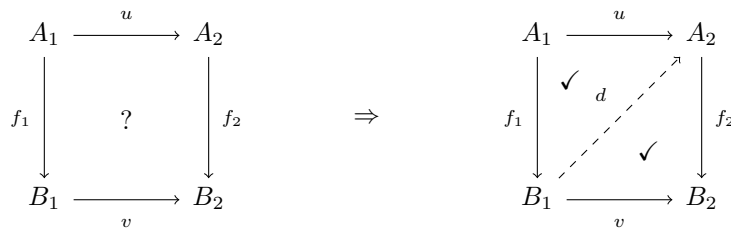
1 Introduction

Showing that two morphisms between two objects are equal is a ubiquitous task in category theory. Those morphisms are typically themselves compositions of other morphisms.

Example 1. Assuming $d \circ f_1 = u$ and $f_2 \circ d = v$, we deduce that $f_2 \circ u = v \circ f_1$, by rewriting u and v using the two assumed equalities.

Proofs by rewriting, as above, does not pose specific challenge when it comes to mechanisation in proof assistants. However, pen and paper proofs typically adopt a different *diagrammatic* approach: the two composition chains of morphisms that we want to prove equal are first drawn as chains of arrows sharing the same start and end points, thus delineating a shape with two *branches* (the two chains of arrows). Thinking of equality as a "filling", a diagrammatic proof consists in decomposing this shape into smaller juxtaposed *inner* shapes whose fillings are justified.

Example 2. A diagrammatic proof of Example 1 consists in the below right diagram, obtained by splitting the square below left into two juxtaposed "filled" triangles.



Diagrammatic proofs include enough information to construct a mechanisable proof by rewriting. This time-consuming translation is ubiquitous when mechanising categorical

results. We present a prototypical diagram editor that can automatically perform this task: from the right diagram of Example 2, the editor generates a Coq proof script that shows the desired equality using the assumed equalities.

Plan of the paper We quickly describe the technology behind the software in Section 2, before presenting the main features in Section 3. In Section 4, we finally mention some related work.

2 Technology

The software consists of three main components: a diagram editor (about 7000 lines of code), and a vscode extension (about 300 lines of code), and a small Coq library (about 100 lines of code).

2.1 The diagram editor

The diagram editor is mainly implemented in Elm, a functional programming language that compiles to JavaScript. \LaTeX labels are rendered using the KaTeX JavaScript library. It is available as a web application that runs in the browser [Laf], as a standalone desktop application (with some additional features) that embeds the web application in an electron runtime.

The diagram editor generates proof scripts relying on the mechanised UniMath mathematical library [VAG⁺].

2.2 The vscode extension

The vscode extension builds upon coq-lsp [CJGAI], which provides the vscode editor with support for the Coq proof assistant. Our extension interacts with the standalone version of the diagram editor in mainly two ways: to render the Coq goal at the cursor as a diagram (if the Coq goal is indeed an equality between morphisms), and to insert the Coq proof generated from the diagram at the cursor location.

2.3 The Coq library

The Coq library introduces some notations and tactics to convert a Coq goal context into the input format of the editor, where the objects are explicitly mentioned. For example, the top right branch of the left diagram in Example 2 is denoted by $u \cdot f_2$ in UniMath. The vscode extension would call our pretty-printing tactic `norm_graph` to convert it into the string $A_1 \dashv\vdash u \rightarrow A_2 \dashv\vdash f_2 \rightarrow B_2$, which is then sent to the editor for display.

3 Features

In this section, we present the editing capabilities of our software in Section 3.1, then the features related to mechanisation of diagrammatic proof in 3.2, and finally, in Section 3.3, we explain how our diagram editor can be used when writing a \LaTeX document with diagrams.

3.1 Diagram editor

Beyond basic editing features, our software offers (among other things) tab management, an optional grid, automatic guessing of labels when completing a (naturality) square, find & replace, z-indices (to handle edge overlaps), quicksaving. A diagram can be exported¹ to \LaTeX , json, or svg. A diagram saved in the json format can be reloaded.

¹We thank Tom Hirschowitz for implementing the \LaTeX export feature.

3.2 Mechanisation

Mechanising a diagrammatic proof with our software involves two steps that we detail in this section: construction of the diagrammatic proof, and the generation of the mechanised proof².

Construction of a diagrammatic proof Exploiting the basic editing features of the software, the user splits a shape by creating a mediating chain of arrows, as in Example 2, and selects one of the newly created inner shapes. The mediating arrows can be unnamed, leaving their definitions to be inferred later. Then, the editor generates a Coq script that states the equality corresponding to the shape. The unnamed arrows become "holes" that will be guessed later by Coq using unification, for example when applying some known equality between known morphisms. Once the proof is complete, the resulting statement is loaded back into the editor to replace the unnamed arrow with its inferred definition, and the proof script is also saved as a distinguished node sitting inside the shape.

Generation of a mechanised proof The algorithm that generates a mechanised proof assumes that the diagram is planar³. The resulting proof script consists of a list of tactics that states the intermediary lemmas corresponding to the inner shapes of the diagram and assemble them to justify equality for the outer shape, introducing associativity steps when required. Each intermediary lemma is provided with a formal proof that was given in the original diagram, as a distinguished node sitting somewhere inside the shape corresponding to the lemma.

3.3 Integration with L^AT_EX

The standalone version of the diagram editor provides some support to help editing L^AT_EX files that include diagrams: it periodically scans the L^AT_EX file to detect embedded diagrams (either inlined as json data, or saved in an external file) in a L^AT_EX comment. If that comment is not followed by the corresponding generated L^AT_EX code, then the editor loads the diagram. When the user saves it, the diagram is stored back into the file, as well as the generated L^AT_EX code corresponding to the diagram. If the user later wants to edit it again, they can simply delete the generated L^AT_EX code, and the editor will load it again as explained above. The same editing process is implemented for LyX, a WYSIWYG L^AT_EX editor for L^AT_EX.

4 Related work

Chabassier's graphical interface [CB] This software consists of a Coq plugin that interacts with a graphical interface implemented in Rust. The latter can render Coq goals as diagrams, with limited editing features. It provides a basic tactic language to make progress on the proof. It can also suggest a list of relevant lemmas that can be applied to the goal, by querying the Coq runtime.

quiver This diagram editor [Ark23] is implemented in JavaScript and runs in the browser. Compared to our software, the styling possibilities are richer, but it misses some helpful features⁴ that our software supports, such as find & replace, copy & paste, or selection extension to connected components. Contrary to our editor, the grid is not optional: vertices cannot be created out of it. Finally, it does not offer any feature to help mechanisation.

²See <https://github.com/amblafont/vscode-yade-example> for an example.

³This constraint induces a canonical choice of the primitive "inner" shapes.

⁴The following examples are mentioned as feature requests on the github repository.

References

- [Ark23] Nathanael ARKOR : quiver. <https://q.uiver.app/>, novembre 2023.
- [CB] Luc CHABASSIER et Bruno BARRAS : A graphical interface for diagrammatic proofs in proof assistants. Contributed talks in the 29th International Conference on Types for Proofs and Programs (TYPES 2023).
- [CJGAI] Ali CAGLAYAN, Emilio J. GALLEGRO ARIAS et Shachar ITZHAKY : Coq LSP. <https://github.com/ejgallego/coq-lsp>.
- [Laf] Ambroise LAFONT : A commutative diagram editor. <https://amblafont.github.io/graph-editor/index.html>.
- [VAG⁺] Vladimir VOEVODSKY, Benedikt AHRENS, Daniel GRAYSON *et al.* : Unimath — a computer-checked library of univalent mathematics. available at <http://unimath.org>.

ANNEXES

Auteurs

Affeldt, Reynald	300
Allain, Clément	116
Andrieux, Martin	205
Bagrel, Thomas	268
Bardin, Sébastien	177
Baudart, Guillaume	10
Bertholon, Guillaume	166
Blondeau-Pâtissier, Lison	190
Boulmé, Sylvain	171
Bury, Guillaume	136
Bühler, David	64
Caspar, Jean	288
Chambart, Pierre	136
Charguéraud, Arthur	166
Closse, Etienne	260
Congard, Sidney	244
Correnson, Arthur	105
Correnson, Loïc	305
Courant, Nathanaëlle	136
Cousineau, Denis	20, 85
Cristia, Maximiliano	40
Dagand, Pierre-Évariste	51
Duboc, Guillaume	264
Dubois, Catherine	40
Faissole, Florian	85
Gauche, Fabien	260
Jeannet, Bertrand	260
Koehler, Thomas	166
Lafont, Ambroise	318
Laurent, Mickaël	256
Laviron, Vincent	136
Le Gall, Tristan	85
Letouzey, Pierre	51
Marché, Claude	20
Mayero, Micaela	12
Miné, Antoine	64
Monniaux, David	171
Munch-Maccagnoni, Guillaume	288
Perrelle, Valentin	64
Pottier, François	146
Razafintsialonina, Mamy	64
Recoules, Frédéric	177
Renaud, Hadrien	240
Riba, Colin	219
Rochel, Jan	85
Rousselin, Pierre	12

Scherer, Gabriel	116
Schmitt, Alan	205
Signoles, Julien	64, 85
Sozeau, Matthieu	15
Stern, Solal	219
Stone, Zachary	300
Taghayor, Ellenor Fatemeh	51
Tasson, Christine	10
Valnet, Milla	136
Villard, Jules	16
Weil, Daniel	260

Sponsors

Le comité d'organisation des JFLA 2024 remercie chaleureusement ses généreux mécènes.



CEA List



GDR GPL



Mitsubishi Electric
R&D Centre Europe



OCamlPro



OCaml Software
Foundation



Tarides



Tweag



Université de Rennes