Journées Francophones des Langages Applicatifs 2018

Sylvie Boldo, Nicolas Magaud

JFLA 2018

Journées Francophones des Langages Applicatifs

24-27 janvier 2018, Banyuls-sur-Mer

Inria informatiques mathématiques

UNIVERSITÉ DE STRASBOURG

Préface

Les 29èmes journées francophones des langages applicatifs (JFLA) se déroulent en 2018 à l'observatoire océanographique de Banyuls-sur-Mer. Après les Pyrénées, les JFLA se déroulent cette année à la mer, plus précisément au bord de la Méditerranée, près de Perpignan. Les JFLA réunissent chaque année, dans un cadre convivial, concepteurs, développeurs et utilisateurs des langages fonctionnels, des assistants de preuve et des outils de vérification de programmes en présentant des travaux variés, allant des aspects les plus théoriques aux applications industrielles.

Cette année, nous avons sélectionné 9 articles de recherche et 8 articles courts. Les thématiques sont variées : preuve formelle, vérification de programmes, modèle mémoire, langages de programmation, mais aussi théorie de l'homotopie et *blockchain*. La sélection a été faite par les membres du comité de programme que nous remercions à partir des 24 soumissions restantes après les 31 soumissions de résumés. À noter que le format des papiers courts séduit, cela concerne presque la moitié des soumissions (11 sur 24). Nos sincères remerciements aux rapporteurs extérieurs au comité de programme : Sophie Bernard, Valentin Blot, David Braun, Lélio Brun, Pierre Chambart, Pierre-Evariste Dagand, Diego Olivier Fernandez Pons, Grégoire Henry, Julien Lopez, Mário Pereira, Raphaël Rieu-Helft, Damien Rouhling, Gabriel Scherer.

Pour compléter le programme, nous bénéficions de deux cours, l'un par Arthur Charguéraud sur la vérification interactive de programmes impératifs en utilisant CFML ; l'autre par Steven Varoumas, Benoît Vaugon et Emmmanuel Chailloux sur la programmation de microcontrôleurs. Et enfin, nous assisterons à deux exposés invités de Thomas Gazagnaire sur les micro-noyaux et MirageOS et de Stéphane Graham-Lengrand sur le typage à la rescousse de l'intégrité des données.

En plus des remerciements scientifiques, nous devons d'énormes remerciements à tous ceux et toutes celles qui nous ont accompagnés sur le chemin logistique et administratif des JFLA : Marion Oswald de la cellule congrès de l'université de Strasbourg, Patricia Fuentès de l'observatoire océanographique de Banyuls, Katia Evrat de l'Inria.

Enfin, nos sincères remerciements à nos généreux sponsors. Cette année encore, les étudiants orateurs ne paient pas les frais d'hébergement et d'inscription. Merci au CEA LIST, à ProofInUse, au GDR GPL, à OcamlPro et à TrustInSoft.

Sylvie Boldo & Nicolas Magaud
Inria Université de Strasbourg

Comité de programme

Sylvie Boldo	Inria Saclay-Île de France, LRI (présidente)
Nicolas Magaud	Université de Strasbourg (vice-président)
Clara Bertolissi	LIF-Universite Aix-Marseille
Timothy Bourke	Inria Paris, ENS
Benjamin Canou	OCamlPro
Zaynah Dargaye	CEA LIST
Alain Frisch	LexiFi
Frédéric Gava	Université de Paris-Est
Alain Giorgetti	FEMTO-ST, Université de Franche-Comté
Kim Nguyen	Université Paris-Sud
François Pottier	Inria Paris
Yann Régis-Gianas	IRIF
Laurence Rideau	Inria Sophia Antipolis - Méditerranée

Comité de pilotage

Pierre Castéran	Université de Bordeaux
Catherine Dubois	École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIIE)
Micaela Mayero	LIPN, Université Paris 13
Alan Schmitt	Inria Rennes - Bretagne Atlantique
Julien Signoles	CEA LIST
Pierre Weis	Inria Paris

Table des matières

Cours invités

Interactive Verification of Imperative Programs using CFML

Arthur Charguéraud

Inria & University of Strasbourg, CNRS, ICube

Abstract

CFML is a tool for interactive verification of higher-order, imperative programs. It leverages *Separation Logic* [5] and *Characteristic Formulae* [1] to reason in Coq about side-effects. The interactive proofs are modular and follow the structure of the code. CFML has been applied to verify classical data structures and algorithms, including binary search trees, hashtables, finger trees, catenable and splittable chunk sequences, Dijkstra's shortest path algorithm, depth first search, vectors, Eratosthenes' sieve, and Union-Find. Moreover, CFML has been recently extended to support formal complexity analyses [2].

In the original version of CFML, developed during my PhD thesis [3], an external tool was used to parse source code in OCaml syntax and to automatically generate the corresponding characteristic formulae, in the form of axioms. The characteristic formula of a term is a higher-order logic formula that describes the semantics of that term, without referring to the syntax of the term, but instead using standard logical connectives such as $\forall$, $\exists$, $\wedge$, etc... This characteristic formula can be exploited to prove that the term considered satisfies a certain specification. Such interactive proofs are conducted with the help of CFML tactics, specialized for processing characteristic formulae.

Recently, I have been working on CFML 2.0, an axiom-free reimplementation of CFML. The characteristic formula generator is now implemented inside Coq, and it is proved correct with respect to a formal semantics of the source language. The generator is able to compute inside Coq: given the abstract syntax tree of a program, it computes its characteristic formula. A similar axiom-free approach was previously demonstrated by Guéneau et al. [4], in the context of the CakeML compiler, verified in HOL4. My work is similar but goes further in terms of the integration of the translation of program values into the corresponding logical values, e.g., to allow specifying OCaml lists directly as Coq lists, without involving explicit conversion predicates.

In this course, I will present the following: description of the syntax and the big-step semantics; definition of Separation Logic combinators and triples; recursive definition of representation predicate, e.g., for describing linked lists; statement of specifications theorems; statement of Separation Logic reasoning rules; definition of the characteristic formula generator; and interactive proofs using CFML tactics. I will also briefly describe two important extensions to the logic: a first one to support read-only permissions, and a second one to support time credits, which allow for asymptotic cost analysis.

In addition, we will see how to use CFML in practice, for verifying concrete programs interactively in Coq. The course will contain hands-on sessions, during which participants will be invited to complete example proofs in Coq (v8.6).

References

[1] Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *International Conference on Functional Programming (ICFP)*, pages 418–430, 2011.

[2] Arthur Charguéraud and François Pottier. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning*, September 2017.

[3] Arthur Charguéraud. *Characteristic Formulae for Mechanized Program Verification.* PhD thesis, Université Paris Diderot, 2010.

[4] Armaël Guéneau, Magnus O. Myreen, Ramana Kumar, and Michael Norrish. *Verified Characteristic Formulae for CakeML*, pages 584–610. Springer Berlin Heidelberg, 2017.

[5] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

La programmation de microcontrôleurs dans des langages de haut niveau

Steven Varoumas[1,3], Benoît Vaugon[2], and Emmanuel Chailloux[1]

[1] Sorbonne Universités, UPMC Univ Paris 06, CNRS, LIP6 UMR 7606,
4 place Jussieu 75005 Paris, France
steven.varoumas@lip6.fr emmanuel.chailloux@lip6.fr
[2] Armadillo, 46 bis, rue de la République, 92170 Vanves, France
benoit.vaugon@gmail.com
[3] CÉDRIC, CNAM, 2 rue Conté F-75141 Paris Cedex 03, France

Introduction

Les microcontrôleurs sont des circuits imprimés programmables aux ressources matérielles (en particulier mémoire) limitées. Ces appareils, omniprésents autant dans les systèmes embarqués critiques que dans les objets du quotidien, sont traditionnellement programmés dans des langages de bas (voire très bas) niveau, comme C ou l'assembleur, dans le but de contrôler finement l'utilisation de leurs ressources. Ces langages natifs sont néanmoins mal adaptés à un modèle de déploiement d'un même programme sur plusieurs microcontrôleurs aux architectures distinctes, et nécessitent des méthodes de programmation et de débogage laborieuses. Dans ce tutoriel, nous présenterons OMicroB, une machine virtuelle OCaml pour microcontrôleurs à faibles ressources, inspirée des travaux précédents sur le projet OCaPIC [8]. Cette machine virtuelle, destinée à être exécutée sur diverses architectures matérielles (AVR, PIC, ARM, ...) permet ainsi de factoriser le développement d'applications, mais aussi de généraliser l'analyse et le débogage du bytecode associé, tout en permettant un usage précautionneux de la mémoire. Nous la mettrons à profit pour réaliser des programmes ludiques destinés à être exécutés sur des microcontrôleurs à faibles ressources, en insistant sur les particularités inhérentes à la programmation de systèmes embarqués.

Plan du tutoriel

1 **La programmation de microcontrôleurs**
En introduction, nous aborderons en détail les caractéristiques techniques de microcontrôleurs typiques, les particularités des techniques de programmation associées à ces appareils, les modèles traditionnels de développement et de débogage des programmes embarqués, ainsi que les contraintes liées aux ressources du matériel.

2 **Une machine virtuelle OCaml générique pour appareils à faibles ressources**
Dans cette première partie, nous présenterons l'approche générique d'OMicroB, et détaillerons les mécanismes de fonctionnement de cette dernière. Nous décrirons les diverses phases d'optimisations apportées au bytecode et à son interprète afin de diminuer l'empreinte mémoire d'un programme OCaml.

3 **Une application OCaml pour Arduboy**
En seconde partie, nous proposerons de réaliser un programme complet OCaml destiné à être exécuté sur un microcontrôleur. Nous tiendrons en particulier compte des interactions du matériel avec les divers composants électroniques lui étant connectés (boutons poussoir, écran, LED, ...), et du style de programmation qu'impliquent ces interactions. Notre application a pour cible la plateforme Arduboy (un montage au format carte de crédit destiné à réaliser des petits jeux vidéo [1]) dotée d'un microcontrôleur aux ressources très limitées (2.5 kio [2] de RAM et 32 kio de mémoire flash). Ce choix de matériel limité en ressources physiques permettra de bien souligner l'intérêt des diverses optimisations d'OMicroB.

4 **Programmation haut niveau et interactions matérielles**
Enfin, nous discuterons des paradigmes de programmations adaptés aux interactions entre l'environnement et les microcontrôleurs, nombreuses dans les programmes embarqués. Nous aborderons en particulier l'exemple de la programmation synchrone (adaptée à notre approche machine virtuelle grâce à OCaLustre [6], une extension synchrone du langage OCaml) et les diverses garanties qu'elle peut offrir au développement de systèmes critiques.

Références

[1] Arduboy. https://arduboy.com.

[2] Environnement de développement Arduino. https://www.arduino.cc/en/Main/Software.

[3] OMicroB. https://github.com/stevenvar/OMicroB.

[4] Xavier Clerc. Description des instructions bytecode OCaml. http://cadmium.x9c.fr/distrib/caml-instructions.pdf.

[5] Xavier Leroy. The ZINC experiment : an economical implementation of the ML language. Technical Report RT-0117, INRIA, February 1990.

[6] Steven Varoumas, Benoît Vaugon, and Emmanuel Chailloux. OCaLustre : une extension synchrone d'OCaml pour la programmation de microcontrôleurs. In *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, 2017.

[7] Benoît Vaugon, Philippe Wang, and Emmanuel Chailloux. Les microcontrôleurs PIC programmés en Objective Caml. In *Vingt-deuxièmes Journées Francophones des Langages Applicatifs (JFLA 2011)*, 2011.

[8] Benoit Vaugon, Philippe Wang, and Emmanuel Chailloux. Programming Microcontrollers in OCaml : the OCaPIC Project. In *International Symposium on Practical Aspects of Declarative Languages (PADL 2015)*, Lecture Notes in Computer Science, 2015.

1. Plusieurs exemplaires d'Arduboy seront disponibles afin de permettre au public de réaliser les exemples proposés.
2. 1 kio = 1024 octets.

Exposés invités

MirageOS : la quête d'un OS plus petit et plus sûr

Thomas Gazagnaire

Résumé

Les techniques d'analyse statique de logiciels ont fait des progrès conséquents en terme de passage à l'échelle ces dernières années et l'analyse de programmes "réels" (plusieurs centaines de millions de lignes de code) est maintenant chose courante dans l'industrie. Bien que très utile, ces analyses sont souvent partielles et se concentrent sur un aspect particulier du problème à analyser.

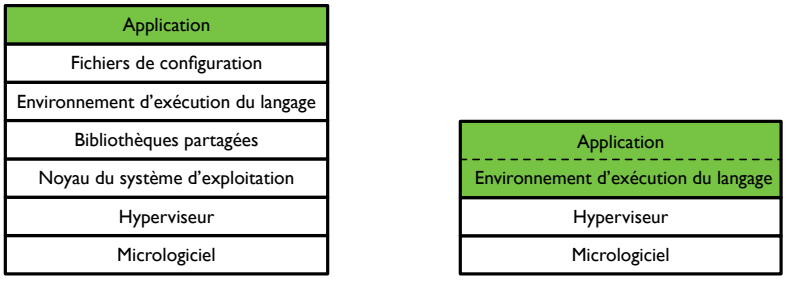
Afin d'aller plus loin il est nécessaire de prendre en compte le contexte d'exécution de l'application. Ces environnements d'exécution sont de plus en plus complexes : (*i*) ils sont découpés en couches d'abstraction indépendantes afin d'assurer l'isolation et le bon fonctionnement des applications dont ils sont responsables et (*ii*) ils sont développés par des communautés utilisant des outils et des méthodes distinctes, ayant un intérêt pour les méthodes formelles plus ou moins développé. Toute cette complexité fait que la vérification de tels systèmes est aujourd'hui impossible.

MirageOS, et plus généralement les *unikernels*, est une approche possible pour attaquer le problème de l'analyse et de la vérification de systèmes réels, environnement d'exécution compris. Cette approche consiste à (*i*) transformer un système d'exploitation en un ensemble de bibliothèques indépendantes écrites dans un langage de haut niveau (pour MirageOS nous avons choisi OCaml) – qui peuvent être certifiées et réutilisées dans différents contextes applicatifs ; et (*ii*) étendre la phase d'édition de liens aux couches basses du système d'exploitation pour générer statiquement des binaires comprenant le code minimal pour l'environnement et génère des systèmes avec une plus petite surface d'attaque, a priori plus facile à analyser et certifier.

1 Introduction

Les systèmes d'exploitation (*OS*) modernes (tels Linux, Windows, MacOS, etc.) sont structurés en différentes couches d'abstraction afin d'assurer au mieux l'isolation des différentes applications dont elles sont responsables. Ces systèmes sont chargés de gérer l'accès concurrent aux ressources physiques de la machine, en orchestrant au mieux les différentes applications et utilisateurs.

Avant de pouvoir entrer dans le détail de MirageOS dans la Section 4, il est important de comprendre pourquoi les systèmes actuels sont devenus si complexes. Pour cela, nous découpons arbitrairement cet environnement d'exécution en six couches logicielles, ordonnées par ordre croissant de privilèges de mode d'exécution, représentées dans la Figure 1 : la gestion des fichiers de configuration (Section 2.1), l'environnement d'exécution du langage (Section 2.2), les bibliothèques partagées (Section 2.3), le noyau du système (Section 2.4), l'hyperviseur (Section 2.5) et enfin le micrologiciel (Section 2.6) qui permet de programmer le processeur et le matériel. Un mode d'exécution plus élevé signifie qu'en général une couche donnée à le contrôle total de l'environnement d'exécution des couches plus basses. Le développeur exerce un contrôle fort sur les couches logicielles supérieures, ce qui lui permet d'utiliser des méthodes d'analyse, vérification ou certification de son choix ; la Section 2 montre comment ce contrôle diminue rapidement au fur et à mesure que le privilège du mode d'exécution augmente : sans contrôle ni connaissances sur le code qui est exécuté, il est impossible de mettre en place des techniques d'analyses classiques.



(a) système d'exploitation moderne (b) MirageOS

FIGURE 1 – Comparaison des différentes couches logicielles entre un système d'exploitation moderne et MirageOS.

Afin de réussir à conserver ce contrôle, l'approche que nous poursuivons – détaillée dans la Section 3 – est celle des *unikernels* [10] : des systèmes ultra spécialisés, qui au moment de la compilation du projet, lient l'application à l'ensemble des couches logicielles nécessaires å son exécution. Comme illustré sur la Figure 1, le binaire produit est minimal : il contient uniquement le code dont il a besoin pour s'exécuter sur la plate-forme de déploiement choisie et il est plus sûr : le système d'exploitation monolithique est découpé en bibliothèques composables et analysables indépendamment – en particulier le développeur garde le contrôle dans le choix du code qui sera présent à l'exécution et dans le niveau d'analyse qu'il désire y faire.

Finalement, la Section 4 présente MirageOS, une implémentation en OCaml du concept des unikernels. Il se base sur (*i*) un ensemble de signatures de module qui modélisent les interfaces du système d'exploitation ; (*ii*) une centaine de bibliothèques qui implémentent ces signatures pour les divers pilotes et protocoles supportés et (*iii*) un outil qui génère le code nécessaire pour assembler une application en fonction de la configuration de son déploiement.

2 Le contexte d'exécution

Le but de cette section est de souligner que l'application, la partie qui nous intéresse réellement et que l'on a parfois passé des années à analyser et certifier est le haut de l'iceberg. En effet, chacune des couches de la Figure 1 présente des vulnérabilités, qui sont ici illustrées par des exemples pris dans la base de "Common Vulnerabilities and Exposures" (CVE) – une base de données en ligne d'informations publiques relatives aux failles de sécurité maintenu par MITRE [1]. Chaque section donne aussi un aperçu des efforts en cours pour fixer ce genre de problèmes à chacun des niveaux. Nous proposons, dans la section 3, d'apporter une solution globale – beaucoup plus radicale – à ce type de problèmes.

2.1 Les fichiers de configuration

Les fichiers de configuration, bien que ne formant pas une couche logicielle à proprement parler, sont souvent source de problèmes. Ces données sont lues au démarrage du programme et sont fournies par l'utilisateur : elles peuvent donc être malicieuses ou défectueuses.

D'une part, ils sont destinés à être facilement lisibles par un "être humain" : leurs formats sont ainsi généralement spécifiés informellement ou partiellement ; par exemple le format

1. https://cve.mitre.org/

YAML dont la plupart des implémentations sont problématiques (Ruby : CVE-2017-2295, Python : CVE-2017-2810). D'autre part, afin de localiser les fichiers de configuration le programme doit, lors de son démarrage, interagir avec l'API POSIX d'accès aux variables d'environnement et au système de fichier qui peuvent avoir des comportements dynamiques complexes : par exemple lors de la phase d'expansion de variable, pour trouver le répertoire racine de l'utilisateur (comme CVE-2004-0747), ou lorsque l'application n'est pas lancée avec le bon utilisateur (comme CVE-2002-1379).

Le choix du format et des défauts de configuration est normalement entièrement en contrôle du développeur de l'application ; les problèmes évoqués précédemment sont donc évitables.

2.2 L'environnement d'exécution du langage

OCaml, comme la plupart des langages compilés qui gèrent la mémoire de leur programme de manière automatique a un environnement d'exécution écrit en C – le rôle de cet environnement est d'instrumenter dynamiquement le programme afin de nettoyer régulièrement sa mémoire des données qui ne sont plus utilisées. Cet environnement est soit lié statiquement lors de l'édition de lien ou lié dynamiquement à l'exécution (comme pour la CLR) – ce qui apporte son propre lot de complications, voir la Section 2.3.

Comme tout programme écrit en C, l'environnement d'exécution peut comporter des vulnérabilités liées au dépassement de borne (CLR : CVE-2013-3134 et OCaml : CVE-2015-8869). L'environnement d'exécution du langage peut aussi accéder aux variables d'environnement, notamment pour configurer certains comportements (pour déboguer ou profiler un programme) – citons CVE-2017-9772 qui permet de faire exécuter du code arbitraire à l'environnement d'exécution OCaml.

Les environnements d'exécution de langage sont contrôlables dans une certaine mesure par un développeur d'application : dans ce cadre le choix du langage de programmation utilisé est clé. La petite taille et la (relative) simplicité de l'environnement d'exécution d'OCaml (représentation homogène et non typée des données à l'exécution, bibliothèque standard minimale, etc) – ainsi que des efforts de recherche récents tels *SecurOCaml* qui essaye d'identifier un sous-ensemble plus sûr de l'environnement d'exécution du langage – en font un bon choix d'environnement d'exécution.

2.3 Les bibliothèques partagées

L'environnement d'exécution du langage (Section 2.2 ainsi que les bibliothèques qui appellent des fonctions C vont, à un moment ou à un autre, faire appel à des primitives de la bibliothèque standard du langage C (*libC*). Cette bibliothèque est utilisée par la plupart des programmes présents sur la machine : afin d'éviter d'avoir à la dupliquer pour chaque exécutable, la *libC* est généralement partagée et chargée dynamiquement lors de l'exécution d'un programme. De plus la *libC* ayant une license GPL, c'est en général la seule manière de l'utiliser pour des applications dont la licence n'est pas compatible. Il existe ainsi un certain nombre de bibliothèques C qui sont toujours chargées dynamiquement (comme GMP qui est utilsée par `Zarith`).

La *libC* est une très grosse bibliothèque partagée : plus de 500 000 lignes de code C, avec son lot typique de dépassements de bornes (comme CVE-2015-0235) et d'accès (pas toujours sûr) aux variables d'environnement (comme CVE-2017-1000366). De plus, contrairement aux fichiers de configurations et à l'environnement d'exécution du langage, ces bibliothèques partagées

sont totalement en dehors du contrôle du développeur de l'application : c'est en effet généralement le rôle de l'administrateur système de s'assurer que la dernière version des bibliothèques partagées est installée – c'est donc lui qui va décider d'une bonne partie de l'environnement d'exécution de l'application.

C'est la possibilité de redonner plus de contrôle au développeur vis à vis de l'environnement d'exécution qui a rendu populaire les approches "DevOps" basées sur les conteneurs logiciels (comme Docker). Ces technologies permettent de distribuer plus facilement des applications en contrôlant exactement quelles bibliothèques partagées sont installées en production. Parallèlement (et souvent en même temps) de nouvelles bibliothèques pour le langage C qui peuvent être liées statiquement sont proposées, telles `musl` qui a été popularisée par la distribution Alpine Linux.

2.4 Le noyau

Le système d'exploitation distingue deux modes de fonctionnement pour les programmes dont il a la charge : le mode utilisateur, qui impose une limite d'exécution à la plupart des applications écrites par un utilisateur et le mode noyau, qui permet d'ordonnancer les applications non privilégiées et de réguler leur accès aux ressources physiques de la machine (réseau, mémoire, disques, etc). L'interface entre le mode noyau et le mode utilisateur est un ensemble d'appels systèmes (ou *syscalls*) que les applications peuvent utiliser pour demander l'accès aux différentes ressources du système.

Les systèmes d'exploitation modernes sont des systèmes complexes : la dernière version du noyau Linux (4.14) comporte plus de 20 millions de ligne de code qui se partage entre pilotes de périphériques, gestion du système de fichiers, ordonnancement des applications et gestion des interfaces avec l'espace utilisateur. Ces systèmes exposent une API de plus en plus riche, soit plusieurs centaines de syscalls : 313 pour Linux[2] et autour de 500 pour Windows[3]. Ce large ensemble de syscalls couvre un spectre large de fonctionnalités ayant différents niveaux d'abstraction : par exemple, il est possible de demander un accès direct aux paquets qui transitent par une carte réseau (en utilisant des fonctions de lecture ou d'écriture sur une page de mémoire partagée entre l'espace utilisateur et le noyau), mais il aussi possible d'utiliser l'API des `socket` et d'avoir accès à un flot de données automatiquement encapsulé, par exemple, par le protocole TPC/IP. Ces protocoles réseau, qui forment la base d'Internet, sont généralement décrits à l'aide de spécifications informelles (les "Request For Comments" ou RFCs[4]) et chaque noyau (Linux, BSD, Windows, etc) implémente ces protocoles de manières subtilement différente. Comme ce code est exécuté en mode privilégié, la moindre vulnérabilité peut entraîner de lourdes conséquences, comme l'accès privilégié à la machine. C'est sans surprise que l'on retrouve ici aussi les vulnérabilités classiques des programmes écrits en C : dépassement de bornes (comme CVE-2005-0209) ou utilisation du pointeur `NULL` (comme CVE-2017-13686).

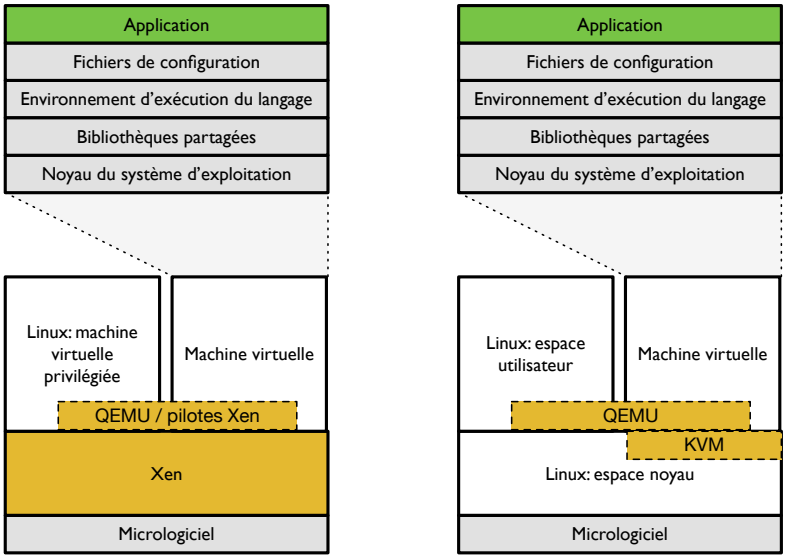
Pour un développeur d'application, il est assez difficile de choisir le noyau sur lequel son application va s'exécuter : seule la virtualisation (voir la Section 2.5) permet de spécifier la version exacte du noyau qui sera utilisée. Certains noyaux exposent des moyens de limiter le type d'appels systèmes que peut appeler une application : de manière assez grossière en utilisant les capacités POSIX ou de manière plus précise avec `seccomp`. Avec ces outils, le développeur peut annoter son programme afin de limiter son environnement d'exécution.

2. https://filippo.io/linux-syscall-table/
3. http://j00ru.vexillium.org/syscalls/nt/32/
4. https://www.ietf.org/rfc.html

2.5 La couche de virtualisation a.k.a. "le Cloud"

Depuis une quinzaine d'années, la plupart des environnements de déploiement d'applications sont virtualisés : une machine physique arbitre l'accès aux ressources physiques (processeurs, horloges, disques, réseau, etc.) entre plusieurs machines virtuelles, isolées grâce à de nouvelles instructions disponibles sur les processeurs modernes (les extensions `VT-x` pour Intel et `AMD-V` pour AMD). Chaque machine virtuelle fait tourner son propre noyau qui gère ses propres applications.

Le logiciel responsable d'orchestrer un ensemble de machines virtuelle est appelé moniteur de machine virtuelle (VMM) ou *hyperviseur*. Il en existe plusieurs types, mais les plus répandus sont Xen [1] et KVM [6], qui sont utilisés par la plupart de fournisseurs d'hébergement public de machines virtuelles (a.k.a. le "cloud") comme Amazon ou Google, et VMWare qui permet à des entreprises de créer leur "cloud" privé. La Figure 2 compare l'architecture de Xen et de KVM. Xen (Figure 2.5) est une nouvelle implémentation des fonctions de base d'un système d'exploitation : gestion de l'orchestration (ici de machines virtuelles plutôt que de processus), gestion d'accès aux ressources physiques, etc. À l'opposé, KVM (Figure 2.5) est une extension du noyau Linux : les machines virtuelles sont ordonnancées et gérées de la même manière que des processus normaux – isolation matérielle en sus.



(a) Xen (b) KVM

FIGURE 2 – Architecture des hyperviseurs.

Bien que les processus d'isolation soient a priori différents, il est intéressant de remarquer que l'accès aux périphériques d'entrées/sorties est virtualisé et géré de manière similaire par l'interposition de nouvelles couches logicielles. Dans les deux cas, l'hyperviseur s'appuie sur un système d'exploitation qui s'exécute en mode privilégié : l'espace utilisateur classique de Linux pour KVM, une machine virtuelle faisant tourner un Linux pour Xen – permettant ainsi d'utiliser l'ensemble des pilotes de périphériques distribués avec Linux. Ce système d'exploitation privilégié sert d'intermédiaire entre les périphériques de la machine physique et les pilotes de périphériques des machines virtuelles, leur permettant d'échanger des données avec le monde

extérieur. Si une machine virtuelle utilise des pilotes de périphériques classiques, le système d'exploitation privilégié doit lui les émuler. Pour Xen comme pour KVM, c'est QEMU (pour "Quick Emulator") qui est utilisé. Dans le cas de Xen, une machine virtuelle peut aussi utiliser des pilotes spécifiques pour un échange de données optimisé.

Sans surprise, comme tout projet complexe écrit en C, Xen, KVM et QEMU sont soumis aux vulnérabilités habituelles. QEMU, en particulier, a vocation à émuler de nombreux périphériques (utilisés par les machines virtuelles ou pas) : c'est un projet qui comporte plus d'un million de lignes de code en C – et qui inclut même une pile TCP/IP ! Ainsi, l'attaque VENOM (CVE-2015-3456) exploite un dépassement de borne dans un pilote de périphérique obsolète : il suffit de charger un pilote de lecteur de disquette dans une machine virtuelle pour prendre le contrôle de la machine physique !

Plus rarement, ce sont des vulnérabilités dans les algorithmes d'isolation des hyperviseurs (Xen : CVE-2017-10912 et KVM : CVE-2010-0306) qui permettent d'exécuter du code arbitraire dans un mode privilégié. Dans tous les cas, une vulnérabilité à ce niveau de la pile logicielle donne en général un accès complet à la machine et permet de compromettre l'exécution de n'importe application.

Pour un développeur d'application il est vraiment difficile de contrôler la version de Xen, KVM ou QEMU utilisée en production : le seul moyen dont il dispose est de changer de fournisseur d'infrastructure virtualisée. Cependant, ces fournisseurs sont conscients des conséquences liées aux vulnérabilités des hyperviseurs : Google s'assure par exemple qu'il y ait toujours deux couches de virtualisation pour chaque application qu'il héberge et a récemment annoncé CrosVM,[5] une alternative à QEMU pour l'émulation de pilotes. D'autres initiatives pour remplacer QEMU sont présentées dans la Section 3.3.

2.6 Le processeur et son micrologiciel

Les processeurs modernes ont leurs propres systèmes d'exploitation. D'une part, des processeurs spécialisés, initialement utilisés pour gérer le démarrage, la mise en veille et la supervision du processeur principal ; ils sont en activité même lorsque la machine est en veille. Jusqu'à très récemment, nous avions peu d'information sur les capacités exactes de ces systèmes ; nous savons maintenant qu'Intel y incorpore un système d'exploitation complet (basé sur MINIX[6], incluant même un pile TCP/IP complète pour exposer un serveur web permettant la mise à jour de son micrologiciel). D'autre part, les processeurs implémentent la spécification UEFI, un véritable système d'exploitation de taille comparable au noyau Linux, qui est lui aussi actif en permanence et tourne dans un mode privilégié par rapport au noyau du système d'exploitation sous-jacent. Ces micrologiciels contiennent eux aussi des vulnérabilités (par exemple CVE-2017-5689) qui permettent de gagner un accès privilégié à la machine en envoyant un paquet spécialement conçu au processeur.

Un développeur d'application n'a a priori aucun contrôle à ce niveau – même si certains essayent de remplacer le micrologiciel par le noyau Linux[7]. Un aspect particulièrement inquiétant de ce type de vulnérabilité est (*i*) qu'il n'est pas détectable par les couches supérieures et (*ii*) qu'il n'est pas possible d'effacer un logiciel malveillant qui refuserait de se mettre à jour : la seule solution serait alors de jeter la carte mère et de la remplacer.

5. https://chromium.googlesource.com/chromiumos/platform/crosvm/
6. http://blog.ptsecurity.com/2017/04/intel-me-way-of-static-analysis.html
7. https://www.youtube.com/watch?v=iffTJ1vPCSo

3 Les unikernels ou la simplification poussée à l'extrême

La section précédente présente le fonctionnement d'un système d'exploitation conçu pour exécuter *plusieurs* applications appartenant à *plusieurs* utilisateurs. De plus, ces systèmes sont capables de faire fonctionner tout type d'applications et en particulier des applications dont le code source n'est pas disponible (ou n'a pas encore été écrit) au moment ou ces systèmes sont déployés.

Dans cette section nous nous intéressons, au contraire, à un cadre d'exécution beaucoup plus restrictif : une seule application, un seul utilisateur, un environnement de déploiement connu et l'ensemble du code source de l'application, du système d'exploitation et des fichiers de configuration disponibles au moment de la compilation. Bien que contraignant, c'est la seule solution qui permet, en pratique de regagner du contrôle sur l'environnement d'exécution. Dans l'industrie, cela se traduit par exemple par l'utilisation des services immuables – ou l'on préfère redéployer un nouveau système plutôt que de modifier un système en cours d'exécution, ainsi que par le mouvement "DevOps", ou c'est le développeur qui est responsable de la mise en production de son application. Finalement, dans l'univers du logiciel embarqué, il est courant d'avoir un contrôle vertical complet sur la pile logicielle utilisée.

L'approche des *unikernels* est constituée de deux axes : développer un système d'exploitation de manière modulaire et étendre la phase d'édition de liens.

3.1 Un système d'exploitation vu comme un ensemble de bibliothèques

Développer un système d'exploitation de manière modulaire[3, 8] permet d'utiliser facilement ces briques logicielles dans différents contextes et d'appliquer les techniques modernes de développement logiciel (tests, certification, preuves, etc) à des composants qui sont habituellement difficiles à modifier et à maintenir. Il existe deux approches : découper des systèmes d'exploitation existants ou écrire de nouveaux systèmes à partir de zéro.

Découper des systèmes d'exploitation existants Une première approche consiste à rendre modulaire les systèmes d'exploitation existants.

— FreeBSD a fait un effort pour découper son noyau en bibliothèques indépendantes et il est possible de ré-utiliser ces composants dans différents contextes. Appelés "rump kernels" [5], ces bibliothèques permettent de compiler des applications C sans modifications majeures si le système de construction du projet est capable de gérer la compilation croisée : seul l'appel système `fork` n'est pas disponible.

— Le principal effort pour apporter plus de modularité au noyau Linux est le projet "Linux Kernel Library" (LKL) [8]. Pour le moment, ces travaux n'ont pas vocation à être intégrés directement dans le noyau ce qui pose des problèmes de maintenance. De même que pour la *libC* (Section 2.3), ce projet pose aussi la question de compatibilité de licences dès lors que l'on veut se lier statiquement aux bibliothèques de LKL (distribuées, comme le noyau Linux sous GPL).

— Microsoft a aussi commencé à découper le noyau Windows en bibliothèques indépendantes avec le project Drawbridge [12]. Ces bibliothèques ont par exemple été utilisées pour écrire une couche de compatibilité entre les noyaux Linux et Windows - ce qui a permit de porter SQL Server (qui utilise l'API Win32) vers Linux.

8. https://github.com/lkl

Écrire de nouveaux systèmes De nombreux projets ont tenté de réécrire tout ou une partie des systèmes d'exploitation avec des outils plus modernes. *FoxNet* [2] re-implémente la pile TCP/IP en SML en s'appuyant sur le langage de module et de foncteurs – MirageOS reprend cette approche et l'étend à l'ensemble du système d'exploitation (voir la Section 4). *seL4* [7] est une implémentation d'un micro-noyau formellement vérifié.

En pratique, les nouveaux systèmes d'exploitation ont un problème fondamental : ils n'arrivent pas à écrire suffisamment rapidement les pilotes des nouveaux périphériques disponibles sur le marché, ce qui les rend très vite obsolètes. Cependant, la virtualisation (Section 2.5) permet d'exposer une interface très stable pour les pilotes de périphériques virtuels : ces pilotes sont donc devenus une cible privilégiés pour les nouveaux systèmes d'exploitation, centrés autour d'un environnement d'exécution pour un langage donné. Ainsi, *MiniOS* qui était initialement un démonstrateur pour écrire une machine virtuelle utilisant les pilotes de périphériques virtuels de Xen, est au final devenu la base de la plupart des unikernels actuels. Les autres unikernels utilisent les pilotes FreeBSD rendus disponibles par rump kernel. Le Tableau 3 donne un aperçu des projets unikernels existants, classés par langage supporté, plate-forme de déploiement et la technologie utilisée pour les pilotes de périphériques.

Projet	**Langage**	**Pilotes**	**Déploiement**
MirageOS	OCaml	MiniOS & solo5	Xen, KVM
rumprun	C	rump kernel	Xen, KVM
HalVM	Haskell	MiniOS	Xen
ClickOS	C	MiniOS	Xen
IncludeOS	C++	MiniOS, solo5	Xen, KVM
DeferPanic	Go	rump kernel	Xen, KVM
LING	Erlang	MiniOS	Xen
OSv	Java	MiniOS	Xen
runtime.js	Javascript	custom	KVM

FIGURE 3 – Résumé des projets unikernels.

3.2 Étendre la phase d'édition de liens

Le deuxième aspect important des unikernels est qu'ils étendent la phase classique d'édition de liens pour analyser plus de code statiquement, en particulier en incluant l'ensemble des couches décrites dans la Section 2. Si l'éditeur de lien utilisé est optimisant, il est capable d'éliminer le code mort et de construire une application dont la surface d'attaque est minimale : uniquement le code nécessaire est présent lors de l'exécution de l'application – en particulier les pilotes de périphériques réseau et disques inutilisés ne sont pas présent dans l'exécutable final. De plus, les paramètres de configuration sont disponibles au moment de la compilation, permettant d'évaluer partiellement l'application avant même son exécution.

L'artefact produit est exécuté comme un processus unique, avec un espace d'adressage unique : en particulier le code de l'application et le code du noyau vont partager le même espace d'adressage, d'où l'utilité de s'assurer que certaines classes de problèmes liées à la corruption mémoire n'aient pas lieu et donc l'intérêt d'utiliser des langages typés de haut niveau.

À noter : le processus d'édition de lien peut s'arrêter à plusieurs niveaux – avec MirageOS il est par exemple possible de choisir entre utiliser la pile TCP/IP du système d'exploitation hôte

ou la pile TCP/IP fournie par la bibliothèque OCaml correspondante ; l'utilisateur peut choisir de spécialiser, au moment de la compilation, l'environnement d'exécution pour le déploiement de l'application : ceci nécessite un contrôle fort sur le processus de construction du projet – rumpkernel utilise la cross-compilation ; MirageOS a développé un outil en ligne de commande pour configurer la compilation du projet — la plupart des autres projets demandent à leur utilisateurs de modifier manuellement les scripts de compilation des projets à compiler.

Au final, les avantages de cette approche sont spectaculaires : le temps de démarrage d'un unikernel est de l'ordre de quelques millisecondes[9, 11] : c'est moins que le temps de retransmission d'un packet TCP, ce qui permet de démarrer des applications à la demande. D'autre part, comme des couches d'orchestration sont supprimées, les latences observées sont en général plus facile à prédire que sur des systèmes d'exploitation classiques, qui sont fortement optimisés pour le débit de transmissions. Les unikernels produisent aussi des binaires petits et qui utilisent peu de mémoire : un serveur DNS écrit en MirageOS mesure environ 200 kilooctets et utilise 10 mégaoctets de mémoire vive.

3.3 Sécuriser les moniteurs de virtualisation

La Section 2.5 a mis en évidence les nombreux problèmes liés à l'utilisation de QEMU comme composant central dans la gestion des entrées-sorties. L'évolution naturelle des unikernels à ce niveau de la pile logicielle consiste à générer un moniteur de machine virtuelle spécialisé pour la supervision de l'application considérée ; cette spécialisation inclus les pilotes de périphériques virtuels (dans la machine virtuelle) et réels (dans le système d'exploitation hôte). C'est l'approche suivie par Solo5 [15] développé par IBM et qui permet à MirageOS de générer, en plus d'un exécutable complet entièrement spécialisé comme vu dans les sections précédentes, un moniteur entièrement spécialisé qui va s'exécuter dans le système d'exploitation hôte et dont la seule tâche est de superviser l'application donnée. L'interface exposée par solo5 est minimale et facilement portable : un moniteur solo5 peut superviser une application déployée sur KVM+QEMU, FreeBSD+byhve, KVM+moniteur spécialisé ou le microkernel certifié *Muen* [9].

4 MirageOS : un exemple d'unikernel écrit en OCaml

MirageOS est un environnement de programmation pour construire des applications système (réseaux, stockage) performantes et portables. Le code peut être développé et testé dans un environnement classique (avec Linux ou MacOS) puis compilé en un artefact complètement indépendant, entièrement spécialisé pour l'environnement de déploiement tel Xen ou KVM (ou directement sur des machines réelles). MirageOS est entièrement écrit en OCaml et comporte un écosystème riche de plus d'une centaine de bibliothèques. Celles-ci fournissent des outils pour développer des applications système, en particulier liées au réseau car MirageOS inclut des bibliothèques pour les protocoles TCP/IP, HTTP et TLS. La gestion de la concurrence est basée sur `lwt` et il n'y a pas de multi-tâche préemptif (car l'appel système `fork` n'est pas disponible).

4.1 Utiliser des signatures de module standardisées

Les bibliothèques de MirageOS partagent toutes un ensemble standardisé de signatures de modules, qui définissent comment les différents composants d'un système d'exploitation

9. https://muen.codelabs.ch

peuvent interagir. Ces signatures comportent en général un type abstrait qui stocke l'état du périphérique ou protocole modélisé, mais il n'expose pas de constructeurs pour ce type : ainsi chaque implémentation est libre d'exposer des constructeurs utilisant les paramètres dont elle a besoin.

Un exemple simplifié de l'interface `Mirage_net.S` qui modélise une carte réseau peut s'écrire de la manière suivante :

```
module type S = sig
  type t
  type buffer = Cstruct.t
  type macaddr = Maccadr.t
  type error = private [> `Unimplemented | `Disconnected]
  val pp_error: Format.formatter → error → unit

  val write: t → buffer → (unit, error) result Lwt.t
  (** [write nf buf] outputs [buf] to netfront [nf]. *)

  val listen: t → (buffer → unit Lwt.t) → (unit, error) result Lwt.t
  (** [listen nf fn] is a blocking operation that calls [fn buf] with
      every packet that is read from the interface. The function can
      be stopped by calling [disconnect] in the device layer. *)

  val mac: t → macaddr
  (** [mac nf] is the MAC address of [nf]. *)
end
```

Une application MirageOS est un foncteur qui utilise ces signatures de module et qui possède une fonction `start`. L'exemple suivant montre une application qui affiche la taille de paquets que reçoit l'interface réseau à laquelle elle est connectée :

```
(* unikernel.ml *)
module Main (N: Mirage_net.S) = struct
  let start n =
    let print buf =
      Logs.info (fun l → l "[%d]" (Cstruct.len %buf));
      Lwt.return ()
    in
    N.listen n print
end
```

Une application plus complexe, telle que le site de MirageOS [10] peut avoir de nombreux paramètres, pour avoir accès, par exemple à plusieurs espaces de stockage de clé/valeur en lecture seulement (représentés par la signature `Mirage_kv_ro.S`) et une interface pour gérer un serveur HTTP afin d'exposer des pages web (représenté par `Cohttp_lwt.Server`) :

```
(* unikernel.ml *)
module Main (FS: Mirage_kv_ro.S) (TMPL: Mirage_kv_ro.S) (S: Cohttp_lwt.Server)
  = struct ... end
```

4.2 Un écosystème de bibliothèques composables

Les signatures de module forment la colonne vertébrale de MirageOS. Une centaine de bibliothèques, disponibles sur *opam*, implémentent ces interfaces. Ces bibliothèques sont de

10. https://mirage.io/

deux types.

D'une part, il existe des implémentations concrètes de ces signatures, accompagnées d'un ou plusieurs constructeurs pour les types abstraits. Historiquement, ces implémentations concernaient les pilotes de périphériques virtuels pour Xen, développés dans le cadre de *XenServer* [14]. Aujourd'hui, elles couvrent un large spectre de protocoles réseau et de stockage.

Par exemple un pilote de carte réseau qui implémente `Mirage_net.S` en lisant une interface réseau en mode "raw" possède cette signature :

```
include Mirage_net.S
val connect: string → t Lwt.t
(** [connect i] opens the interface [i] (for instance "eth0") in raw mode. *)
```

D'autre part, il existe des foncteurs qui prennent en paramètre une ou plusieurs signatures de modules de MirageOS. Ces foncteurs implémentent eux aussi des signatures de module de MirageOS, habituellement de plus haut niveau et incluent un ou des constructeurs pour les types abstraits de ces signatures. C'est le cas par exemple de la couche ethernet : elle est paramétrée par une interface réseau (`Mirage_net.S`) et elle fournit une abstraction du protocole ethernet (`Mirage_ethif.S`) ainsi qu'un constructeur :

```
module Make (N:Mirage_net.S) : sig
  include Mirage_ethif.S with type netif = N.t

  val connect : ?mtu:int → netif → t Lwt.t
  (** [connect ?mtu netif] connects an ethernet layer on top of the raw
      network device [netif]. The Maximum Transfer Unit may be set via the
      optional [?mtu] parameter, otherwise a default value of 1500 will be
      used. *)
end
```

Finalement, la plupart des bibliothèques écrites en OCaml pur (sans liens avec du code C) et qui n'utilisent pas le module `Unix` peuvent être intégrés facilement dans une application MirageOS– le Tableau 4 donne aperçu des bibliothèques disponibles lorsque des interactions avec le système sont nécessaires.

Domaine	Bibliothèques
Interfaces Réseau	tuntap, vmnet, rawlink
Protocoles Réseau	ethernet, ARP, ICMP, IPv4, IPv6, UCP, TCP, DHCP, DNS, TFTP, HTTP
Protocoles de Stockage	FAT32, Git, tar, AES-CCM, ramdisk, B-trees
Sécurité	x509, ASN1, TLS, OTR
Crypto	MD5, SHA1, SHA224, SHA256, SHA384, SHA512, BLAKE2B, BLAKE2S, RMD160, 3DES, AES, DH, DSA, RSA, Fortuna, ECB/CBC/CCM/GCM modes

FIGURE 4 – Aperçu des bibliothèques disponibles avec MirageOS.

Certaines de ces bibliothèques ont fait l'objet d'attentions particulières : c'est le cas du protocole TLS [4] qui a été testé de manière rigoureuse [4] et du format de stockage des B-trees dont le code est généré à partir d'une spécification écrite en Isabelle/HOL [13].

4.3 Un outil pour configurer le contexte d'exécution

Avant d'être déployée avec un environnement d'exécution spécialisé, une applications MirageOS a vocation à être développée comme une application classique. Il est donc courant de vouloir configurer l'application selon différents modes, par exemple :

— `mirage configure --target=unix --net=socket` configure un binaire Unix dont le trafic réseau utilise l'API des *sockets* exposées par le noyau, ainsi que le ferait une application classique. Ce mode est utile pour que les outils classiques de déboguage et de profilage puissent fonctionner (avec les limites habituelles liées à `lwt`).

— `mirage configure --target=unix --net=direct` configure un binaire Unix servant un trafic HTTP via une pile réseau écrite en OCaml, en utilisant l'interface réseau par défaut (`tap0` pour Linux) en mode "raw" Ce mode est utile pour tester la pile réseau.

— `mirage configure --target=xen --net=direct` configure un unikernel qui pourra être exécuté par Xen et qui utilise une interface réseau virtuelle. C'est ce mode qui est utilisé lors de la mise en production.

Le lien entre les signatures de module standardisées et les différentes implémentations disponibles est donc fait par l'outil *mirage*. Cet outil prend en entrée une application MirageOS :

1. un fichier `unikernel.ml` contenant un foncteur `Main` ayant au moins une fonction `start` ;
2. un fichier `config.ml` contenant une représentation de ce foncteur et de ses paramètres ;
3. des options en lignes de commande pour sélectionner les paramètres de déploiement.

Afin de spécifier la forme de l'application MirageOS à compiler, le fichier de configuration utilise une représentation abstraite d'un sous-ensemble du langage des modules d'OCaml qu'expose la bibliothèque *libmirage* (Figure 5). Les signatures de foncteur sont déclarées en utilisant le combinateur `@->`, qui représente la composition des arguments et la fonction `foreign` permet de nommer un foncteur. Les applications de foncteur sont représentées par le combinateur `($)`. La fonction `register` permet d'enregistrer le module principal de l'application.

Considérons, par exemple, le fichier de configuration suivant :

```
(* config.ml *)
let main = foreign "Unikernel.Main" (network @→ job) in
register "console" [ main $ default_network ]
```

La bibliothèque *libmirage* expose chaque signature de module standardisée comme une valeur de type `typ` : dans l'exemple précédent `network` représente ainsi la signature `Mirage_net.S`. *libmirage* expose aussi une valeur de type `impl` pour chaque implémentation disponible, ce qui permet de configurer, manipuler et assembler des environnements d'exécution complexes au sein même du langage. De plus, *libmirage* expose un mécanisme standard pour déclarer des paramètres à passer sur la ligne de commande au moment de la configuration ou de l'exécution. Ainsi, la variable `default_network` dans l'exemple précédent est une valeur de type `impl` qui "lit" sur la ligne de commande le mode de déploiement de l'application et sélectionne les paquets *opam* à installer ainsi que les bibliothèques à lier à l'application finale.

Grâce à ces mécanismes d'introspection, *mirage* peut par exemple décrire toutes les options possibles d'un projet avec `mirage -help`, ou l'ensemble des modules liés dans un mode d'exécution avec `mirage describe`.

```
(** mirage.mli *)

type 'a typ
(** The type of values representing module types. *)

val (@→): 'a typ → 'b typ → ('a → 'b) typ
(** Construct a functor type from a type and an
    existing functor type. This corresponds to
    prepending a parameter to the list of functor
    parameters. *)

type 'a impl
(** The type of values representing module implementations. *)

val ($): ('a → 'b) impl → 'a impl → 'b impl
(** [m $ a] applies the functor [a] to the functor [m]. *)

val foreign: string → → 'a typ → 'a impl
(** [foreign name libs packs constr typ] states that the module named
    by [name] has the module type [typ]. *)
```

FIGURE 5 – Représentation d'un sous-ensemble du langage des modules en OCaml.

5 Conclusion

La complexité des environnements d'exécution généralistes de ces systèmes rend aujourd'hui leur analyse impossible. Nous avons montré qu'il était possible au contraire d'architecturer un environnement d'exécution ultra spécialisé autour de chaque application et que les bénéfices de cette approche en terme de contrôle de l'organisation logicielle permettent d'envisager une analyse et une certification de ces nouveaux artefacts. Cependant, il reste encore du chemin à faire afin de rendre MirageOS encore plus sûr : continuer l'effort de vérification formelle des bibliothèques disponibles ; et surtout certifier l'environnement d'exécution du langage OCaml, qui devient maintenant l'élément de sûreté critique de ce nouveau système. Finalement, il serait intéressant d'explorer l'utilisation de MirageOS (ou d'un système équivalent) pour remplacer les micrologiciels des processeurs.

Remerciements Un grand merci à Louis Gesbert, Frédéric Bour et Romain Calascibetta pour leurs retours constructifs et suggestions.

Références

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5) :164–177, October 2003.

[2] Edoardo Biagioni, Robert Harper, and Peter Lee. A Network Protocol Stack in Standard ML. *Higher-Order and Symbolic Computation*, 14(4) :309–356, Dec 2001.

[3] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel : an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, Colorado, USA, 1995. ACM.

[4] David Kaloper-Meršinjak, Hannes Mehnert, Anil Madhavapeddy, and Peter Sewell. Not-Quite-So-Broken TLS : Lessons in Re-Engineering a Security Protocol Specification and Implementation. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 223–238, Washington, D.C., 2015. USENIX Association.

[5] Antti Kantee. *Flexible Operating System Internals : The Design and Implementation of the Anykernel and Rump Kernels*. PhD thesis, Aalto University, Otakaari 1, 02150 Espoo, Finland, 2012.

[6] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. KVM : the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

[7] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4 : Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[8] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul T. Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal of Selected Areas in Communications*, 14(7) :1280–1297, 1996.

[9] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, Dave Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, Jon Crowcroft, and Ian Leslie. Jitsu : Just-In-Time Summoning of Unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, Oakland, CA, 2015. USENIX Association.

[10] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels : Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM.

[11] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459–473, Berkeley, CA, USA, 2014. USENIX Association.

[12] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library OS from the top down. In *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 291–304, Newport Beach, California, USA, 2011. ACM.

[13] Tom Ridge. A B-tree library for OCaml. In *OCaml Workshop 2017*, 2017.

[14] David Scott, Richard Sharp, Thomas Gazagnaire, and Anil Madhavapeddy. Using functional programming within an industrial product group : perspectives and perceptions. In *Proc. 15th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 87–92, Baltimore, Maryland, USA, September 27–29 2010.

[15] Dan Williams and Ricardo Koller. Unikernel Monitors : Extending Minimalism Outside of the Box. In Austin Clements and Tyson Condie, editors, *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016*. USENIX Association, 2016.

Guaranteeing data provenance and integrity by typing, and application to modular, bug-free, automated reasoning

Stéphane Graham-Lengrand

CNRS - École Polytechnique - SRI International
graham-lengrand@lix.polytechnique.fr

Who produced this data? Did anyone tinker with it? These questions about data provenance may arise if, e.g., one wants to trace which packages from an operating system were involved in producing some data. One may trust some software components and not others, as software may be buggy or malicious. Indeed a motivation for tracing provenance is often the question of *trust*.

Trust is a central issue whenever we use a machine to help us with mathematical reasoning. Data in this case may simply be the information that "Theorem A holds". Who claims it? Which piece of code was able to affect this data?

In this context, trust is often conveyed, as best as possible, by the existence of a software component, usually called the *kernel*, upon which the correctness of such data solely relies. Any other piece of code is prevented from affecting (the correctness of) this data, which can be achieved in different ways.

One of these ways is to have a *proof-checker* as kernel: it runs last, after some data has been produced that supposedly justify why Theorem A holds, and the proof-checker checks whether this data does justify the claim. The justifying data, depending on its nature and on how much work the proof-checker has to do, can be a *trace*, a (proof) *certificate*, a *proof-term* or any kind of *proof object*. This is used for instance in the proof assistant Coq [Coq], where the kernel, at the QED point, checks a proof object that has been constructed by the user and/or Coq's automation. This is also the most prominent approach in automated reasoning for trusting the produced outputs: Automated Theorem Provers (in a large sense) are sophisticated search engines, and can accumulate just enough information during their runs so that, at the end, some form of proof certificate can be recovered and fed to a proof-checker.

Another way is to design the theorem proving software with an architecture that identifies a kernel upon which the correctness of theorems solely depends, and allows intertwining calls to the kernel within the execution of non-kernel code. The data produced by kernel calls should simply not be corrupted by non-kernel code, in the rest of the execution: its integrity should be guaranteed. This is a typical application of some programming abstractions that can be found in most languages used today, be they object-oriented or functional: *hiding* object fields or type definitions while only *exporting an API* of primitives that manipulate the data in controlled, trusted ways.

At the origins of the ML language family (typed functional languages such as SML, OCaml, etc) lies such an abstraction for trusted theorem proving: LCF [Mil79, GMW79]. In the LCF approach, used for instance in Isabelle [Isa] and most of the HOL-based proof assistants, a type `theorem` is used for theorems, i.e. formulae that can be proved. While type `theorem` may simply be an alias for the type of formulae, this is only known inside the LCF kernel and hidden by the kernel's interface. Instead, the interface exports primitives such as

```
modus_ponens : theorem -> theorem -> theorem
```

which takes as arguments two theorems, supposedly of the form $A \Rightarrow B$ and A, respectively, and returns B in type `theorem`. In case the arguments are not of the correct form, the primitive

fails, which de facto amounts to performing a proof-checking step. The invariant that these primitives satisfy is that every effectively constructed inhabitant of type `theorem` is indeed a theorem. As the definition of type `theorem` is hidden outside the kernel, every inhabitant of `theorem` is necessarily constructed by the kernel's primitives, which ensures the invariant is preserved regardless of the execution of non-kernel code. If the whole computation outputs an inhabitant of `theorem` that gets revealed as formula A, then the claim that Theorem A holds is *correct-by-construction*, if of course the kernel is correct. Similar abstractions can be found in systems such as Coq [Coq] (for well-formed typing environments), Why3 [BFM+13], and many others.

The LCF abstraction is an example where typing is used to guarantee the provenance (from the kernel), as well as the integrity, of inhabitants of type `theorem`. Further use of typing can help guaranteeing provenance and integrity of data, beyond the simple case of one software component producing the data to be traced.

Similarly to LCF where type `theorem` is used as an earmark for kernel-produced data, the types that we use for traced data are *tainted* by the components that have produced the data. Type polymorphism in functional programming languages such as ML can be used to handle an arbitrary range of components, using for example a type `'taint data` whose instances `taint1 data` and `taint2 data` are used for data produced by Component1 and Component2, respectively.

To secure the correspondence between taints and components, building an inhabitant of type `taint data` for some specific taint requires going through a trusted authority that taints the data's type according to the producer. The `taint` type corresponding to a component can be an abstract type, of which only the component knows an inhabitant. That inhabitant works as the component's "passport", shown to the authority's clerk in order to taint the data, but never communicated otherwise, and not built into the data either. The following simple OCaml module implements this idea for data made of strings:

```
module Authority = (struct
  type 'taint data = Data : string -> 'taint data
  let build _passport blah = Data blah
end: sig
  type 'taint data = private Data : string -> 'taint data
  val build : 'taint -> string -> 'taint data
end)
```

The `'taint` parameter of type `data` could be seen as a *phantom type*, not really being used in the definition of `'taint data` itself: from that definition alone, any value `Data(s)`, for some string `s`, is of type `taint data` for any `taint`. But the role of the module is to restrict this freedom by only allowing the construction of `Data(s)` in type `taint data` if an inhabitant of `taint` can be shown, implementing the notion of "passport". The `private` keyword in the module's interface ensures that external code can always read the data as a string but can only build it through the use of the authority's `build` function.

Note that in this approach, data provenance information is carried in the types, not in the data itself. This has the advantage of ensuring data provenance and integrity statically, using OCaml's type-checker, and not induce any runtime overhead. But it may also be convenient to allow computation to benefit from the provenance information, which can be modelled using component *handlers*. The handlers can be connected to the taints by making the handlers' type a *Generalized Algebraic DataType* (GADT) [CH03, XCC03]. Moreover, in order to keep the range of components open, that GADT can be an *extensible* one, requiring each component to declare not only their taint type but also their handler, via an extension of the GADT. The

introduction of the extensible GADT for handlers, and the definition of the (minimal) interface for components can be given in OCaml as follows (note that `..` below is OCaml syntax, not an ellipsis):

```
type _ handlers = ..
module type Component = sig
  type taint
  type _ handlers += Hdl : taint handlers
end
```

Data and handlers can then be paired, in a way that is safe in the sense that data will only be paired with the handler for the component that produced it. This is ensured by the matching `'taint` parameter in the following pairing type, which is yet another GADT that existentially quantifies over the `'taint` parameter:

```
type data_handler = DataHdl : ('taint Authority.data)*('taint handlers) -> data_handler
```

Computation can then manipulate data together with a handler for a component, safely assuming that the component produced the data, so that provenance information is used dynamically.

Examples can be given where components are agents such as Alice and Bob, and invariants that are guaranteed by using this simple set-up are, e.g., that Bob only reveals his name if Alice has asked the question.

Back to theorem proving, this set-up can lift the LCF principles to other contexts than that of the kernel vs. non-kernel architecture. That architecture makes sense when theorem proving is used to formalise a mathematical corpus, decomposing its developments into the atomic steps and axioms of a unique foundational framework such as set theory or type theory. This is one of the main drives behind Interactive Theorem Proving, with global consistency being one of the top priorities.

But our set-up can help structuring more modular enterprises of theorem proving, where we do not have a monolithic kernel, but we simply want to use various reasoning components, some of which we trust more than others. We have used it for instance in the Psyche platform [GL13], the *Proof-Search FactorY for Collaborative HEuristics*. Psyche is centered around the process of proof-search rather than proof-checking or proof-construction, adapting the LCF approach accordingly, so as to keep strong correctness guarantees. With various proof-search processes interacting with each other, especially in the context of Satisfiability-Modulo-Theories (SMT), it was highly desirable to consider more refined guarantees than the binary invariant guaranteed-correct / not-guaranteed-correct. Tracing which piece of code contributes to which output was the drive behind the present development. In SMT-solving, this approach guarantees in particular that the theorems are really proved in the specified combination of theories. The CDSAT branch of Psyche [CDS], which implements the Conflict-Driven Satisfiability framework [BGLS17, BGLS18] for SMT-solving, does not simply trace which theories are involved in a theorem, but it also separates, for each theory, the code that implements the theory-specific inferences (relatively small and trusted) from the code that controls the theory-specific heuristics for the search. This separation is used, as in LCF, to make sure that only the former can affect the nature of the produced answer (Provable or Not Provable).

This separation also opens the door to an extensive experimentation of various proof-search strategies, as the correctness guarantees provided by typing are never jeopardised. This constitutes an approach to the "Strategy Challenge in SMT-solving" brought forward by De Moura and Passmore [dMP13], urging the SMT community to take inspiration from the technical solutions used in, e.g., the LCF community.

References

[BFM+13] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. The Why3 platform 0.81, 2013. Tutorial and Reference Manual. http://hal.inria.fr/hal-00822856

[BGLS17] M. P. Bonacina, S. Graham-Lengrand, and N. Shankar. Satisfiability modulo theories and assignments. In L. de Moura, editor, *Proc. of the 26th Int. Conf. on Automated Deduction (CADE'17)*, volume 10395 of *LNAI*. Springer-Verlag, 2017.

[BGLS18] M. P. Bonacina, S. Graham-Lengrand, and N. Shankar. Proofs in conflict-driven theory combination. In J. Andronick and A. Felty, editors, *Proc. of the 7th Int. Conf. on Certified Programs and Proofs (CPP'18)*. ACM Press, 2018.

[CDS] The CDSAT system. https://github.com/disteph/cdsat

[CH03] J. Cheney and R. Hinze. First-class phantom types. Technical Report CUCIS TR2003-1901, Cornell University, Ithaca, NY, USA, 2003.

[Coq] The Coq Proof Assistant. http://coq.inria.fr/

[dMP13] L. M. de Moura and G. O. Passmore. The strategy challenge in SMT solving. In M. P. Bonacina and M. E. Stickel, editors, *Automated Reasoning and Mathematics - Essays in Memory of William W. McCune*, volume 7788 of *LNCS*, pages 15–44. Springer-Verlag, 2013.

[GL13] S. Graham-Lengrand. Psyche: a proof-search engine based on sequent calculus with an LCF-style architecture. In D. Galmiche and D. Larchey-Wendling, editors, *Proc. of the 22nd Int. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods (Tableaux'13)*, volume 8123 of *LNCS*, pages 149–156. Springer-Verlag, 2013.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: a mechanized logic of computation*, volume 78 of *LNCS*. Springer-Verlag, 1979.

[Isa] The Isabelle theorem prover. http://isabelle.in.tum.de/

[Mil79] R. Milner. LCF: A way of doing proofs with a machine. In J. Becvár, editor, *Proc. of the the 8th Int. Symp. on Mathematical Foundations of Computer Science*, volume 74 of *LNCS*, pages 146–159. Springer-Verlag, 1979.

[XCC03] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In A. Aiken and G. Morrisett, editors, *Proceedings of the Thirtieth SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 224–235. ACM, 2003.

Articles

A Value-based Memory Model for Deductive Verification*

Quentin Bouillaguet[1,2], François Bobot[1], Mihaela Sighireanu[2], and Boris Yakobowski[1]

[1] CEA, LIST, Software Reliability Laboratory, France
firstname.lastname@cea.fr
[2] IRIF, University Paris Diderot and CNRS, France
firstname.lastname@irif.fr

Abstract

Collaboration of verification methods is crucial to tackle the challenging problem of software verification. This paper formalizes the collaboration between Eva, a static analyzer, and WP, a deductive verification tool, both provided by the Frama-C platform, and concerned with the verification of C programs. The collaboration focuses on verification of programs using pointers, where most deductive verification tools are limited to C programs that do not contain union types, pointer arithmetics, or type casts. We remove some of these limitations from WP by transferring information computed by Eva, which soundly supports these features. We formalize this collaboration by defining a memory model that captures the information on memory inferred by the points-to analysis of Eva, and complies with the abstract memory model used by WP to generate verification conditions. The memory model defined combines a raw memory model with a typed memory model. It captures the low-level operations on pointers allowed by C and provides information about the partition of the memory in disjoint memory regions. This expressivity increases the realm of programs dealt by WP and its efficiency in generation of verification conditions.

1 Introduction

Software verification is a challenging problem for which different solutions have been proposed. Two of these solutions are deductive verification (DV) and static analysis (SA). While deductive verification is interested in checking precise and expressive properties of the input code, static analysis targets checking a fixed class of properties. This loss in expressivity of properties in static analysis is counterbalanced by its high degree of automation. Conversely, deductive verification requires efforts from the user that has to specify the properties to be checked and other annotations, e.g., loop invariants. Using these specifications, DV tools build verification conditions which are formulas in various logic theories and send them to specialized solvers. SA tools may also need to solve constraints, but they generally employ ad-hoc algorithms that soundly decide the satisfiability of constraints.

The complementarity of the two methods have motivated to search new methods that combine deductive verification and static analysis. One of the methods proposed, e.g., [1], consists in first applying static analysis to check some fixed properties of the input code and to infer invariants about the states of the program at each program point. The kind of invariants that may be inferred are interval constraints on the integer or float variables, pointer aliasing, and the shape of the memory. Then, these inferred invariants are injected in the specifications used by the deductive verification tool in order to ease the specification task or to strengthen the existing specifications.

*This work was partially supported by ANR project VECOLIB, grant ANR-14-CE28-0018-03.

Such a collaboration is possible when two conditions are met. The first condition is the availability of a channel that eases the dialogue between such tools. The second condition is the soundness of this dialogue, i.e., the translation of invariants inferred by static analysis into the specification logic used by the deductive verification tool preserves the meaning of properties. The Frama-C platform [19] satisfies the first constraint because it includes both static analysis and deductive verification tools, Eva resp. WP plugins, for the verification of C programs. These tools communicate by associating validity statuses (True, False, Unknown) to the *alarms* present in the program. Alarms are logical assertions, written in the ACSL [5] specification language, that eliminate the runs in which a statement leads to an undefined behavior (e.g., `assert(d != 0);` removes undefined behavior for the statement `r = 1/d;`). Collaboration by alarms is sound due to the common semantics of program expressions, but it is not extensible to more complex properties, e.g., separation of memory regions.

In this paper, we propose a solution that enables a more intimate collaboration of these methods. The idea is to transfer information from SA to DV in the form of a logic formula that encodes the state invariant inferred by the static analysis. This idea has been already applied in the context of test generation in [1], but only for C programs without pointer manipulation. Our solution may infer properties about pointers like aliasing, pointer restricted to a fixed memory region, etc. These properties are valid in a model of the program memory that captures pointer manipulation, e.g., pointer arithmetics, updates at arbitrary locations, type casts. One issue is that this specific memory model is not currently available in WP.

Our work provides this missing link between Eva and WP. We propose a new memory model that captures precisely the one used in Eva and satisfies the constraints of the abstract memory model used by WP to generate verification conditions. A consequence of this connection is that the points-to information computed by Eva is used by WP, like in [28], to know which memory regions are separated. This information is important for WP to generate smaller verification conditions. Indeed, WP may use different memory models for these regions, thus generating smaller constraints, or avoiding their generation altogether.

The memory model defined in this work is actually independent of the static analyzer used. Indeed, we define an interface that contains which information the memory model requires from the abstract values computed by the static analyzer. We implemented this interface for the abstract domain used by Eva, but it could be implemented for other abstract domains that compute abstractions of points-to relations.

To motivate such a collaboration, let us consider the C code from Figure 1. The function `copy` copies the first two elements of the array pointed to by `b` into the one pointed to by `a`. The ACSL specification of the function requires the validity of memory locations that are accessed, while the postcondition expresses that `a` ends up containing the initial contents of `b`. Notice that this specification is incomplete: if the cells `a[0]` and `b[1]` are not separated, the post-conditions is *not* verified. However, in all the calls of `copy` from `main`, this separation hypothesis *is* verified, and Eva is able to infer automatically the separation property for this program. Hence, by injecting this information in WP, we could prove the post-condition.[1]

```
/*@ requires \valid(&a[0..1]) && \valid(&b[0..1]);
      ensures a[0] == \old(b[0]) && a[1] == \old(b[1]);
*/
void copy(int* a, int* b) {
  a[0] = b[0]; a[1] = b[1];
}
extern int t[4], u[5];
void main() {
  copy(&t[0], &u[2]);
}
```

Figure 1: Verification in presence of aliasing

[1] Of course, we are weakening the overall verification, as the specification of the function `copy` is no longer valid in all possible calling contexts. In essence, we assume the *closed world* hypothesis, where `copy` is supposed

Related work Several memory models have been proposed to capture the semantics of programs manipulating pointers. All these models view the memory as a collection of disjoint regions. Two main classes may be distinguished: (i) the regions are typed by the value stored, which is a good abstraction for type-safe languages, (e.g., Java-like [2,4], HOL [23]) and (ii) the regions are seen as raw arrays of bytes to capture low-level manipulations of memory in C and it is used in static analyzers (e.g., HAVOC [10], VCC [6], MemCAD [9], Infer [8], SMACK [26], Eva [7]) or in deductive verification tools, (e.g., Caduceus [15] or VeriFast [18]). Most of tools based on the second class of memory models allow also typing of memory regions. The abstract memory model of WP supports both classes of models and WP provides concrete implementations for both classes [13].

The use of abstract memory model to capture refinements of these models has been proposed in the CompCert project [22]. The abstract memory model of WP is inspired by the second version of this project [21], but the concrete model we propose has not been considered in CompCert. In [27] is proposed a method to design static analyzers based on an abstract memory model. Eva is not built following these principles for efficiency reasons.

In [17] is used a static analysis based on region inference for the partitioning of a memory model. The analysis employed is less precise than the points-to analysis in Eva because the loss of precision for one location could force many precise locations to be collapsed in the same region. Recent work [28] proposes a precise points-to analysis to infer information about the separation of memory regions in order to decrease the size of verification conditions generated by deductive verification tools. Although Eva is doing a less precise analysis, it is still able to infer such separation properties, and we define in addition a formalized channel to transfer such information to WP. The authors of [6] explore different memory models to generate with VCC a benchmark of problems for SMT solvers. By implementing an additional concrete memory model for WP, we can use it to provide such benchmarks.

Separation Logic [24] is used in many verification tools for C (e.g., GRASSHoper [25], HIP/Sleek [11], Infer, VeriFast) due to the simpler specification of disjointness between memory regions. ACSL includes a separating conjunction operator (understood by WP and Eva), but it is far weaker than the standard separating conjunction operator.

Paper organization Section 2 introduces a simple programming language that exhibits the pointer features we are interested in. On this language, we illustrate the notions of abstract memory model and verification condition generation based on this model. Section 3 provides two memory model employed by WP for the verification of C programs and discusses their advantages and limitations. Section 4 presents the Eva static analyzer and its memory model. The memory model we formalized for Eva in WP is presented in Section 5. We conclude by presenting the experimental results obtained in Section 6.

2 Deductive Verification for a Toy Language

To illustrate our contribution, we introduce in this section a toy programming language with support for record types, pointer arithmetics, type casts, and updates at arbitrary locations in the memory. We define its syntax and its semantics with respect to an abstract memory model. Then, we sketch the generation of verification conditions for this language in a first order logic based on a generic interface with the underlying memory model.

to be called only from the `main` function.

$\mathtt{n} \in \mathbb{N}, \mathtt{i} \in \mathbb{Z}$	integer constants	`arith`	arithmetic type in $\{\mathtt{i8}, \mathtt{u8}, \mathtt{i16}, \ldots, \mathtt{u64}\}$
$\mathtt{rt} \in \mathtt{crec}$	record type names	$\mathtt{f} \in \mathtt{cfld}$	field names
$\mathtt{v} \in \mathtt{cvar}$	program variables	$\mathtt{op} \in \mathtt{O}$	unary and binary arithmetic operators

program types	$\mathtt{ctyp} \ni \mathtt{t}$	$::= \mathtt{u} \mid \mathtt{rt} \mid \mathtt{t}[\mathtt{n}]$
scalar types	$\mathtt{styp} \ni \mathtt{u}$	$::= \mathtt{arith} \mid \mathtt{t}\ \mathtt{ptr}$
expressions	$\mathtt{expr} \ni \mathtt{e}$	$::= \mathtt{i} \mid \mathtt{lv} \mid \mathtt{a} \mid (\mathtt{arith})\mathtt{e} \mid \mathtt{op}\ \mathtt{e} \mid \mathtt{e}\ \mathtt{op}\ \mathtt{e}'$
addresses	$\mathtt{addr} \ni \mathtt{a}$	$::= \&\mathtt{v} \mid \mathtt{e} \mid \mathtt{a}.\mathtt{f} \mid \mathtt{a}[\mathtt{e}]$
left values	$\mathtt{lval} \ni \mathtt{lv}$	$::= *_{\mathtt{u}}\mathtt{a}$
statements	$\mathtt{stmt} \ni \mathtt{s}$	$::= \mathtt{lv} = \mathtt{e} \mid \mathtt{assert}\ \mathtt{e}$

Figure 2: Syntax of our C-like toy language

2.1 A Toy Language

Figure 2 lists the constructs of this language. For simplicity, we consider only integer (signed or unsigned) arithmetic types. User defined types are pointer types, static size array types, and record types. A record type declares a list of typed fields with names from a set `cfld`; for simplicity, we suppose that each field has a unique name. Expressions combine integer constants and address expressions using operators in `O` (that includes arithmetic operations, equality and relational comparisons, left and right shifts and bitwise operations), and casts into an arithmetic type. Address expressions contains constant addresses (i.e., locations of program variables), expressions in `expr` of pointer type, an address shifted by a field in `a.f`, and an address shifted by a natural value obtained from the valuation of an integer expression in `a[e]`. A left value of some type `t` is obtained by dereferencing an address expression `a` of type `t ptr` in $*_{\mathtt{t}}\mathtt{a}$. We consider only simple statements for assignment and assertion testing. Classic control statements can be dealt using standard techniques in deductive verification.

We consider only well typed programs. In the following, we model the results of the type checker on a program by a set of semantic functions as follows. For program types, $|.| : \mathtt{ctyp} \to \mathbb{N}$ maps each type to its size in bytes (like `sizeof`). Also, we consider the conversion function $\mathtt{convert}(v, \mathtt{arith})$ for integer values v to some arithmetic type as defined, e.g., in [21]. For a field `f`, $\mathtt{offset}(\mathtt{f})$ gives the offset of this field in its definition record. For any field, variable, expression, or address, $\mathtt{cty}(.)$ returns its type in `ctyp`.

2.2 Abstract Dynamic Semantics

We define the small-step semantics of our language using an abstract memory model that is reminiscent of the first abstract memory model defined in [21,22] for CompCert, enriched with some notations to increase readability of our presentation. Figure 3 summarizes the elements of this abstract memory model. The states of the memory are represented by an abstract data type *mem*. A memory state stores several *memory blocks*, each block being uniquely identified by a value in *block*. The empty memory is denoted by the constant *emp*. Pointer values, called locations, are represented by pairs (b, o) of block identifier b and a byte offset o inside the block. We denote by *loc* the set of such pairs and provide two operations to build them: $\mathit{base}(\mathtt{v})$ gives the location of a program variable `v`, and $\mathit{shift}(\ell, n)$ computes the location obtained by shifting the offset of location ℓ by n bytes. The shift operation abstracts pointer arithmetics. Memory blocks store values of type *val*, which may be integer or location values. The typing function $\mathtt{cty}(.)$ is extended to locations based on the typing of the program variable used as base of the

Types	mem,	$block$,
	$loc \triangleq$	$block \times \mathbb{N}$,
	$val \triangleq$	$Vint(\mathbb{Z}) \mid Vptr(loc)$
Constants	$emp:$	mem
Operations	$base:$	$\texttt{cvar} \to loc$
	$shift:$	$loc \to \mathbb{N} \to loc$
	$load:$	$mem \to \texttt{styp} \to loc \rightharpoonup val$
	$store:$	$mem \to \texttt{styp} \to loc \to val \rightharpoonup mem$

Figure 3: Abstract memory model

$$
\begin{array}{lll}
[\![\,\texttt{i}\,]\!](m) & \triangleq Vint(\texttt{i}) & \\
[\![\,*_{\texttt{u}}\texttt{a}\,]\!](m) & \triangleq load(m, \texttt{u}, [\![\,\texttt{a}\,]\!](m)) & \\
[\![\,\texttt{a}\,]\!](m) & \triangleq Vptr([\![\,\texttt{a}\,]\!](m)) & [\![\,*_{\texttt{u}}\texttt{a} = \texttt{e}\,]\!](m) \triangleq \mathsf{store}(m, \texttt{u}, [\![\,\texttt{a}\,]\!](m), [\![\,\texttt{e}\,]\!](m)) \\
[\![\,(\texttt{arith})\texttt{e}\,]\!](m) & \triangleq \texttt{convert}([\![\,\texttt{e}\,]\!](m), \texttt{arith}) & [\![\,\texttt{assert e}\,]\!](m) \triangleq \text{if } [\![\,\texttt{e}\,]\!](m) \not\sim 0 \text{ then } m \text{ else } \bot \\
[\![\,\texttt{op e}\,]\!](m) & \triangleq \widehat{\texttt{op}}([\![\,\texttt{e}\,]\!](m)) & \\
[\![\,\texttt{e op e}'\,]\!](m) & \triangleq \widehat{\texttt{op}}([\![\,\texttt{e}\,]\!](m), [\![\,\texttt{e}'\,]\!](m)) &
\end{array}
$$

$$
\begin{array}{ll}
[\![\,\&\texttt{v}\,]\!](m) & \triangleq base(\texttt{v}) \\
[\![\,\texttt{e}\,]\!](m) & \triangleq \ell \text{ if } [\![\,\texttt{e}\,]\!] = Vptr(\ell) \\
[\![\,\texttt{a.f}\,]\!](m) & \triangleq shift([\![\,\texttt{a}\,]\!](m), \texttt{offset(f)}) \\
[\![\,\texttt{a[e]}\,]\!](m) & \triangleq shift([\![\,\texttt{a}\,]\!](m), |\texttt{cty(a[0])}| \times \texttt{convert}([\![\,\texttt{e}\,]\!](m), \texttt{u32}))
\end{array}
$$

Figure 4: Generic semantics of expressions ($[\![\,\cdot\,]\!] : mem \to \texttt{expr} \rightharpoonup val$), addresses ($[\![\,\cdot\,]\!] : mem \to \texttt{addr} \rightharpoonup loc$), and statements ($[\![\,\cdot\,]\!] : mem \to \texttt{stmt} \rightharpoonup mem$); $\widehat{\texttt{op}}$ denotes type dependent operations, e.g., addition with pointer operand is done using *shift*.

location. The axiomatization of reading and storing operations is similar to the one in [21, 22]. We use partial functions for them in order to denote potential failures; we denote by $\bot$ the undefined value.

Figure 4 defines the semantics of expressions, addresses, and statements with respect to a memory state m, via the overloaded functions $[\![\,\cdot\,]\!]$. The semantic functions are partial: the undefined case $\bot$ cuts the evaluation.

2.3 Generating Verification Conditions

To fix ideas, we recall here the principle used to generate verification conditions (VC) and we apply it to the above toy language. This general principle is adapted in WP as explained in [3].

Verification conditions are generated from Hoare's triple $\{P\}\ s\ \{Q\}$ with P and Q formulas in some logic theory $\mathcal{L}$ used for program annotations and $\texttt{s}$ a program statement. For this, WP (and most deductive verification tools) employs the efficient weakest precondition computation method proposed in [16, 20]. It computes a formula $R(\vec{v}_b, \vec{v}_e)$ in $\mathcal{L}$ that specifies the relation between the states of the program before and after the execution of $\texttt{s}$, which are represented by the set of logic variables $\vec{v}_b$ (resp. $\vec{v}_e$). The VC built is $\forall \vec{v}_b, \vec{v}_e.\ \big(P(\vec{v}_b) \wedge R(\vec{v}_b, \vec{v}_e)\big) \implies Q(\vec{v}_e)$ and it is given to solvers for $\mathcal{L}$ to check its validity.

To compile a program statement $\texttt{s}$ into a formula $R(\cdot,\cdot)$, the tool uses the dynamic semantics of the language, given in Figure 4 for our toy language. The abstract memory model mem used in this semantics is represented by a *memory model environment* (called simply environment)

Types	mem,	loc
	$\mathsf{val} \triangleq$	$\mathsf{Vint}(\mathcal{E}_{\mathbb{I}}) \mid \mathsf{Vptr}(\mathsf{loc})$
Constants	$\mathsf{emp}:$	$2^{\mathtt{cvar}} \to \mathsf{mem}$
Operations	$\mathsf{base}:$	$\mathtt{cvar} \to \mathsf{loc}$
	$\mathsf{shift}:$	$\mathsf{loc} \to \mathcal{E}_{\mathbb{I}} \to \mathsf{loc}_\perp$
	$\mathsf{load}:$	$\mathsf{mem} \to \mathtt{styp} \to \mathsf{loc} \to \mathsf{val}_\perp$
	$\mathsf{store}:$	$\mathsf{mem} \to \mathtt{styp} \to \mathsf{loc} \to \mathsf{val} \to (\mathsf{mem} \times \mathcal{E}_{\mathbb{B}})_\perp$

Figure 5: Interface for memory model environments

that keeps the information required by the VC generation, for example the current set of variables used for modeling the state at the current point. Figure 5 defines a *signature for memory model environments* that captures the essential features required by the compilation of statements in our toy language into VC formulas. In the next sections, we supply several memory model environments that demonstrate the abstraction capabilities of this signature. In particular, we provide in Section 5 an environment that is parameterized by the abstract domain of Eva. Notice that, in WP, the VC generation follows a pass over the program syntax which collects, for each program statement, the safety conditions, (called alarms in the introduction), that eliminate the runs leading to undefined behaviors. For this reason, the VC generation does not collect such constraints in $R(\cdot,\cdot)$ and focuses only on the encoding of the semantics of statements.

The signature in Figure 5 should be compared with the abstract memory model from Figure 3. The environment shall define a type mem providing information about the state of the memory and a type loc encoding information on memory locations. By omitting memory blocks from this signature, we permit a more abstract relation between memory locations and program variables, based only on the operation base. This abstraction allows to define an efficient environment for programs without pointer manipulation (see Section 3). The environment shall provide a type for basic values stored in the memory: integers and locations. The integer values are specified by integer terms in $\mathcal{L}$, denoted by $\mathcal{E}_{\mathbb{I}}$. The empty environment for a set V of active program variables is provided by emp(V). The arithmetic operation on locations is encoded in operation shift, if this operation is supported by the memory model. Indeed, we use an error monad with error value $\perp$ for the result type of some operations to point out that these operations may be defined only under some conditions (e.g., no dereference, no alias, variable used only by reference, ...). We provide in Section 3.1 an example of environment where the operation shift is undefined. The updating of the memory environment store produces a new environment and a formula in $\mathcal{L}$, in the set $\mathcal{E}_{\mathbb{B}}$.[2]

The logic theory used to compile VCs, $\mathcal{L}$, shall satisfy the following constraints. It must be a multi-sorted first order logic that embeds the logic theory used to annotate programs in our toy language. (In Frama-C, the logic theory for annotations is defined using ACSL [5].) It includes the boolean theory (sort $\mathbb{B}$), the bit vector theory (sort $\mathbb{V}$) for bit operations, the integer arithmetic theory (sort $\mathbb{I}$), the array theory (with polymorphic type $array(\alpha,\beta)$ and classic operations for selection $a[k]$ and update $a[k \longleftarrow v]$), abstract data types (or at least polymorphic pairs with component selection by *fst* and *snd*), and uninterpreted functions. We suppose that $\mathcal{L}$ includes a conditional operator if-then-else for term building denoted by

[2] In languages with conditionals, environments should also provide a function $\mathsf{join} : \mathsf{mem} \to \mathsf{mem} \to \mathsf{mem} \times \mathcal{E}_{\mathbb{B}}$ which is used to join execution paths. It returns the new environment to use and a set of equalities which make the environments equal.

$ite(\cdot,\cdot,\cdot)$. We denote by $\mathcal{E}$ the set of logic terms built in $\mathcal{L}$ using the constants, operations, and variables in a set $\mathcal{X}$. We also suppose that an infinite number of fresh variable can be generated. For a logic sort τ, we denote by $\mathcal{E}_\tau$ the terms of type τ. The expressivity of $\mathcal{L}$ determines the precision of the VC generated, and the subset of the programming language constructs that may be dealt precisely by the VC generator. We illustrate this problem in the next section, where we present two memory model environments used by WP.

To ease the reading of environment definitions, we distinguish the logic terms by using the mathematical style and by underlining the terms of $\mathcal{L}$, e.g., $\underline{x+x}$. For example, the logic term $\mathsf{m[l]}\underline{+x}$ is built from a VC generator term $\mathsf{m[l]}$ that computes a logic term of integer type and the logic sub-term $\cdot + x$. For example, the VC generated for the Hoare's triple $\{P\}\ *_{\texttt{i8}}(\&\texttt{r.f}) = 5\ \{Q\}$ is $\underline{P \wedge e_1 \implies Q}$ where:

m_0	$\triangleq$	$\mathsf{emp}(\{\texttt{r}\})$
l_0	$\triangleq$	$\mathsf{shift}(\mathsf{base}(\texttt{r}), \texttt{offset(f)})$
m_1, e_1	$\triangleq$	$\mathsf{store}(\mathsf{m}_0, \texttt{i8}, \mathsf{l}_0, \mathsf{Vint}(5))$
$\underline{P}$		generated from P using environment m_0
$\underline{Q}$		generated from Q using environment m_1

3 Memory Models in WP

This section presents two memory model environments employed by WP for the generation of VCs. Notice that the logic theory $\mathcal{L}$ in which the VCs are encoded is the one provided by the Qed [12] module. Qed includes a programmatic interface to $\mathcal{L}$, several rewriters used to simplify the encoded formulas and translators to the input language of different solvers.

The memory model environments presented here could be combined to obtain an environment that provides the best memory model for each location. For simplicity, we present them separately in this section.

3.1 Simple Memory Model

$$\mathsf{mem} \triangleq \mathsf{array}\ (\texttt{cvar}, \mathcal{X}) \qquad \mathsf{loc} \triangleq \texttt{cvar}$$

$$\mathsf{emp}(\mathsf{V}) \triangleq \begin{cases} \bot & \text{if any v of V has pointer type} \\ [\mathtt{v}_1 \longleftarrow \alpha_1, \cdots, \mathtt{v}_n \longleftarrow \alpha_n] & \text{with } \{\mathtt{v}_1, \cdots, \mathtt{v}_n\} = \mathtt{V} \text{ and } \alpha_1, \cdots, \alpha_n \text{ fresh in } \mathcal{X} \end{cases}$$

$$\mathsf{base}(\mathtt{v}) \triangleq \mathtt{v} \qquad \mathsf{shift}(\mathsf{l}, \underline{e}) \triangleq \bot$$

$$\mathsf{load}(\mathsf{m}, \texttt{arith}, \mathsf{l}) \triangleq \mathsf{m[l]} \qquad \mathsf{load}(\mathsf{m}, \texttt{t ptr}, \mathsf{l}) \triangleq \bot$$

$$\mathsf{store}(\mathsf{m}, \mathtt{t}, \mathsf{l}, \texttt{Vint}\ (\underline{e})) \triangleq (\mathsf{m}[\mathsf{l} \longleftarrow \alpha], \underline{\alpha = e}) \quad \text{with } \alpha \in \mathcal{X} \text{ a fresh variable}$$

Figure 6: Simple memory model environment

The simplest memory model environment provided by WP is limited to programs of our toy language (and of C) that does not employ pointers. Only arithmetic type variables x are representable and dereferences are present only in left values $*(\&\texttt{x})$. For this reason, the environment type mem associates each program variable v to one logic variable from $\mathcal{X}$. The locations (in loc) are defined by the set of program variables cvar. The

```
2 int x = 10;
  int y = 11;
  /*@ assert x == 10; @*/
```

Figure 7: Simple program

implementation of the interface defined in Figure 5 is given in Figure 6. Notice that some operations are not implemented. The advantage of such a memory model is that read-over-writes are statically separated for the prover. Therefore, the program in Figure 7 is compiled into $\underline{x_0 = 10 \land y_0 = 11 \implies x_0 = 10}$ where x_0 and y_0 are logic variables (in $\mathcal{X}$) to which the memory model environment maps the variables `x` and `y`. The solver does not need to check that the assignment of `y` does not have any effect on the value of `x`.

3.2 Typed Memory Model

The next environment is restricted to the well-typed subset of the C programming language, which includes our toy language. The separation of memory regions is done per type, which is similar to "components as array" model of Burstall-Bornat. The definition of this environment is given in Figure 8. A location in this memory environment is a pair of integers (b_v, o) where b_v models the base of program variable `v` and o models the offset. Therefore, the environment type loc represents pairs of integer terms computing the base and the offset. The memory type mem maps each scalar type `u` in `styp` to an array logic variable $\alpha_{\mathtt{u}}$ representing the memory region storing values of type `u`. These arrays are indexed by a pair of integers representing locations, i.e., type of $\alpha_{\mathtt{u}}$ is $\mathit{array}(\mathbb{I} \times \mathbb{I}, \mathtt{u})$.

$$\mathsf{mem} \triangleq \mathsf{array}\ (\mathtt{styp}, \mathcal{X}) \qquad \mathsf{loc} \triangleq \mathbb{I} \times \mathcal{E}_{\mathbb{I}}$$

$$\mathsf{emp}(\mathtt{V}) \triangleq [\mathtt{u}_1 \longleftarrow \alpha_1, \cdots, \mathtt{u}_n \longleftarrow \alpha_n] \quad \begin{array}{l}\text{with } \{\mathtt{u}_1, \cdots, \mathtt{u}_n\} \text{ scalar types in } \mathtt{V} \\ \text{and } \alpha_1, \cdots, \alpha_n \text{ fresh array variables}\end{array}$$

$$\mathsf{base}(\mathtt{v}) \triangleq (\underline{b_{\mathtt{v}}, 0}) \qquad \text{with } b_{\mathtt{v}} \in \mathbb{I} \text{ fixed for } \mathtt{v}$$

$$\mathsf{shift}(\mathsf{l}, \underline{e_{\mathbb{I}}}) \triangleq (\underline{\mathit{fst}(\mathsf{l})}, \underline{\mathit{snd}(\mathsf{l}) + e_{\mathbb{I}}})$$

$$\mathsf{load}(\mathsf{m}, \mathtt{u}, \mathsf{l}) \triangleq \begin{cases} \mathsf{Vint}(\mathsf{m}[\mathtt{u}][\underline{\mathsf{l}}]) & \text{if } \mathtt{u} \in \mathtt{arith} \\ \mathsf{Vptr}(\mathsf{m}[\mathtt{u}][\underline{\mathsf{l}}]) & \text{otherwise} \end{cases}$$

$$\mathsf{store}(\mathsf{m}, \mathtt{u}, \mathsf{l}, \mathsf{Vint}(\underline{e})) \triangleq (\mathsf{m}[\mathtt{u} \longleftarrow \alpha], \underline{\alpha = (\mathsf{m}[\mathtt{u}])[\mathsf{l} \longleftarrow e]}) \quad \text{with } \alpha \in \mathcal{X} \text{ a fresh array variable}$$

$$\mathsf{store}(\mathsf{m}, \mathtt{u}, \mathsf{l}, \mathsf{Vptr}(\underline{e})) \triangleq (\mathsf{m}[\mathtt{u} \longleftarrow \alpha], \underline{\alpha = (\mathsf{m}[\mathtt{u}])[\mathsf{l} \longleftarrow e]}) \quad \text{with } \alpha \in \mathcal{X} \text{ a fresh array variable}$$

Figure 8: Typed memory model environment

Such a memory model is relevant for a larger class of programs, but the generated VCs are more complex. For example, the formula obtained in this memory model for the program in Figure 7 is $\underline{\big(\alpha_1 = \alpha_0[(b_{\mathtt{x}}, 0) \longleftarrow 10] \land \alpha_2 = \alpha_1[(b_{\mathtt{y}}, 0) \longleftarrow 11]\big) \implies \alpha_2[(b_{\mathtt{x}}, 0)] = 10}$ where $\alpha_0, \alpha_1, \alpha_2 \in \mathcal{X}$ are logic arrays variables and $b_{\mathtt{x}}$ and $b_{\mathtt{y}}$ are distinct constants fixed for program variables `x` and `y`.

4 Value Analysis in Eva

The Eva plug-in implements a static *value analysis* based on the principles of abstract interpretation [14]. Abstract interpretation links a *concrete* semantics, typically the set of all possible executions of a program, to a more coarse-grained, *abstract* semantics. Any program transformation must have an abstract encoding that captures all possible outcomes defined by the concrete semantics. This ensures that the abstract semantics is a sound approximation of the runtime behavior of the program. For each instruction of the program, the information inferred by Eva includes two components:

1. the status True/False/Unknown of each alarm generated for the instruction (recall that these alarms prevent any undefined behavior from taking place) and
2. for each memory location involved in the instruction, an over-approximation of the values it may contain.

In this work, both kinds of information above are useful, for different reasons. Firstly, the alarms already proved (status True or False) by Eva are removed from the set of properties to be proven by WP. Secondly, we use the over-approximation inferred for pointer values (which is equivalent to a points-to information) to build the memory model environment (mainly the set of memory blocks) used by WP uses to build VCs (Section 5).

4.1 Abstractions for values

memory abstraction	$\mathtt{mem}^\# \ni \mathtt{m}^\#$
extended integers	$\mathtt{i}^\#_\infty ::= i \mid -\infty \mid +\infty$
abstract integers	$\mathtt{I}^\# \ni \mathtt{i}^\# ::= \{i, \dots\} \mid [\mathtt{i}^\#_\infty .. \mathtt{i}^\#_\infty], n\%n$
abstract bases	$\mathtt{b}^\# ::= \mathtt{v} \mid \mathbf{0}$
abstract locations	$\mathtt{loc}^\# \ni \mathtt{l}^\# ::= \{(\mathtt{b}^\#, \mathtt{i}^\#), \dots\}$
abstract values	$\mathtt{val}^\# \ni \mathtt{v}^\# ::= \mathtt{l}^\#$

Figure 9: Abstractions used for memory and scalars in Eva

The abstractions used by Eva to represent integers and pointers are given in Figure 9 and detailed below. We omit the details on the abstraction used for memory states (domain $\mathsf{mem}^\#$) which are highly technical and not relevant for this work.

Abstract integers values: If the over-approximation computed for an integer value is a small set of constant values, the abstract value used is exactly this set. If the computed set is large (i.e., exceeds some threshold), it is represented as a potentially unbounded interval with congruence information. For instance, $x \in [3..255], 3\%4$ means that x is such that $3 \leq x \leq 255 \wedge x \equiv 3 \bmod 4$. The congruence information is very important to precisely encode alignment constraints for pointer arithmetics.

Abstract location values: Eva assumes (and verifies) that the program performs no invalid (e.g., out-of-bounds) array/pointer accesses. For this, pointers are seen as offsets with respect to a symbolic block location called *base* $\mathtt{b}^\#$ and have no relation with the locations in the virtual memory space used during the concrete execution. Blocks of different bases are implicitly separated: it is impossible to move from the block of one base to another using pointer arithmetic. Base locations can be (1) the location assigned to a local or global variable, (2) the location of the formal parameter of a function and (3) the **0** location, that stands for the NULL pointer. Offsets are plain integers and abstracted by values in $\mathtt{I}^\#$.

The abstraction of pointers as a set of pairs built from a base location and an offset is slightly more precise than the traditional abstraction in which a pointer is mapped to the pair of (i) the set of possible bases and (ii) the set of possible offsets for all bases. Hence, Eva can represent precisely a pointer that could be either NULL, equal to the address of a variable `x`, or equal to the addresses of the cells 3 to 10 of an array `T` of 16 bits integers. Expressed as an element of $\mathtt{loc}^\#$, and assuming that offsets represent a number of bytes, this set of possible values for a pointer is abstracted by $\{(\mathbf{0}, \{0\}); (\mathtt{x}, \{0\}); (\mathtt{T}, ([4..18], 0\%2))\}$.

Abstract values: Abstract values $\mathtt{v}^\#$ are exactly abstract locations. The abstraction of integer variables is given by the offset part of an abstract location with base **0**.

4.2 Abstract operations

$$\llbracket \mathtt{i} \rrbracket^\#(\mathsf{m}^\#) \triangleq \{(\mathbf{0}, \{i\})\} \qquad \llbracket \mathtt{op}(\mathtt{e}_1, \mathtt{e}_2) \rrbracket^\#(\mathsf{m}^\#) \triangleq \widehat{op}^\#(\llbracket \mathtt{e}_1 \rrbracket^\#(\mathsf{m}^\#), \llbracket \mathtt{e}_2 \rrbracket^\#(\mathsf{m}^\#))$$
$$\llbracket \&\mathtt{v} \rrbracket^\#(\mathsf{m}^\#) \triangleq \{(\mathtt{v}, \{0\})\} \qquad \mathsf{shift}^\#(\{(b_k^\#, i_k^\#), \dots\}, i^\#) \triangleq \{(b_k^\#, i_k^\# +^\# i^\#), \dots\}$$

$$\begin{aligned} \mathsf{ct}^\# &: \mathsf{mem}^\# \to \mathtt{styp} \to \mathtt{cvar} \times \mathbb{N} \to \mathsf{val}^\# \\ \mathtt{valid}^\# &: \mathsf{loc}^\# \to \mathbb{N} \to \mathsf{loc}^\# \\ \mathsf{load}^\#(\mathsf{m}^\#, \mathtt{u}, \mathtt{l}^\#) &\triangleq \sqcup^\#_{l \in \gamma(\mathtt{l}^\#)} \mathsf{ct}^\#(\mathsf{m}^\#, \mathtt{u}, l) \\ \llbracket *_{\mathtt{u}}\mathtt{a} \rrbracket^\#(\mathsf{m}^\#) &\triangleq \mathsf{load}^\#(\mathsf{m}^\#, \mathtt{u}, \mathtt{valid}^\#(\llbracket \mathtt{a} \rrbracket^\#(\mathsf{m}^\#), |\mathtt{u}|)) \end{aligned}$$

Figure 10: Selected abstract operations

We give in Figure 10 the definition of most interesting abstract operations for the operations of our language. Integers and addresses are injected into the abstractions described above in the obvious way. Binary operators are handled using corresponding abstract operators $\widehat{\mathtt{op}}^\#$. We omit arithmetic casts, that are handled by the integer abstraction. An abstract shift operation $\mathsf{shift}^\#$ is used to shift pointers by an integer: it preserves the base and shifts the offset. The semantics for $\llbracket \mathtt{a.f} \rrbracket^\#$ and $\llbracket \mathtt{a[e]} \rrbracket^\#$ are the same as in Figure 4, except they use $\mathsf{shift}^\#$.

We omit the full definition of abstract semantics for memory reads and updates, because of the complexity of the memory abstraction. We only sketch the semantics for a memory access $\llbracket *_{\mathtt{u}}\mathtt{a} \rrbracket^\#_{\mathsf{m}}$, based on three functions. The first function, $\mathsf{ct}^\#$, has as input an abstract memory $\mathsf{m}^\#$, a scalar type $\mathtt{u}$, and a concrete location (b, o). It reads the content of $\mathsf{m}^\#$ for the base address b, at offset o, over $|\mathtt{u}|$ bytes, and returns it as an abstract value. The second function, $\mathsf{load}^\#(\mathsf{m}^\#, \mathtt{u}, \mathtt{l}^\#)$, joins (using the abstract join $\sqcup^\#$) the abstract contents computed by $\mathsf{ct}^\#$ at the concrete locations abstracted by $\mathtt{l}^\#$. This operation requires that $\gamma(\mathtt{l}^\#)$, the concretization of $\mathtt{l}^\#$, is a set of finitely many locations. The last function, $\mathtt{valid}^\#$, takes as argument an abstract location $\mathtt{l}^\#$ and a size s. It selects from $\mathtt{l}^\#$ the pairs $(b^\#, i^\#)$ that allow a *valid* access of s bytes. In particular, this function trims the pairs of $\mathtt{l}^\#$ that would lead to out-of-bound accesses, including those on the base $\mathbf{0}$. (These locations are ruled out by the alarms that guarantee that the instruction executes without an undefined behavior.) Since our abstract bases have a known C type, the number of concrete locations in $\mathtt{valid}^\#(\mathtt{l}^\#, |\mathtt{u}|)$ is always finite. Hence, $\llbracket *_{\mathtt{u}}\mathtt{a} \rrbracket^\#_{\mathsf{m}}$ can be written in terms of $\mathsf{load}^\#$ operations.

4.3 Handling function calls

Eva performs whole-program analyses. Function calls are treated by symbolic inlining. The arguments of the functions are evaluated, assigned to the formal parameters of the callee, and the body of the callee is analyzed. (Recursion is not supported.) In particular, this means that the address of a local variable of the caller can be used transparently in the callee, as shown in Figure 11. The abstraction for `p` in `f` is $\{(\mathtt{x}, \{0\})\}$, where `x` is a variable which is *not* syntactically in scope in `f`. This is the main difficulty in translating the results of Eva to another, modular verification tool, e.g. WP, and the reason why we need a custom memory model.

```
void f(int *p) {
  *p = 1;
}
void main() {
  int x;
  f(&x);
}
```

Figure 11: Pointers to local variables

5 Deriving a Memory Model from an Abstract Domain

In this section, we introduce our memory model environment based on a value analysis performed by a static analyzer. The set of disjoint memory blocks building the memory is derived from the bounded set of memory blocks computed by this analysis. We first provide a signature defining what is required from the static analyzer, then define our generic environment on top of this signature, and finally instantiate the signature with Eva.

5.1 Signature required from the static analyzer

$$
\begin{array}{l}
\textbf{sig}\ \mathsf{Domain}\ : \\
\quad \textbf{type}\ \mathcal{V} \qquad\qquad \textbf{type}\ \mathcal{S} \\
\quad \mathsf{base} : \mathtt{cvar} \to \mathcal{V} \qquad \mathsf{load} : \mathcal{S} \to \mathtt{styp} \to \mathcal{V} \to \mathcal{V} \\
\quad \mathsf{shift} : \mathcal{V} \to \mathcal{E}_{\mathbb{I}} \to \mathcal{V} \\
\quad \mathsf{domain} : \mathcal{V} \to 2^{\mathtt{cvar}}
\end{array}
$$

Figure 12: Domain signature

Figure 12 defines the signature that any static analyzer must implement in order to build our memory model environment on top of its results. The analyzer shall provide a sort $\mathcal{V}$ for the abstraction of values and another for its abstraction of memory states $\mathcal{S}$. In addition, it shall provide basic operations over those sorts, including shifting on abstract values (function shift), querying the domain of a value (i.e., the blocks to which it refers given by function domain) and reading the abstract value at a given abstract location in some abstract memory state (function load). In order to ensure the soundness of our memory model, the static analyzer shall provide sound approximations for the functions in the above signature. We suppose for the rest of this section that this assumption is true.

5.2 Generic Model

Given a domain D satisfying the signature Domain, we define a memory model environment in Figure 5. The separation of memory is statically encoded by the finite number of disjoints blocks associated to bases. As in the typed model in Section 3.2, a pointer value is represented by a pair (b, o) where b is the representation of a base into our logic theory (encoded as an integer), and o is an offset in *bits* from this base. Locations are then encoded as a pair of a pointer of our model, associated with its abstract location $\mathtt{l}^{\#} \in \mathtt{loc}^{\#}$ counterpart. A possible solution is to encode the memory into a single array indexed by base identifiers. This solution is improved using the results provided by the static analysis, i.e., the finite set of bases a pointer may points-to. Then, we do a static separation of the memory in a finite set of arrays, one by base. Values stored at each base (block) are represented by an array $\mathit{array}(\mathbb{I}, \underline{\mathtt{t}})$ with $\underline{\mathtt{t}}$, as defined in Section 3.2.

$$
\begin{array}{ll}
\mathsf{mem} \triangleq \mathsf{array}(\mathtt{cvar}, \mathcal{X}) & \\
\mathsf{loc} \triangleq \mathcal{E}_{\mathbb{I}\times\mathbb{I}} \times \mathtt{loc}^{\#} & \\
\mathsf{emp}(\mathtt{V}) \triangleq [\mathtt{v}_1 \longleftarrow \alpha_1, \cdots, \mathtt{v}_n \longleftarrow \alpha_n] & \text{with } \{\mathtt{v}_1, \cdots, \mathtt{v}_n\} \text{ the bases computed by SA tool} \\
 & \text{and } \alpha_1, \cdots, \alpha_n \text{ fresh array variables in } \mathcal{X} \\
\mathsf{base}(\mathtt{v}) = (\underline{(b_{\mathtt{v}}, 0)},\ \mathsf{D.base}(\mathtt{v})) & \\
\mathsf{shift}((\underline{(b, o)}, \mathtt{l}^{\#}), e_{\mathbb{I}}) = (\underline{(b, o + e_{\mathbb{I}})},\ \mathsf{D.shift}(\mathtt{l}^{\#}, e_{\mathbb{I}})) &
\end{array}
$$

Our generic model could be designed to use the different over-approximations of the abstract state at *each* statement. For the simplicity of the presentation, we consider that the analysis using the domain D is terminated and we use the over-approximation $s^\#$ of the abstract state *for all* the statements simultaneously.

Reading a value from an abstract memory state m is obtained by statically dispatching the actual base value among the possible bases obtained from the domain.

$$\mathsf{load}(\mathsf{m}, \mathtt{t}, (\underline{(b,o)}, \mathtt{l}^\#)) = \begin{cases} \mathtt{Vint}\,(e) & \textit{if } \mathtt{t} \in \mathtt{arith} \\ \mathtt{Vptr}\,\big(e, \mathsf{D.load}(s^\#, \mathtt{t}, \mathtt{l}^\#)\big) & \textit{otherwise} \end{cases}$$
$$\textit{where} \begin{cases} e \triangleq \underline{\mathit{ite}(b = b_{\mathtt{v}_1}, \mathsf{m}[\mathtt{v}_1][o], \cdots, \mathit{ite}(b = b_{\mathtt{v}_{n-1}}, \mathsf{m}[\mathtt{v}_{n-1}][o], \mathsf{m}[\mathtt{v}_n][o]) \cdots)} \\ \text{when } \mathsf{D.domain}(\mathtt{l}^\#) = \{\mathtt{v}_1, \cdots, \mathtt{v}_n\} \end{cases}$$

The store operation is implemented with a parallel conditional write on each chunk that may be pointed-to. The generated formula is linear on the size of the set of chunks given by the domain.

$$\mathsf{store}(\mathsf{m}, \mathtt{t}, (\underline{(b,o)}, \mathtt{l}^\#), v) = \mathsf{m}', \underline{e'}$$
$$\textit{where} \begin{cases} v = \mathsf{Vint}(\underline{e'}) \text{ or } v = \mathsf{Vptr}((\underline{e'}, \mathtt{l}_v^\#)) \\ \mathsf{m}' = \mathsf{m}[\mathtt{v}_1 \longleftarrow \alpha_1, \cdots, \mathtt{v}_n \longleftarrow \alpha_n] \\ e' \triangleq \underline{\bigwedge_{i=1}^n \mathit{ite}(b = b_{\mathtt{v}_i}, \alpha_i = \mathsf{m}[\mathtt{v}_i][\underline{o \longleftarrow e'}, \alpha_i = \mathsf{m}[\mathtt{v}_i]\underline{]}} \\ \text{when } \mathsf{D.domain}(\mathtt{l}^\#) = \{\mathtt{v}_1, \cdots, \mathtt{v}_n\} \end{cases}$$

This use of *ite* in load and store is reminiscent of the encoding with arrays. Indeed, if the points-to analysis always returns for D.domain the set of all variables, the VC generated does not win any concision. On the other hand, if the points-to analysis always returns a singleton for calls to D.domain, the model produces a formula as simple as the one obtained using the simple memory model in Section 3.1, but for a larger class of programs that may use pointers.

5.3 An instance on top of Eva

We now show how to build an instance of the signature Domain of Section 5.1 using the value analysis provided by Eva. The functions required in the signature are given below. Values are simply abstract locations, making most operations immediate. When computing domain, we skip the **0** base, because this corresponds to out-of-bounds accesses, implicitly forbidden by the alarms generated for the instruction.

$$\begin{aligned} \textbf{module}\ \mathsf{Eva} : \mathsf{Domain}\ = \\ \textbf{type}\ \mathcal{V} &\triangleq \mathtt{loc}^\# \\ \mathsf{base}(\mathtt{v}) &\triangleq \{(\mathtt{v}, \{0\})\} \\ \mathsf{shift}(\mathtt{l}^\#, e_{\mathbb{I}}) &\triangleq \begin{cases} \mathsf{shift}^\#(\mathtt{l}^\#, \{k\}) & \text{if } e_{\mathbb{I}} \text{ is a constant } k \\ \mathsf{shift}^\#(\mathtt{l}^\#, ([-\infty..+\infty], 0\%1)) & \text{otherwise} \end{cases} \\ \mathsf{domain}(\{(b_k^\#, o_k^\#), \dots\}) &\triangleq \{b_k^\#, \dots\} \setminus \{\mathbf{0}\} \\ \textbf{type}\ \mathcal{S} &\triangleq \mathtt{mem}^\# \\ \mathsf{load}(\mathtt{m}^\#, \mathtt{t}, \mathtt{l}^\#) &\triangleq \mathsf{load}^\#(\mathtt{m}^\#, \mathtt{t}, \mathtt{l}^\#) \end{aligned}$$

The over-approximations computed by Eva on values of expressions can also be used as additional constraints to obtain more precise VCs, especially for integer variables or offsets. However, this is not required by the signature of Figure 12, and is thus optional.

6 Experiments

We have implemented in OCaml the proposed memory model. The implementation is available as a plug-in of Frama-C. It includes 560 LoC for the module that encodes the memory model and implements the interface (signature) required by the WP for the memory models. In addition, the plug-in includes 65 LoC for the glue between Eva and WP, which calls Eva plug-in, interprets its results using the memory model proposed, and launches the WP plug-in. The plug-in will be released within the Frama-C platform once it will be more mature.

The plug-in presents additional features. For example, it transfers more informations from Eva than the sound partitioning of the memory. Indeed, the static analysis of Eva returns also the size of each element of the partition computed and an over-approximation of the set of values for each variable, at each program point. Our plug-in transfers this information about the contents of variables to WP, as additional assertions.

The interface provided by WP allows us to compare our memory model with existing memory models. Unfortunately, most of examples in the test benchmark of WP use only features supported by the existing memory models (i.e., the ones presented in Section 3), and the points-to analysis does not improve the quality of VC generated. We are currently testing our plug-in on the benchmark of the SV-COMP competition. One challenge with this benchmark is to obtain code annotated for WP, because many interesting examples involve loops, for which a loop invariant is often required for WP. We are currently experimenting with exporting the results of Eva for the contents of variables as invariants in order to avoid manual annotations.

An example that is present in the benchmark of WP and which motivated this work is the one presented in the introduction (see Section 1). In this example, the default memory model of WP (see Section 3.2) cannot prove the post-condition of function `copy`. Since separation is done per type, the model is not able to specify that parameters `a` and `b`, which have the same type, cannot alias. Therefore, the following clause

$$(a + \mathsf{sizeof}(a) \leq b) \vee (b + \mathsf{sizeof}(b) \leq a)$$

is added to the generated VC, unless specified as a pre-condition of the function. The invariants inferred by the analysis with Eva for this example specify that `a` (resp. `b`) points-to a block of memory indexed by variable `t` (resp. by `u`). From the declaration in `main`, Eva infers that those blocks are disjoint. Our plug-in transfers this information to WP through our memory model, and this is sufficient to automatically prove the post-condition.

7 Conclusion

To summarize our contributions, we have defined a framework that allows to transfer the knowledge inferred by a static analyzer on the program memory regions to a deductive verification tool that supports generation of verification conditions parameterized by a memory model. One of the challenging point was to find a simple formalization of this framework, which is independent of the specificities of tools employed. We implemented this framework in Frama-C, and reported on preliminary experimental results.

We hope to be able to report on additional benchmark for which a collaboration of WP with Eva is fruitful in the near future. We also envision various directions as potential future works.

First, one of the current limitations of WP lies in its handling of pointer casts and unions (when used for type punning): most programs are statically rejected as impossible to be encoded precisely within the typed memory model. But the low-level memory model of Eva gives us precise memory information, which could be used to determine whether the casts endanger the

soundness of using the typed memory model. Alternatively, we could define an entirely new model for blocks whose contents are used in type-unsafe way.

Second, another interesting extension would be to support dynamic allocation and deallocation. Currently, the model used by Eva for calls to `malloc` and `free` uses a weak update semantics, while WP does not handle these functions. It is likely that some form of support on the WP side, especially on the interface of memory models, will be required.

Third, we mentioned in Section 6 that we export more than the memory partitioning information to WP. However, the information which is transferred remains limited to simple cases. We believe it is possible to improve this exchange of information, typically for arrays that may be represented very concisely by Eva.

Finally, we would also like to extend the information exchanged with WP w.r.t. pointers. Currently, we rely on the fact that pointers pointing to different blocks are separated. However, it would also be interesting to infer more fine-grained separation. Typically, two pointers pointing to the same block, but in separate parts of this block. This would permit proving that it is safe to call `copy(&t[0],&t[2]);` in the example of Figure 1. This information could be inferred from the abstract offsets of the pointer values. Another interesting information to exchange is the validity of pointers, so that the precondition of `copy` also becomes superfluous. This should be quite straightforward, due to the validity information inferred by Eva and used to check that pointer accesses are in bounds.

References

[1] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Marion. Sound and quasi-complete detection of infeasible test requirements. In *ICST'15*, pages 1–10. IEEE Computer Society, 2015.

[2] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of FMCO*, number 4111 in LNCS, pages 364–387. Springer, 2005.

[3] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of PASTE'05*, pages 82–87. ACM, 2005.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proceedings of CASSIS*, LNCS, pages 49–69. Springer, 2004.

[5] P. Baudin, J. C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008.

[6] S. Böhme and M. Moskal. Heaps and data structures: A challenge for automated provers. In *Proceedigns of CADE-23*, volume 6803 of *LNCS*, pages 177–191. Springer, 2011.

[7] D. Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions*. PhD thesis, University of Rennes, 2017.

[8] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.

[9] B.-Y. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *Proceedings of SAS*, volume 4634 of *LNCS*, pages 384–401. Springer, 2007.

[10] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A low-level memory model and an accompanying reachability predicate. *STTT*, 11(2):105–116, February 2009.

[11] W. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. Comput. Program.*, 77(9):1006–1036, 2012.

[12] L. Correnson. Qed. computing what remains to be proved. In *Proceedings of NFM*, volume 8430 of *LNCS*, pages 215–229. Springer, 2014.

[13] L. Correnson and F. Bobot. Exploring memory models with Frama-C/WP, 2017. Personal comunication.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[15] J. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *Proceedings of CAV*, volume 4590 of *LNCS*, pages 173–177. Springer, 2007.

[16] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. *SIGPLAN Not.*, 36(3):193–205, Jan. 2001.

[17] T. Hubert and C. Marché. Separation analysis for deductive verification. In *Proceedings of HAV*, pages 81–93, Braga, Portugal, Mar. 2007.

[18] B. Jacobs and F. Piessens. The VeriFast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, 2008.

[19] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015.

[20] K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.

[21] X. Leroy, A. W. Appel, S. Blazy, and G. Stewart. The CompCert Memory Model, Version 2. Research Report RR-7987, INRIA, June 2012.

[22] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *J. Autom. Reasoning*, 41(1):1–31, 2008.

[23] F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In *Proceedings of CADE-19*, volume 2741 of *LNCS*, pages 121–135. Springer, 2003.

[24] P. W. O'Hearn, J. C. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of CSL*, volume 2142 of *LNCS*, pages 1–19. Springer, 2001.

[25] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic with trees and data. In *Proceedings of CAV*, pages 711–728. Springer, 2014.

[26] Z. Rakamaric and A. J. Hu. A scalable memory model for low-level code. In *Proceedings of VMCAI*, volume 5403 of *LNCS*, pages 290–304. Springer, 2009.

[27] P. Sotin, B. Jeannet, and X. Rival. Concrete memory models for shape analysis. *Electr. Notes Theor. Comput. Sci.*, 267(1):139–150, 2010.

[28] W. Wang, C. Barrett, and T. Wies. Partitioned memory models for program analysis. In *Proceedings of VMCAI*, volume 10145 of *LNCS*, pages 539–558. Springer, 2017.

agdarsec – Total Parser Combinators

Guillaume Allais[1]

Radboud University, Nijmegen, The Netherlands
guillaume.allais@ens-lyon.org

Abstract

Parser combinator libraries represent parsers as functions and, using higher-order functions, define a DSL of combinators allowing users to quickly put together programs capable of handling complex recursive grammars. When moving to total functional languages such as Agda, these programs cannot be directly ported: there is nothing in the original definitions guaranteeing termination.

In this paper, we will introduce a 'guarded' modal operator acting on types and show how it allows us to give more precise types to existing combinators thus guaranteeing totality. The resulting library is available online together with various usage examples at https://github.com/gallais/agdarsec.

1 Introduction

Parser combinators have made functional languages such as Haskell shine. They are a prime example of the advantages Embedded Domain Specific Languages [8] provide the end user. She not only has access to a set of powerful and composable abstractions but she is also able to rely on the host language's existing tooling and libraries. She can get feedback from the static analyses built in the compiler (e.g. type and coverage checking) and can exploit the expressivity of the host language to write generic parsers thanks to polymorphism and higher order functions.

However she only gets the guarantees the host language is willing to give. In non-total programming languages such as Haskell this means she will not be prevented from writing parsers which will unexpectedly fail on some (or even all!) inputs. Handling a left-recursive grammar is perhaps the most iconic pitfall leading beginners to their doom: a parser never making any progress. Other issues one may want guarantees about range from unambiguity to complexity with respect to the input's size.

We start with a primer on parser combinators and follow up with the definition of a broken parser which is silently accepted by Haskell. We then move on to Agda [16] and introduce combinators to define functions by well-founded recursion. This allows us to define a more informative notion of parser and give more precise types to the combinators commonly used. We then demonstrate that broken parsers such as the one presented earlier are rejected whilst typical example can be ported with minimal modifications.

Remark: Agda-centric Although we do use some Agda-specific techniques in order to have a codebase as idiomatic as possible, we do not expect the reader to be well-versed in them. We insert remarks similar to this one throughout the paper to clarify confusing points, and give pointers to more in-depth explanations to the interested reader.

This work is however not limited to Agda: it can be ported to other dependently-typed languages and we have already done so for Coq [13] (https://github.com/gallais/parseque) and Idris [3] (https://github.com/gallais/idris-tparsec).

2 A Primer on Parser Combinators

2.1 Parser Type

Let us start by reminding ourselves what a parser is. Although we will eventually move to a more generic type, Fritz Ruehr's rhyme gives us the *essence* of parsers:

> A Parser for Things
> is a function from Strings
> to Lists of Pairs
> of Things and Strings!

This stanza translates to the following Haskell type. We use a **newtype** wrapper to have cleaner error messages:

```
newtype Parser a = Parser { runParser
  :: String           -- input string
  → [(String, a)] }   -- pairs of leftovers and values
```

It is naturally possible to run such a parser and try to extract a value from a valid run. Opinions may vary on the exact definition of a successful parse: should a run with leftover characters or an ambiguous result be accepted? We decide against both in the following code snippet but it is not a crucial point.

```
parse :: Parser a → String → Maybe a
parse p s = case filter (null ∘ fst) (runParser p s) of
  [(_, a)] → Just a
  _        → Nothing
```

Once we are equipped with this type of parsers and a function to run them, we can start providing some examples of parsers and parser combinators.

2.2 (Strongly-Typed) Combinators

The most basic parser is the one that accepts any character. It succeeds as long as the input string is non empty and returns one result: the tail of the string together with the character it just read.

```
anyChar :: Parser Char
anyChar = Parser $ λs → case s of
  []     → []
  (c : s) → [(s, c)]
```

However what makes parsers interesting is that they recognize structure. As such, they need to reject invalid inputs. The parser only accepting decimal digits is a bare bones example. It can be implemented in terms of *guard*, a higher order parser checking that the value returned by its argument abides by a predicate which can easily be implemented using functions from the standard library.

```
guard :: (a → Bool) → Parser a → Parser a
guard f p = Parser $ filter (f ∘ snd) ∘ runParser p
```

```
digit :: Parser Char
digit = guard (∈ "0123456789") anyChar
```

These two definitions are only somewhat satisfactory: the result of the *digit* parser is still *stringly-typed*. Instead of using a predicate to decide whether to keep the value, we can opt for a validation function of type $a \to Maybe\ b$ which returns a witness whenever the check succeeds. To define this refined version of *guard* called *guardM* we can again rely on the standard library:

```
guardM :: (a → Maybe b) → Parser a → Parser b
guardM f p = Parser $ catMaybes ∘ fmap (traverse f) ∘ runParser p
```

- *traverse f* of type *(String, a) → Maybe (String, b)* takes apart a pair, applies f to the second component and rebuilds the pair *under* the *Maybe* type constructor,
- *fmap* applies this function to all the elements in the list obtained by running the parser p,
- and *catMaybes* of type *[Maybe (String, b)] → [(String, b)]* only retains the values which successfully passed the test.

In our concrete example of recognizing a digit, we want to return the corresponding `Int`. Once more the standard library has just the right function to use together with *guardM*: *readMaybe* of (specialised) type *String → Maybe Int*.

```
digit :: Parser Int
digit = guardM (readMaybe ∘ (:[])) anyChar
```

2.3 Expressivity: Structures, Higher Order Parsers and Fixpoints

We have seen how we can already rely on the standard library of the host language to seamlessly implement combinators. We can leverage even more of the existing codebase by noticing that the type constructor `Parser` is a `Functor`, an `Applicative` [14], a `Monad` and also an `Alternative`.

Functor means that given a function of the right type, we can alter the values returned by a parser. That is, we have a function (which corresponds to the infix combinators (<$>)):

```
fmap :: (a → b) → Parser a → Parser b
```

Applicative means two things. First, that given a value of type `a`, we can define a parser for values of type `a`. Second, that given a parser for a function and a parser for its argument we can run both and apply the function to its argument. That is, we have two functions:

```
pure  :: a → Parser a
(<*>) :: Parser (a → b) → Parser a → Parser b
```

Monad means that we are entitled to inspect the result of a first parser to decide which one to run next. This brings us beyond the realm of context-free grammars. That translates into the existence of one function:

```
(>>=) :: Parser a → (a → Parser b) → Parser b
```

Alternative means that for all type `a`, `Parser a` forms a monoid. It allows us to take the disjunction of various parsers, the failure of one leading to the next being used.

$$empty :: Parser\ a$$
$$(<|>) :: Parser\ a \to Parser\ a \to Parser\ a$$

Our first example of a higher order parser was *guard* which takes as arguments a validation function as well as another parser and produces a parser for the type of witnesses returned by the validation function.

The two parsers *some* and *many* turn a parser for elements into ones for non-empty and potentially empty lists of such elements respectively. They concisely showcase the power of mutual recursion, higher-order functions and the `Functor`, `Applicative`, and `Alternative` structure.

$$some :: Parser\ a \to Parser\ [a] \qquad many :: Parser\ a \to Parser\ [a]$$
$$some\ p = (:) <\$> p <*> many\ p \qquad many\ p = some\ p <|> pure\ [\,]$$

Remark: Non-Commutative The disjunction combinator is non-commutative as ultimately we obtain a list (and not a set) of possible results. As such the definitions of *some* and *many* will try to produce the longest list possible as opposed to a flipped version of many which would start by returning the empty list and slowly offer longer and longer matches.

3 The Issue with Haskell's Parser Types

The ability to parse recursive grammars by simply declaring them in a recursive manner is however dangerous: unlike type errors which are caught by the typechecker and partial covers in pattern matchings which are detected by the coverage checker, termination is not guaranteed.

The problem already shows up in the definition of *some* which will only make progress if its argument actually uses up part of the input string. Otherwise it may loop. However this is not the typical hurdle programmers stumble upon: demanding a non empty list of nothing at all is after all rather silly. The issue manifests itself naturally whenever defining a left recursive grammar which leads us to introducing the prototypical such example: `Expr`, a minimal language of arithmetic expressions.

$$\texttt{Expr} ::= \texttt{<Int>} \mid \texttt{<Expr>}\ \text{`+`}\ \texttt{<Expr>}$$

The intuitive solution is to simply reproduce this definition by introducing an inductive type for `Expr` and then defining the parser as an alternative between a literal on one hand and a sub-expression, the character '+', and another sub-expression on the other.

$$\textbf{data}\ Expr = Lit\ Int \mid Add\ Expr\ Expr$$

$$expr :: Parser\ Expr$$
$$expr = Lit <\$> int <|> Add <\$> expr <* char\ \texttt{'+'} <*> expr$$

However this leads to an infinite loop. Indeed, the second alternative performs a recursive call to *expr* even though it hasn't consumed any character from the input string.

The typical solution to this problem is to introduce two 'tiers' of expressions: the *base* ones which can only be whole expressions if we consume an opening parenthesis first and the *expr* ones which are left-associated chains of *base* expressions connected by '+'.

```
base :: Parser Expr
base = Lit <$> int <|> char '(' *> expr' <* char ')'

expr :: Parser Expr
expr = base <|> Add <$> base <* char '+' <*> expr'
```

This presentation is still sub-optimal; users would traditionally be encouraged to used combinators such as *chainl* to avoid this issue. We will only discuss later in Section 5.3.

This approach can be generalised when defining more complex languages by having even more tiers, one for each *precedence level*, see for instance Section 6. An extended language of arithmetic expressions would for instance distinguish the level at which addition and subtraction live from the one at which multiplication and division do.

Our issue with this solution is twofold. First, although we did eventually manage to build a parser that worked as expected, the compiler was unable to warn us and guide us towards this correct solution. Additionally, the blatant partiality of some of these definitions means that these combinators and these types are wholly unsuitable in a total setting. We could, of course use an escape hatch and implement our parsers in Haskell but that would both be unsafe and mean we would not be able to run them at typechecking time which we may want to do if we embed checked examples in our software's documentation, or use compilation-time configuration via e.g. dependent type providers [5].

4 Indexed Sets and Course-of-Values Recursion

Our implementation of Total Parser Combinators is in Agda, a total dependently typed programming language and it will rely heavily on indexed sets. But the indices will not be playing any interesting role apart from enforcing totality. As a consequence, we introduce combinators to build indexed sets without having to mention the index explicitly. This ought to make the types more readable by focusing on the important components and hiding away the artefacts of the encoding.

The first kind of combinators corresponds to operations on sets which are lifted to indexed sets by silently propagating the index. We only show the ones we will use in this paper: the pointwise arrow and product types and the constant function. The second kind of combinator corresponds to universal quantification: it turns an indexed set into a set.

$$\_\longrightarrow\_ : (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set})$$
$$(A \longrightarrow B)\ n = A\ n \to B\ n$$

$$\kappa : \mathsf{Set} \to (I \to \mathsf{Set})$$
$$\kappa\ A\ n = A$$

$$\_\otimes\_ : (I \to \mathsf{Set}) \to (I \to \mathsf{Set}) \to (I \to \mathsf{Set})$$
$$(A \otimes B)\ n = A\ n \times B\ n$$

$$[\_] : (I \to \mathsf{Set}) \to \mathsf{Set}$$
$$[\ A\ ] = \forall\ \{n\} \to A\ n$$

Remark: Mixfix Operators In Agda underscores correspond to positions in which arguments are to be inserted. It may be a bit surprising to see infix notations for functions taking three arguments but they are only meant to be partially applied.

Remark: Implicit Arguments We use curly braces so that the index we use is an *implicit* argument we will never have to write: Agda will fill it in for us by unification.

We can already see the benefits of these aliases. For instance the fairly compact expression $[\ (\kappa\ P \otimes Q) \longrightarrow R\ ]$ corresponds to the more verbose type $\forall\ \{n\} \to (P \times Q\ n) \to R\ n$.

Last but not least, we introduce a type constructor which takes a $\mathbb{N}$-indexed set and produces the set of valid recursive calls for a function defined by course-of-values recursion. By analogy to modal logic we call it $\Box$ after the "necessity" modality whose interpretation in a Kripke semantics is eerily similar to our type constructor.

```
record □_ (A : ℕ → Set) (n : ℕ) : Set where
 constructor mkBox
 field call : ∀ {m} → .(m < n) → A m
```

Remark: Record Wrapper Instead of defining $\Box$ as a function like the other combinators, we wrap the function space in a record type. This prevents normalisation from unfolding the combinator too eagerly and makes types more readable during interactive development.

Remark: Irrelevance The argument stating that m is strictly smaller than n is preceded by a dot. In Agda, it means that this value is irrelevant and can be erased by the compiler. In Coq, we would define the relation `_<_` in `Prop` to achieve the same.

This construct can also be understood as analogous to the later modality showing up in Guarded Type Theory [17]. It empowers the user to give precise types in a total language to programs commonly written in partial ones (see e.g. the definition of *fix* below). The first thing we can notice is the fact that $\Box$ **is a functor**; that is to say that given a natural transformation from A to B, we can define a natural transformation from $\Box\ A$ to $\Box\ B$.

```
map : [ A ⟶ B ] → [ □ A ⟶ □ B ]
call (map f A) m<n = f (call A m<n)
```

Remark: Copatterns The definition of map uses the $\Box$ field named call on the *left hand side*. This is a copattern [1], meaning that we explain how the definition is *observed* (via call) rather than *constructed* (via mkBox).

Because less than (`_<_`) is defined in terms of less than or equal (`_⩽_`), ⩽-refl which is the proof that `_⩽_` is reflexive is also a proof that any n is strictly smaller than $1 + n$. We can use this fact to write the following extract function:

```
extract : [ □ A ] → [ A ]
extract a = call a ⩽-refl
```

Remark: Counit The careful reader will have noticed that this is not quite the *extract* we would expect from a comonad: for a counit, we would need a natural transformation between $\Box\ A$ and A i.e. a function of type $[\ \Box\ A \longrightarrow A\ ]$. We will not be able to define such a function: $\Box\ A\ 0$ is isomorphic to the unit type so we would have to generate an $A\ 0$ out of thin air. The types A for which $\Box$ has a counit are interesting in their own right: they are inhabited at every single index as demonstrated by fix later on.

Even though we cannot have a counit, we are still able to define a comultiplication thanks to the fact that _<_ is transitive.

```
duplicate : [ □ A ⟶ □ □ A ]
call (call (duplicate A) m<n) p<m = call A (<-trans p<m m<n)
```

Remark: Identifiers in Agda Any space-free string which is not a reserved keyword is a valid identifier. As a consequence we can pick suggestive names such as $m<n$ for a proof that $m < n$ (notice the extra spaces around the infix operator $(<)$).

Exploring further the structure of the functor □, we can observe that just like it is not quite a comonad, it is not quite an applicative functor. Indeed we can only define *pure*, a natural transformation of type [$A \longrightarrow \square\ A$], for the types A that are downwards closed. Providing the user with *app* is however possible:

```
app : [ □ (A ⟶ B) ⟶ (□ A ⟶ □ B) ]
call (app F A) m<n = call F m<n (call A m<n)
```

Finally, we can reach what will serve as the backbone of our parser definitions: a safe, total fixpoint combinator. It differs from the traditional Y combinator in that all the recursive calls have to be guarded.

```
fix : ∀ A → [ □ A ⟶ A ] → [ A ]
```

If we were to unfold all the type-level combinators and record wrappers, the type of fix would correspond exactly to strong induction for the natural numbers. Hence its implementation also follows the one of strong induction: it is a combination of a call to extract and an auxiliary definition fix□ of type [□ $A \longrightarrow A$] → [□ A].

Remark: Generalisation A similar □ type constructor can be defined for any induction principle relying on an accessibility predicate. Which means that a library's types can be cleaned up by using these combinators in any situation where one had to give up structural induction for a more powerful alternative.

5 Parsing, Totally

As already highlighted in Section 3, *some* and *many* can yield diverging computations if the parser they are given as an argument succeeds on the empty string. To avoid any such issue, we adopt a radical solution: for a parser's run to be considered successful, it must have consumed some of its input. Some nullability can be recovered later (see Section 5.2) when defining combinators where one of the sub-parses is allowed to fail.

This can be made formal with the Success record type: a Success of type A and size n is a value of type A together with the leftovers of the input string of size strictly smaller than n.

```
record Success (A : Set) (n : ℕ) : Set where
 constructor _^_'_
 field value    : A
       {size}   : ℕ
       .small   : size < n
       leftovers : Vec Char size
```

Remark: Implicit Field Like the arguments to a function can be implicit, so can a record's fields. The user can leave them out when building a value: they will be filled in by unification.

Coming back to Fritz Ruehr's rhyme, we can define our own Parser type: a parser for things up to size n is a function from strings of length m less or equal to n to lists of Successes of size m.

```
record Parser (A : Set) (n : ℕ) : Set where
 constructor mkParser
 field runParser : ∀ {m} → .(m ⩽ n) → Vec Char m →
                   List (Success A m)
```

5.1 Our First Combinators

Now that we have a precise definition of Parsers, we can start building our library of combinators. Our first example anyChar can be defined by copattern-matching and then case analysis on the input string: if it is empty then the list of Successes is also empty, otherwise it contains exactly one element which corresponds to the head of the input string and its tail as leftovers.

```
anyChar : [ Parser Char ]
runParser anyChar _ s with s
... | []       = []
... | c :: cs = (c ^ ⩽-refl , cs) :: []
```

Unsurprisingly guardM is still a valid higher-order combinator: filtering out results which do not agree with a predicate is absolutely compatible with the consumption constraint we have drawn. To implement guardM we can once more reuse existing library functions. Relying this time on Agda's standard library rather than Haskell's, the set of available function is slightly different. We use for instance gfilter which turns a List A into a List B provided a predicate $A \to$ Maybe B and combine sequence and Success's map to obtain a function akin to *traverse*.

```
guardM : (A → Maybe B) → [ Parser A ⟶ Parser B ]
runParser (guardM p A) m⩽n s =
 gfilter (sequence ∘ Success.map p) (runParser A m⩽n s)
```

Demonstrating that Parser is a functor goes along the same lines: using List's and Success's maps. Similarly, we can prove that it is an Alternative: failing corresponds to returning the empty list no matter what whilst disjunction is implemented using concatenation.

```
_<$>_ : (A → B) → [ Parser A ⟶ Parser B ]
```

```
fail : [ Parser A ]          _<|>_ : [ Parser A ⟶ Parser A ⟶ Parser A ]
```

So far the types we have ascribed to our combinators are, if we ignore the ℕ indices, exactly the same as the ones one would find in any other parsec library. In none of the previous combinators do we run a second parser on the leftovers of a first one. All we do is either manipulate or combine the results of one or more parsers run in parallel, potentially discarding some of these results on the way.

However when we run a parser *after* some of the input has already been consumed, we could safely perform a *guarded* call. This being made explicit would be useful when using fix to define

a parser for a recursive grammar. Luckily Parser is, by definition, a downwards-closed type. This means that we may use very precise types marking all the guarded positions with □; if the user doesn't need that extra power she can very easily bypass the □ annotations by using box:

```
box : [ Parser A ⟶ □ Parser A ]
```

The most basic example we can give of such an annotation is probably the definition of a conjunction combinator _<&>_ taking two parsers, running them sequentially and returning a pair of their results. The second parser is given the type □ Parser B instead of Parser B which we would expect to find in other parsec libraries.

```
_<&>_ : [ Parser A ⟶ □ Parser B ⟶ Parser (A × B) ]
```

We can immediately use all of these newly-defined combinators to give a safe, total definition of some which takes a parser for A and returns a parser for List$^+$ A, the type of non-empty lists of As. It is defined as a fixpoint and proceeds as follows: it either combines a head and a non-empty tail using _::$^+$_ : $A \to$ List$^+$ $A \to$ List$^+$ A or returns a singleton list.

```
some : [ Parser A ] → [ Parser (List⁺ A) ]
some p = fix _ $ λ rec → uncurry _::⁺_ <$> (p <&> rec)
                       <|> (_:: []) <$> p
```

Remark: Inefficiency Unfortunately this definition is inefficient. Indeed, in the base case some p is going to run the parser p twice: once in the first branch before realising that *rec* fails and once again in the second branch. Compare this definition to the Haskell version (after inlining *many*) where p is run once and then its result is either combined with a list obtained by recursion or returned as a singleton:

```
some :: Parser a → Parser [a]
some p = (:) <$> p <*> (some p <|> pure [])
```

5.2 Failure is Sometimes an Option

This inefficiency can be fixed by introducing the notion of a potentially failing sub-parse. We use the convention that ? marks the argument of a combinator which is allowed to fail e.g. <&?> is the version of the conjunction <&> whose second argument is allowed to fail whilst <?&> may let its first argument do so. Which, in terms of types, translates to:

```
_<&>_  : [ Parser A ⟶ □ Parser B ⟶ Parser (A × B)       ]
_<&?>_ : [ Parser A ⟶ □ Parser B ⟶ Parser (A × Maybe B) ]
_<?&>_ : [ Parser A ⟶   Parser B ⟶ Parser (Maybe A × B) ]
```

The *some* p <|> *pure* [] pattern used in the definition of *some* can be translated in our total setting to a recursive call which is allowed to fail. This leads to the following rethought definition of some p. The inefficiency of the previous version has disappeared: p is run once and dependening on the success or failure of the recursive call it is either added to a non-empty list of values or returned as a singleton.

```
some : [ Parser A ] → [ Parser (List⁺ A) ]
some p = fix _ $ λ rec → cons <$> (p <&?> rec) where
```

```
cons : (A × Maybe (List⁺ A)) → List⁺ A
cons (a , just as)  = a ::⁺ as
cons (a , nothing) = a :: []
```

Remark: Non-Compositional The higher-order parser some expects a *fully* defined parser as an argument. This makes it impossible to use it as one of the building blocks of a larger, recursive parser. Ideally we would rather have a combinator of type [Parser A ⟶ Parser (List⁺ A)]. This will be addressed in the next subsection.

The potentially failing conjunction combinator _<&?>_ can be generalised to a more fundamental notion _&?»=_ which is a combinator analogous to a monad's *bind*. On top of running two parsers sequentially (with the second one being chosen based on the result obtained by running the first), it allows the second one to fail and returns both results.

```
_&?»=_ : [ Parser A ⟶ (κ A ⟶ □ Parser B) ⟶ Parser (A × Maybe B) ]
```

These definitions make it possible to port a lot of the Haskell definitions where one would use a parser which does not use any of its input. Instead of encoding a potentially failing parse using the pattern p<|>*pure* v, we can explicitly use a combinator acknowledging the authorized failure. And this is possible without incurring any additional cost as the optimised version of some showed.

5.3 Left Chains

The pattern used in the solution presented in Section 3 can be abstracted with the notion of an (heterogeneous) left chain which takes a parser for a seed, one for a constructor, and one for and argument. The crucial thing is to make sure *not* to use the parser one is currently defining as the seed.

```
hchainl :: Parser a → Parser (a → b → a) → Parser b → Parser a
hchainl seed con arg = seed >>= rest where
  rest :: a → Parser a
  rest a = do {f ← con; b ← arg; rest (f a b)} <|> pure a
```

We naturally want to include a safe variant of this combinator in our library. However this definition relies on the ability to simply use *pure* in case it's not possible to parse an additional constructor and argument and that is something we simply don't have access to.

This forces us to find the essence of *rest*, the auxiliary definition used in *hchainl*: its first argument is not just a value, it is a Success upon which it builds until it can't anymore and simply returns. We define schainl according to this analysis: it is a bare bones version of *hchainl*'s *rest* where *con* and *arg* have already been replaced by a single *con* function.

```
schainl : [ Success A ⟶ □ Parser (A → A) ⟶ List ∘ Success A ]
```

A key thing to notice is that we build a list of Successes at the *same* index as the input Success and Parser which will make this combinator compositional (as opposed to *some* defined in Section 5.2). This as a cost in terms of the readability of the definition of schainl. But ultimately all of this complexity only shows up in the implementation of our library: the end user can blissfully ignore these details.

From this definition we can derive iterate which takes a parser for a seed and a parser for a function and kickstarts a call to schainl on the result of the parser for the seed.

```
iterate : [ Parser A ⟶ □ Parser (A → A) ⟶ Parser A ]
```

Finally, hchainl can be implemented using iterate, the applicative structure of Parser and some of the properties of □.

```
hchainl : [ Parser A ⟶ □ Parser (A → B → A) ⟶ □ Parser B ⟶ Parser A ]
```

As we have mentioned when defining schainl, the combinator hchainl we have just implemented does not expect fully-defined parsers as arguments. As a consequence it can be used inside a fixpoint construction. Both the parser for the constructor and the one for its B argument are guarded whilst the one for the A seed is not. This means that trying to define a left-recursive grammar by immediately using a recursive substructure on the left is now a *type error*. But it still possible to have some on the right or after having consumed at least one character (typically an opening parenthesis, cf. the `Expr` example in Section 3).

6 Fully Worked-Out Example

From hchainl, one can derive chainl1 which is not heterogeneous and uses the same parser for the seed and the constructors' arguments. This combinator together with the idea of precedence mentioned in Section 3 is typically used to implement left-recursive grammars. Looking up the documentation of the `parsec` library on hackage [11] we can find a fine example: an extension of our early arithmetic language (corresponding grammar on the right hand side).

```
expr   = term `chainl1` addop
term   = factor `chainl1` mulop
factor = parens expr <|> integer
mulop  = do {symbol "*"; pure (*)}
       <|> do {symbol "/"; pure (div)}
addop  = do {symbol "+"; pure (+)}
       <|> do {symbol "-"; pure (-)}
```

```
Expr   := Term   | Term AddOp Expr
Term   := Factor | Factor MulOp Term
Factor := <Int>  | '(' Expr ')'
AddOp  := '+'    | '-'
MulOp  := '*'    | '/'
```

One important thing to note here is that in the end we not only get a parser for the `expr`essions but also each one of the intermediate categories `term` and `factor`. Luckily, our library lets us take fixpoints of any sized types we may fancy. As such, we can define a sized record of parsers for each one of the syntactic categories:

```
record Language (n : ℕ) : Set where
  field expr   : Parser Expr n
        term   : Parser Term n
        factor : Parser Factor n
```

Here, unlike the Haskell example, we decide to be painfully explicit about the syntactic categories we are considering: we mutually define three inductive types representing *left-associated* arithmetic expressions.

```
data Expr : Set where
  Emb : Term → Expr
  Add : Expr → Term → Expr
  Sub : Expr → Term → Expr

data Term : Set where
  Emb : Factor → Term
  Mul : Term → Factor → Term
  Div : Term → Factor → Term

data Factor : Set where
  Emb : Expr → Factor
  Lit : ℕ → Factor
```

The definition of the parser itself is then basically the same as the Haskell one. Contrary to a somewhat popular belief, working in a dependently-typed language does not force us to add any type annotation except for the top-level one.

```
language : [ Language ]
language = fix Language $ λ rec →
  let addop = Add <$ char '+' <|> Sub <$ char '-'
      mulop = Mul <$ char '*' <|> Div <$ char '/'
      factor = Emb <$> parens (map expr rec) <|> Lit <$> decimal
      term   = hchainl (Emb <$> factor) (box mulop) (box factor)
      expr   = hchainl (Emb <$> term)   (box addop) (box term)
  in record { expr = expr ; term = term ; factor = factor }
```

Although quite close to the Haskell version, We can notice four minor changes:

Firstly, the intermediate parsers need to be declared before being used which effectively reverses the order in which they are spelt out.

Secondly, the recursive calls are now explicit: in the definition of `factor`, expr is mapped under the □ to project the recursive call to the Expr Parser out of Language.

Thirdly, we use hchainl instead of chainl1 because breaking the grammar into three distinct categories leads us to parsing *h*eterogeneous left chains.

Fourthly, we have to insert calls to box to lift Parsers into boxed ones whenever the added guarantee that the call will be guarded is of no use to us. This last point however does not stand in Coq nor Idris which have a mechanism to declare implicit coercions and where the typechecker can insert these calls to box automatically for us.

7 More Power: Switching to other Representations

We have effectively managed to take Haskell's successful approach to defining a Domain Specific Language of parser combinators and impose type constraints which make it safe to use in a total setting. All of which we have done whilst keeping the concision and expressivity of the original libraries. A natural next question would be the speed and efficiency of such a library.

Although we have been using a concrete type for Parser throughout this article, our library actually implements a more general one. It uses Agda's instance arguments throughout thus letting the user pick the representation they like best.

Firstly, there is nothing special about vectors of characters as an input type: any sized input off of which one can peel characters one at a time would do. Users may instead use Haskell's `Text` packaged together with an irrelevant proof that the given text has the right length and a binding for *uncons*. This should lead to a more efficient memory representation of the text being analysed.

Secondly, there is no reason to limit ourselves to `Char` as the unit of information to be processed. Near all of our combinators are fully polymorphic over the kind of tokens they can deal with. To run the parser, the user will have to define an appropriate tokenizer for their use case. The library provides a trivial one for Char.

Thirdly, there is no reason to force the user to get back a List of successes: any Functor which is both a Monad and an Alternative will do. This means in particular that a user may for instance instrument a parser with a logging ability to be able to return good error messages, have a (re)configurable grammar using a `Reader` transformer or use a `Maybe` type if they want to make explicit the fact that their grammar is unambiguous.

8 Related Work

This work is based on Hutton and Meijer's influential functional pearl [9] which builds on Walder's insight that exception handling and backtracking can be realised using a list of successes [18]. Similar Domain Specific Languages have been implemented in various functional languages such as Scala [15] or, perhaps more interestingly for us, Rust [6] where the added type-level information about ownership can help implement a guaranteed zero-copy parser.

8.1 Total Parser Combinators

When it comes to total programming languages, Danielsson's library [7] is to our knowledge the only attempt so far at defining a library of total parser combinators in a dependently-typed host language. He reifies recursive grammars as values of a mixed inductive-coinductive type and tracks at the type level whether a sub-grammar accepts the empty word and, as a consequence, whether one can meaningfully take its fixpoint.

The reified approach allows him to define a grammar's semantics in terms of multisets of words and prove sound a variant of Brzozowski derivatives [4] as well as study the equational theory of parsers. The current implementation, based on the Brzozowski derivatives, is however of complexity at least exponential in the size of the input.

Our approach, although not able to tackle certification like Danielsson's, is however more lightweight. Using only strong induction on the natural numbers, it is compatible with languages a lot less powerful than Agda. Indeed there is no need for good support for mixed induction and coinduction in the host language. And although we do rely on enforcing invariants at the type-level, one could mimic these in languages with even weaker type systems by defining an *abstract* □ and only providing the user with our set of combinators which is guaranteed to be safe.

8.2 Certified Parsing

Ambitious projects such as CompCert [12] providing the user with an ever more certified toolchain tend to bring to light the lack of proven-correct options for very practical concerns such as parsing. Jourdan, Pottier and Leroy's work [10] fills that gap by certifying the output of an untrusted parser generator for LR(1) grammars. This approach serves a different purpose than ours: parser combinators libraries are great for rapid prototyping and small, re-configurable parsers for non-critical applications.

Bernardy and Jansson have implemented in Agda a fully-certified generalisation of Valiant's algorithm [2] by deriving it from its specification. This algorithm gives the best asymptotic bounds on context-free grammar, that is the `Applicative` subset tackled by parser combinators.

9 Conclusion and Future Work

Starting from the definition of "parsers for things as functions from strings to lists of strings and things" common in Haskell, we have been able to (re)define versatile combinators. However the type system was completely unable to root out some badly-behaved programs, namely the ones taking the fixpoint of a grammar accepting the empty word or non well-founded left recursive grammars. Wanting to use a total programming language, this led us to a radical solution: rejecting all the parsers accepting the empty word. Luckily, it was still possible to recover a notion of "potentially failing" sub-parses via a *bind*-like combinator as well as defining combinators for left chains. Finally we saw that this yielded a perfectly safe and only barely more verbose set of total parser combinators.

In the process of describing our library we have introduced a set of type-level combinators for manipulating indexed types and defining values by strong induction. If we want to provide our users with the tools to modularly prove some of the properties of their grammars, we need to come up with proof combinators corresponding to the value ones. As far as we know this is still an open problem.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[2] Jean-Philippe Bernardy and Patrik Jansson. Certified context-free parsing: A formalisation of Valiant's algorithm in Agda. *arXiv preprint arXiv:1601.07724*, 2016.

[3] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[4] Janusz A Brzozowski. Derivatives of regular expressions. *Journal of the ACM (JACM)*, 11(4):481–494, 1964.

[5] David Raymond Christiansen. Dependent type providers. In *Proceedings of the 9th ACM SIGPLAN workshop on Generic programming*, pages 25–34. ACM, 2013.

[6] Geoffroy Couprie. Nom, a byte oriented, streaming, zero copy, parser combinators library in rust. In *Security and Privacy Workshops (SPW), 2015 IEEE*, pages 142–148. IEEE, 2015.

[7] Nils Anders Danielsson. Total parser combinators. In *ACM Sigplan Notices*, volume 45, pages 285–296. ACM, 2010.

[8] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[9] Graham Hutton and Erik Meijer. Monadic parsing in Haskell. *Journal of functional programming*, 8(4):437–444, 1998.

[10] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. Validating LR (1) parsers. In *ESOP*, volume 7211, pages 397–416. Springer, 2012.

[11] Daan Leijen, Paolo Martini, and Antoine Latter. Parsec documentation. https://hackage.haskell.org/package/parsec-3.1.11/docs/Text-Parsec.html, 2017. Retrieved on 2017-10-30.

[12] Xavier Leroy et al. The CompCert verified compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[13] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[14] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

[15] Adriaan Moors, Frank Piessens, and Martin Odersky. Parser combinators in Scala. Technical report, 2008.

[16] Ulf Norell. Dependently typed programming in Agda. In *AFP Summer School*, pages 230–266. Springer, 2009.

[17] Andrea Vezzosi. Guarded recursive types in type theory. Licentiate thesis, Chalmers University of Technology, 2015.

[18] Philip Wadler. How to replace failure by a list of successes. In *Functional Programming Languages and Computer Architecture*, pages 113–128. Springer, 1985.

Domaines spatio-temporels: un tour d'horizon

David Janin*

UMR CNRS LaBRI, Bordeaux INP
Université de Bordeaux
janin@labri.fr

Résumé

Une échelle spatio-temporelle est définie comme un ensemble (partiellement) ordonné dont les éléments sont appelé instants. Deux instants comparables sont placés dans le temps, l'un avant l'autre, ou l'un après l'autre. Deux instants incomparables sont placés dans l'espace, l'un à coté de l'autre. Certains ensembles ordonnées indexés sur de telles échelles sont appelés domaines spatio-temporels, des domaines qu'on peut transformer à l'aide de fonctions croissantes, dès lors qu'elles agissent uniformément sur les échelles spatio-temporelles sous-jacentes. Nous proposons dans cet article un tour d'horizon des propriétés tout à fait remarquable de ces domaines et de leur transformations ainsi que leurs liens avec de nombreux modèles et concepts apparus en sémantique des systèmes informatisés et des langages de programmation : des topologies ultra-métriques aux structures d'évènements, du parallélisme synchrone à la concurrence, qu'ils soient temporisés ou pas.

1 Introduction

La « convergence numérique » est aujourd'hui un enjeu industriel et sociétal majeur. Aux yeux du grand public, elle signifie la convergence des supports de communication tels que textes, sons, photos, vidéos, animations, interfaces de saisies, etc., et de leurs domaines d'utilisation tels que musique, cinéma, télévision, téléphonie, apprentissage, conduite assistée, aide à la personne, etc. De nouvelles applications des technologies du numérique témoignent chaque jour de l'effectivité de cette convergence du numérique.

En informatique, cette convergence fait que de nombreux systèmes informatisés se composent maintenant de sous-systèmes hétérogènes qui agissent et interagissent sur des échelles de temps et d'espaces différentes. Peu ou prou, les concepteurs de ces systèmes sont confrontés à la difficile mise au point d'interfaces hybrides, au sens le plus général, qui doivent permettre à ces sous-composants d'interagir de façon cohérente en échangeant et en transformant des données spatiales et/ou temporelles de nature différente.

L'objet de cet article est de développer une notion de domaines spatio-temporisés qui doit permettre, à termes, de modéliser de façon homogène et unifiée la sémantique de ces interfaces hybrides et des comportements des sous-systèmes hétérogènes qui interagissent à travers elles. Ce faisant, nous proposons une sorte de relecture, abstraite, de plusieurs concepts et approches développés ces dernières décennies pour la modélisation et/ou la programmation de systèmes informatisés temporisés complexes.

D'un point de vue théorique, les domaines temporisés et les fonctions Δ-synchrones qui leurs sont associées offrent toutes les caractéristiques d'un modèle adéquat : des produits, des co-produits, des exponentiels, qu'ils soient synchrones ou asynchrones, ainsi que des opérateurs de points-fixes qui sont eux-même temporisés. En pratique, nous ne proposons pas (encore) de réelle mise en œuvre. Quelques examples s'appuyant sur la notion de signaux d'évènements temporisés sur des échelles de temps arbitraires illustrent cependant tout au long de cet article le potentiel applicatif de notre approche.

*Ce travail a été soutenu par le centre Inria de Bordeaux Sud-Ouest de septembre 2016 à février 2017.

2 Ensembles temporisés

Rappelons qu'un *ensemble (partiellement) ordonné* est un ensemble P muni d'une relation binaire, notée $\leq$, qui est réflexive, transitive et anti-symétrique, c'est à dire que pour toute $x, y, z \in P$ on a $x \leq x$ (réflexivité), si $x \leq y$ et $y \leq z$ alors $x \leq z$ (transitivité) et si $x \leq y$ et $y \leq x$ alors $x = y$ (anti-symmétrie). Etant donné un ensemble partiellement ordonné P, on note

$$\downarrow x = \{y \in P : y \leq x\} \quad \text{et, resp.,} \quad \downarrow X = \{y \in P : \exists x \in X, y \leq x\}$$

la *clôture par le bas* de tout élement $x \in P$ et, resp., toute partie $X \subseteq P$. On dira qu'une partie $X \subseteq P$ est *close par le bas* lorsque $X = \downarrow X$.

Définition 2.1 (Echelles de temps)**.** Une *échelle spatio-temporelle*, ou, plus simplement, *échelle de temps*, est simplement définie comme un ensemble partiellement ordonné T dont les éléments sont appelés *instants*. Pour tout $u, v \in T$, si $u \leq v$ on dit que l'instant u est situé *avant* l'instant v, ou bien que l'instant v *après* l'instant u. Dans le cas où u et v ne sont pas comparables, on dit que u et v sont *à coté* l'un de l'autre. Les liens entre temps et espace ne sont pas spécifiés plus que cela. Ils dépendent de l'échelle spatio-temporelle choisie et de l'interprétation que l'on souhaite lui donner.

Définition 2.2 (Ensembles temporisés)**.** Soit T une échelle de temps. Un *ensemble temporisé* sur T est un ensemble partiellement ordonné P équipé d'une *projection temporelle* $\pi : P \to T$ telle que :

(IN1) si $x \leq y$ alors $\pi(x) \leq \pi(y)$, i.e. la projection temporelle est croissante,

(IN2) il existe un unique élément $x \downarrow u \in P$ tel que $\pi(x \downarrow u) = u$ et $x \downarrow u \leq x$,

pour tout element $x, y \in P$ et tout instant $u \in T$ tel que $u \leq \pi(x)$. L'élément $x \downarrow u$ est appelé la *coupure temporelle* de x en u.

Remarque 2.3. Les conditions (IN1) et (IN2) peuvent être schématisées par le diagramme suivant.

$$\begin{array}{ccccc} x\downarrow u & \xrightarrow{\leq} & x & \xrightarrow{\leq} & y \\ {\scriptstyle \exists!\pi}\downarrow & \text{(IN2)} & \downarrow{\scriptstyle \pi} & \text{(IN1)} & \downarrow{\scriptstyle \pi} \\ u & \xrightarrow[\leq]{} & \pi(x) & \xrightarrow[\leq]{} & \pi(y) \end{array}$$

Lorsque P et T sont tout deux vus comme des catégories, la projection temporelle $\pi : P \to T$ définit ce qu'on appelle une *fibration discrète*.

Exemple 2.4 (Ensembles auto-temporisés)**.** Tout ensemble ordonné T est un ensemble temporisé sur lui-même en prenant l'identité comme projection temporelle.

Exemple 2.5 (Sous-ensembles temporisés)**.** Soit P un ensemble temporisé. Soit $X \subseteq P$ une partie de P close par le bas. Alors l'ensemble X muni de la restriction de la relation d'ordre et de la projection temporelle de P à X est un ensemble temporisé.

Exemple 2.6 (Signaux temporisés)**.** Soit T une échelle de temps et E un ensemble de valeurs d'évènement. Un *signal partiel temporisé* est une paire $(u, X) \in T \times \mathcal{P}(T \times E)$ telle que pour tout $(t, e) \in X$ on a $t \leq u$. L'ensemble de ces signaux partiels temporisés, noté $Sig(T, E)$, est équipé d'un *ordre préfixe* défini par $(u, x) \leq (v, Y)$ lorsque $u \leq v$ et $X = \{(t, e) \in Y : t \leq u\}$ pour tout $(u, X), (v, Y) \in Sig(T, E)$. Avec la projection temporelle $\pi : Sig(T, E) \to T$ définie par $\pi(u, X) = u$ pour tout $(u, X) \in Sig(T, E)$, l'ensemble $Sig(T, E)$ devient ainsi un ensemble temporisé sur T.

Remarque 2.7. Dans l'exemple ci-dessus, un signal temporisé sur T est modélisé comme une fonction $s : T \to \mathcal{P}(E)$ qui indique l'ensemble des valeurs $s(t) \subseteq E$ qui sont reçues à chaque instant $t \in T$. Un tel signal peut, de façon équivalente, être représenté par l'ensemble $X_s = \{(t, e) \in T \times E : e \in s(t)\}$. En posant $X_s \downarrow u = \{(t, e) \in X_s : t \leq u\}$, le signal partiel temporisé $(u, X_s \downarrow u) \in Sig(T, E)$ représente alors la *trace* du signal s jusqu'à l'instant $u \in T$. L'ensemble temporisé $Sig(T, E)$ permet donc de représenter de façon structurée les traces de ces signaux telles qu'elles évoluent dans le temps (et l'espace) défini par T.

Définition 2.8 (Coupure locale)**.** Soit P un ensemble temporisé sur une échelle de temps T. Pour tout $x \in P$, la propriété (IN2) induit une fonction de *coupure locale* $cut_x : \downarrow \pi(x) \to P$ définie pour tout $u \in T$ tel que $u \leq \pi(x)$ par $cut_x(u) = x \downarrow u$.

Lemme 2.9 (Propriété de la coupure locale)**.** *Soient $x, y \in P$ et $u, v \in T$ tels que $u, v \leq \pi(x)$.*
(1) $x \leq y$ si et seulement si $\pi(x) \leq \pi(y)$ et $x = y \downarrow \pi(x)$,
(2) $u \leq v$ si et seulement si $x \downarrow u \leq x \downarrow v$.
En particulier, $\downarrow \pi(x)$ et $\downarrow x$ sont deux (sous-)ensembles partiellement ordonnée isomorphes, la coupure locale cut_x étant un isomorphisme.

Remarque 2.10. Un corollaire immédiat du lemme précédent est qu'un élement $x \in P$ est *minimal* dans P si est seulement si sa projection $\pi(x) \in T$ est minimal dans T. Par contre, les éléments maximaux d'un ensemble temporisé peuvent avoir une projection temporelle arbitraire comme illustré par l'ensemble $T \times \{0, 1\}$ ordonné par (la clôture réflexive de) $(u, b) < (v, c)$ lorsque $u < v$ et $b = 0$ pour tout $(u, b), (v, c) \in T \times \{0, 1\}$, avec la première projection comme projection temporelle.

3 Notions dérivées

La notion d'ensembles temporisés peut être reliée à de nombreux autres concepts existants en modélisation de système informatique ou en sémantique des langages de programmation.

Cohérence spatio-temporelle. La définition suivante est inspirée de la notion de cohérence utilisée par Girard pour la logique linéaire [13]. Cependant, en l'absence d'interprétation significative de la dualité qu'elle pourrait induire sur les ensembles temporisés, nous n'établissons aucune connection réelle avec la logique linéaire.

Définition 3.1. Soit P un ensemble temporisé sur T. Soient $x, y \in P$. On dit que x et y sont *temporellement cohérents*, ce qui est noté $x \frown y$, lorsque, pour tout $x', y' \in P$, si $x' \leq x$ et $y' \leq y$ avec $\pi(x') = \pi(y')$ alors $x' = y'$, ou, de façon équivalente, $x \downarrow u = y \downarrow u$ pour tout $u \leq \pi(x), \pi(y)$.

Lemme 3.2. *Pour tout $x, y \in P$ on a $x \leq y$ si et seulement si $x \frown y$ et $\pi(x) \leq \pi(y)$.*

Remarque 3.3. Un sous-ensemble $X \subseteq P$ est dit *temporellement cohérent* lorsque pour tout $x, y \in X$ on a $x \frown y$. On peut vérifier que toute partie dirigée est cohérente, tout comme la clôture par le bas de tout ensemble cohérent. On peut aussi vérifier que si X est cohérent alors la restriction de la projection temporelle π à l'ensemble X est un isomorphisme d'ensembles partiellement ordonnés.

Remarque 3.4. Un ensemble temporisé est une structure d'évènement au sens de Winskel [23]. On hérite ainsi des mêmes interprétations comportementales. Techniquement, la relation d'équivalence induite par la projection temporelle, i.e. deux éléments sont équivalents lorsqu'ils ont même projection, est, à un détail technique près, ce que Winskel appel une symmétrie [23]. Dans notre cas, cette symmétrie à la particularité de caractériser la relation de cohérence. La notion d'ensembles temporisés est plus restrictive que la notion de structures d'évènements.

Distance temporelle. D'autres approches de modélisation de systèmes concurrents s'appuient sur la notion de distances ultra-métriques généralisées utilisées en particulier par Lee, Liu et Matsikoudis [20, 17, 18]. Cette notion apparait aussi avec les ensembles temporisés.

Définition 3.5. Soit P un ensemble temporisé sur T. Soit $\mathcal{P}^{\downarrow}(T)$ l'ensemble des parties de T closes par le bas, ordonnées par inclusion inverse, c'est à dire avec T considéré comme le plus petit élement. La fonction $d : P \times P \to \mathcal{P}^{\downarrow}(T)$ définie par :

$$d(x, y) = \{t \in T : t \leq \pi(x), \pi(y), x \downarrow t = y \downarrow t\}$$

lorsque $x \neq y$ et par $d(x, y) = T$ lorsque $x = y$ appelée la *distance temporelle* induite par P.

On vérifie sans peine que pour tout $x, y \in P$, on a bien $d(x, y) = \downarrow d(x, y)$. Autrement dit, l'ensemble $d(x, y) \subseteq T$ est bien clos par le bas.

Lemme 3.6. *La distance temporelle $d : P \times P \to \mathcal{P}^{\downarrow}(T)$ induite par P est une distance ultra-métrique généralisée [20], c'est à dire qu'on a :*
(1) $d(x, y) = T$ si et seulement si $x = y$ (séparation),
(2) $d(x, y) = d(y, x)$ (symmetrie),
(3) $d(x, y) \supseteq d(x, z) \cap d(z, y)$ (inégalité ultra-métrique),
pour tout $x, y, z \in P$.

Remarque 3.7. L'ensemble $\mathcal{P}^{\downarrow}(T)$ est un treillis complet ; la borne inf. (resp. la borne sup.) de tout ensemble de parties de T closes par le bas est l'intersection (resp. l'union) de ces parties. Question ouverte : étant donné un ensemble Q, un treillis complet T et une fonction de distance ultra-métrique généralisée $d : Q \times Q \to T^{op}$, on peut se demander sous quelles conditions l'ensemble Q peut être ordonné et équipé d'une projection $\pi : Q \to T$ de telle sorte qu'il soit un ensemble temporisé dont la distance induite est la distance d.

Pré-faisceaux temporels. Une dernière caractérisation des ensembles temporisés, cette fois ci complète, passe par la notion catégorique de pré-faisceaux ; une notion utilisée en modélisation des systèmes concurrents par, notamment, Catani, Stark et Winskel [7, 6] pour modéliser la sémantique des calculs de processus.

Définition 3.8. Un pré-faisceau sur T est un foncteur $F : T^{op} \to Set$, l'ensemble ordonnée T étant vu comme la catégorie dont les objets sont les éléments de T et les flèches les pairs d'objets $(u, v) \in T \times T$ telles que $u \leq v$.

Lemme 3.9. *Soit P un ensemble temporisé sur l'échelle de temps T. La (double) fonction $F : T^{op} \to Set$ définie par :*
(1) $F(t) = \{x \in P : \pi(x) = t\}$,
(2) $F(u \leq v)(x) = x \downarrow u$ pour tout $x \in F(v)$,
pour tout $t, u, v \in T$ avec $u \leq v$ est un pré-faisceau sur T.

Lemme 3.10. *Soit $F : T^{op} \to Set$ un pré-faisceau sur T. On définit :*
(1) $P = \Sigma_{t \in T} F(t) = \{(t, x) : t \in T, x \in F(t)\}$,
(2) $(u, x) \leq (v, y)$ lorsque $u \leq v$ et $F(u \leq v)(y) = x$,
(3) $\pi(u, x) = u$,
pour tout $(u, x), (v, y) \in P$. Alors P est un ensemble temporisé sur T.

Remarque 3.11. L'ensemble P défini ci-dessus à partir du pré-faisceau F est aussi appelé la catégorie des éléments de F. C'est un cas particulier de la construction plus générale dite de Grothendieck. On vérifie sans peine que les transformations décrites ci-dessus sont, à isomorphismes près, des bijections.

4 Fonctions temporisées

Fonctions Δ-synchrones. La classe la plus générale de fonctions entre deux ensembles temporisés que nous considérons dans cet article est la classe des fonctions appelée Δ-synchrone qui peuvent agir sur (et changer) les échelles de temps sous-jacentes mais de façon uniforme.

Définition 4.1. Soient P et Q deux ensembles temporisés, le premier sur une échelle U, le second sur une échelle V. Une fonction $f : P \to Q$ est appelé *Δ-synchrone* lorsqu'il existe une fonction *monotone* $\delta : U \to V$, appelée la *projection temporelle* de f, telle que :

(SD1) $\pi(f(x)) = \delta \circ \pi(x)$,

(SD2) si $x \leq y$ alors $f(x) \leq f(y)$,

(SD3) $f(x \downarrow u) = f(x) \downarrow \delta(u)$,

pour tout $x, y \in P$ et tout $u \in U$ tel que $u \leq \pi(x)$. Les propriétés (SD1) et (SD2) sont illustrées ci-dessous.

$$\begin{array}{ccc} U & \xleftarrow{\pi_P} & P \\ {\scriptstyle\delta}\downarrow & \text{(SD1)} & \downarrow{\scriptstyle f} \\ V & \xleftarrow[\pi_Q]{} & Q \end{array} \qquad \begin{array}{ccc} P & \xrightarrow{\leq_P} & P \\ {\scriptstyle f}\downarrow & \text{(SD2)} & \downarrow{\scriptstyle f} \\ Q & \xrightarrow[\leq_Q]{} & Q \end{array}$$

Exemple 4.2 (Fonctions auto-synchrones)**.** Toute fonction monotone $\delta : U \to V$ est Δ-synchrone avec elle-même comme projection temporelle.

Lemme 4.3. *Sous l'hypothèse (SD1), les propriétés (SD2) et (SD3) sont équivalentes.*

Fonctions causales. Lorsque les échelles de temps en entrée et en sortie d'une fonction sont les mêmes, on peut définir une notion de causalité temporelle.

Définition 4.4. Une fonction Δ-synchrone $f : P \to Q$ de projection temporelle $\delta : T \to T$ est *temporellement causale* lorsque pour tout $t \in T$ on a $t \leq \delta(t)$.

Remarque 4.5. Autrement dit, lorsque f est temporellement causale, notre connaissance de la sortie $f(x)$ est posterieure à notre connaissance de l'entrée x. En effet, de la propriété (SD1) on déduit immédiatement que pour tout $x \in P$ on a $\pi(x) \leq \pi(f(x))$.

Fonctions synchrones. On s'interesse aussi à la classe particulière des fonctions causales dites synchrones au sens où les sorties de ces fonctions sont produites en même temps (et au même endroit) que leurs entrées.

Définition 4.6 (Fonctions synchrones)**.** Une fonction $f : P \to Q$, Δ-synchrone, de projection temporelle $\delta : U \to V$ est une fonction *synchrone* lorsque $U = V$ et $\delta = id_U$. Autrement dit, on doit avoir

(SI1) $\pi(f(x)) = \pi(x)$,

(SI2) si $x \leq y$ alors $f(x) \leq f(y)$,

(SI3) $f(x \downarrow u) = f(x) \downarrow u$,

pour tout $x \in P$ et $u \in U$ tel que $u \leq \pi(x)$.

Théorème 4.7. *Soit T une échelle de temps. Alors la catégorie $TPoset(T)$ des ensembles temporisés sur T et des fonctions synchrones est* équivalente *à la catégorie $Psh(T)$ des pré-faisceaux et des transformations naturelles sur ces pré-faisceaux.*

Le lemme suivant, qui relie les notions de fonctions synchrones et de fonctions causales, illustre la distinction qu'on peut faire entre la *connaissance* qu'on peut avoir de la sortie d'une fonction, dans le cas d'une fonction Δ-synchrone causale, avec la *production* de cette sortie en même temps qu'une entrée est reçue, dans le cas de sa projection synchrone.

Lemme 4.8. *Soit $f : P \to Q$ une fonction Δ-synchrone causale de projection temporelle $\delta : T \to T$. Alors la* projection synchrone *$f_c : P \to Q$, définie par $f_c(x) = f(x)\downarrow\pi(x)$ pour tout $x \in P$, est une fonction synchrone.*

Nous reviendrons sur la notion de fonctions causales section 7 en étudiant les points-fixes des fonctions Δ-synchrones d'un ensemble temporisé dans lui-même.

Synchrone vs. Δ-synchrone. Les notions de fonctions Δ-synchrones et de fonctions synchrones sont étroitement liées comme l'illustre le théorème suivant.

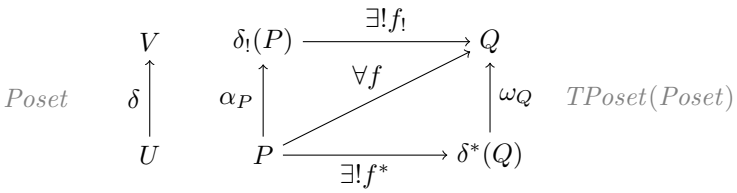
Théorème 4.9 (Extension de Kan à gauche)**.** *Soit $\delta : U \to V$ une fonction monotone. Il existe deux foncteurs $\delta_! : TPoset(U) \to TPoset(V)$ et $\delta^* : TPoset(V) \to TPoset(U)$ et, pour tout ensemble P temporisé sur U et Q temporisé sur V, deux fonctions Δ-synchrones $\alpha_P : P \to \delta_!(P)$ et $\omega_Q : \delta^*(Q) \to Q$ telles que, pour toute fonction Δ-synchrone $f : P \to Q$ de projection temporelle δ :*

(1) il existe une unique fonction synchrone $f_! : \delta_!(P) \to Q$ telle que $f = f_! \circ \alpha_P$,

(2) il existe une unique fonction synchrone $f^* : P \to \delta^*(Q)$ telle que $f = \omega_Q \circ f^*$.

Autrement dit, f se factorise de façon unique à travers α_P et à travers ω_Q.

La situation décrite par le théorème 4.9 est illustrée ci-dessous. En théorie des catégories, il y a une adjonction $\delta_! \dashv \delta^*$.



Remarque 4.10. En général, l'extension de Kan à gauche décrite par le foncteur α_P dans le théorème 4.9 n'est pas effective [15] au sens où la définition de $\delta_!(P)$ repose sur la construction, pour tout $x \in P$, de la classe des traces indiscernables de x via δ. Sous certaines hypothèses, dont on donne un exemple ci-dessous, cette construction devient effective sur les signaux temporisés.

Exemple 4.11 (Signaux temporisés)**.** En poursuivant l'exemple des signaux temporisé prenons $P = Sig(U, A)$ et $Q = Sig(V, B)$. Le théorème précédent peut être partiellement illustré de la façon suivant.

Pour le triangle inférieur, on pose $\delta^*(Q) = Sig^*(\delta, B)$ définit comme l'ensemble de toutes les paires $(u, Y) \in U \times \mathcal{P}(V \times B)$ telles que $v \leq \delta(u)$ pour tout $(v, b) \in Y$ avec la coupure définie par $(u, Y)\downarrow u' = (u', Y')$ lorsque $Y' = \{(v, b) \in Y : v \leq \delta(u')\}$ pour tout $(u, Y) \in Sig^*(\delta, B)$ et $u' \leq u$.

On peut alors définir la fonction $\omega_Q : Sig^*(\delta, B) \to Sig(V, B)$ par $\omega_Q(u, Y) = (\delta(u), Y)$ pour tout $(u, Y) \in Sig^*(\delta, B)$ at, pour toute fonction $f : Sig(U, A) \to Sig(V, B)$ comme ci-dessus, la fonction $f^* : Sig(U, A) \to Sig^*(\delta, B)$ (nécessairement) définie par $f^*(u, X) = (u, Y)$ lorsque $f(u, X) = (\delta(u), Y)$ pour tout $(u, X) \in Sig(U, A)$.

Remarque 4.12. Chaque échelle de temps décrivant une sorte de granularité à laquelle le comportement d'un système est observé, cette construction n'est pas sans rappeler les techniques d'interprétations abstraites développées depuis quelques années par Cousot et al. [10].

Préservation de la cohérence temporelle. Pour finir notre présentation des fonctions Δ-synchrones, on constate que, a priori, elles ne préservent pas la cohérence comme le montre l'exemple suivant.

Exemple 4.13. Soit $U = \{\bot, a, b\}$, auto-temporisé, avec minimum $\bot$ et a et b incomparables. Soit $P = U$ ordonné comme U mais temporisé sur $V = \{0, 1\}$ par $\pi(\bot) = 0$ et $\pi(a) = \pi(b) = 1$. Alors la fonction $f : U \to P$ définie par $f(\bot) = \bot$, $f(a) = a$ et $f(b) = b$ est Δ-synchrone de projection temporelle $\delta : U \to V$ définie par $\delta(\bot) = 0$ et $\delta(a) = \delta(b) = 1$. On vérifie que f ne préserve pas la cohérence puisque $a \frown b$ dans U alors que $f(a) = a \not\frown b = f(b)$ dans P. Le lemme suivant donne une condition suffisante pour qu'une fonction Δ-synchrone préserve la cohérence.

Lemme 4.14. *Supposons que U et V sont des inf. semi-treillis, c'est à dire des ensembles ordonnées pour lesquels tout élement u et v ont une borne inférieure $u \wedge v$. Supposons que $\delta : U \to V$ préserve les inf., i.e. pour tout $u_1, u_2 \in U$ on a $\delta(u_1 \wedge u_2) = \delta(u_1) \wedge \delta(u_2)$. Soit $f : P \to Q$ Δ-synchrone de projection temporelle δ. Alors f preserve la cohérence.*

Remarque 4.15. Cette restriction à la catégorie *SLat* des semi-treillis et des fonctions préservant les inf. pour les échelles de temps et leurs transformations ne change rien à la validité du théorème 4.9. De plus, on peut vérifier que dans ce cas, les ensembles temporisés sont des semi-treillis conditionnels, c'est à dire que toute paire d'élement x, y majoré par un élement z admet une borne inf. $x \wedge y = z{\downarrow}\pi(x) \wedge \pi(y)$. Dans ce cas, on vérifie que toute fonction Δ-synchrone est stable au sens de Berry [1, 2] et son extension point-à-point aux ensembles cohérents est linéaire au sens de Girard [13].

Propriétés catégoriques des fonctions Δ-synchrones. Etant donnée la catégorie *Poset* des ensembles (partiellement) ordonnées et des fonctions mononotes, on peut considérer la catégorie $TPoset(Poset)$ des ensembles temporisés et des fonctions Δ-synchrones. Il n'est pas difficile de montrer que c'est une catégorie cartésienne close, les constructions de l'objet terminal, du produit et de l'exponentiel étant analogue aux constructions usuels dans *Set*.

Plus précisément, l'objet terminal est le singleton $\{*\}$ temporisé sur lui-même. Pour tout ensemble temporisés P sur l'échele U et Q sur l'échelle V, le produit $P \times Q$ est ordonné point-à-point et simplement temporisé sur $U \times V$. L'exponentiel Q^P des fonctions Δ-synchrones de P dans Q est aussi ordonnée point-à-point et temporisé sur V^U. D'un certaine façon, ces produits et exponentielles sont des constructeurs *asynchrones* puisqu'ils ne requièrent aucune synchronisations des échelles de temps ou des fonctions Δ-synchrones qu'ils combinent.

On peut aussi, afin de préserver notre interprétation de la cohérence temporelle, se restreindre à la catégorie $TPoset(SLat)$ des ensembles temporisés sur des semi-treillis et des fonctions Δ-synchrones de projections temporelles qui préservent les inf. de ces semi-treillis. Les éléments terminaux, les produits et les exponentielles restent définis de la même façon.

5 Propriétés catégoriques des fonctions synchrones

Le théorème 4.9 nous montre que les propriétés des fonctions Δ-synchrones sont intimement liées aux propriétés des fonctions synchrones. Dans cette section, nous étudions les propriétés de la catégorie $TPoset(T)$ des ensembles temporisés sur T et des fonctions synchrones qui se trouve

être un topos. Néanmoins, nous ne ferons qu'évoquer cette propriété remarquable puisque nous présenterons plutôt les constructions qui en découlent sous le jour de leur interprétation possible en sémantique des langages de programmations temporisés.

Lemme 5.1 (Horloge)**.** *Soit P un ensemble temporisé sur T. Alors sa projection temporelle $\pi : P \to T$ est l'*unique *fonction synchrone de P dans T.*

Remarque 5.2. Autrement dit, l'échelle de temps T, auto-temporisée, est l'*objet terminal* dans la catégorie $TPoset(T)$. D'une certaine façon, l'échelle de temps T peut-être vue dans la catégorie $TPoset(T)$ comme une horloge qui a la capacité d'égrener les instants. Par exemple, toute fonction synchrone $c : T \to P$ peut être vue comme une *constante temporisée* dont la trace se développe avec les instants de T.

Observons au passage qu'une telle fonction n'existe pas forcément. Son existence revient à spécifier dans P un sous-ensemble cohérent $X \subseteq P$, clos par le bas, tel que $\pi(X) = T$. Un tel ensemble n'existe pas nécessairement. En effet, comme nous l'avons vu, tout sous-ensemble d'un ensemble temporisé qui est clos par le bas définit aussi un ensemble temporisé.

Définition 5.3 (Produit synchrone)**.** Soient P, Q deux ensembles temporisés sur T. Le produit synchronisé de P et Q est défini par $P \otimes Q = \{(x, y) \in P \times Q : \pi_P(x) = \pi_Q(y)\}$ ordonné point-à-point avec comme projection temporelle $\pi(x, y) = \pi(x)$ $(= \pi(y))$ pour tout $(x, y) \in P \otimes Q$.

Lemme 5.4. *Alors $P \otimes Q$ est un ensemble temporisé sur T. Les projection $p_1 : P \otimes Q \to P$ et $p_2 : P \otimes Q \to Q$ sont des fonctions synchrones. De plus, toute paire de fonctions synchrones $f : R \to P$ et $g : R \to Q$ se factorise de façon unique à travers $P \otimes Q$ via les projections p_1 et p_2, i.e. il existe une unique fonction synchrone $f \times g : R \to P \otimes Q$ telle que $f = p_1 \circ (f \times g)$ et $g = p_2 \circ (f \times g)$.*

Remarque 5.5. Autrement dit, $P \otimes Q$ équipé des projections p_1 et p_2 est bien le *produit* de P et Q dans la catégorie $TPoset(T)$. D'une certaine façon, la fonction $f \times g : R \to P \otimes Q$ représente l'execution parallèle et synchrone des fonctions f et g. C'est l'analogue, dans le cas temporisé, du produit de flux qu'on trouve couramment en programmation par flot de donnée ou en *arrow programming* [14]. Tout se passe comme si les traces des calculs modélisés par $P \otimes Q$ se déroulent sur deux processeurs synchronisés sur T, le premier réalisant un calcul dont la trace est dans P, le second réalisant un calcul dont la trace est dans Q.

Exemple 5.6 (Signaux temporisés)**.** En poursuivant nos exemples sur les signaux temporisés, remarquons que le produit synchrone $Sig(U, A) \otimes Sig(U, B)$ peut être directement modélisé par $Sig(U, A \oplus B)$ où $A \oplus B$ est la somme disjointe des ensembles de valeurs d'évènements A et B.

Remarque 5.7. On peut aussi vérifier que la somme disjointe $P \oplus Q$ de deux ensembles temporisés conduit à définir le co-produit P et Q dans la catégorie $TPoset(T)$, l'ensemble temporisé vide valant pour object initial. Ce co-produit peut être utilisé pour modéliser le résultat d'une alternative. Néanmoins, on peut constater que cette alternative devra, dans la plupart des cas, être résolue statiquement, « avant » le lancement du programme. En effet, si toute paire d'instants de T admet un minorant, alors deux éléments de P et Q plongé dans $P \oplus Q$ sont nécessairement incohérents et ne peuvent donc apparaître dans une même trace de calcul.

Définition 5.8 (Coupure temporelle d'une fonction synchrone)**.** Soit $f : P \to Q$ une fonction synchrone sur T. Pour tout $u \in T$, on définit la coupure temporelle de f en u comme la fonction

$$f{\downarrow}u : P{\downarrow}u \to Q{\downarrow}u$$

définie par $P\downarrow u = \{x \in P : \pi(x) \leq u\}$, $Q\downarrow u = \{y \in Q : \pi(y) \leq u\}$, et $(f\downarrow u)(x) = f(x)$ pour tout $x \in P\downarrow u$.

Remarque 5.9. Dans la définition précédente, les ensembles $P\downarrow u$ et, resp., $Q\downarrow u$, sont des sous-ensembles de P et, resp. Q clos par le bas. Ce sont donc bien des ensembles temporisés sur $T\downarrow u$ dès lors qu'ils sont équipés des restrictions des projections temporelles et de la relation d'ordre sur P et, resp., sur Q. Comme f est synchrone, on vérifie alors qu'on a bien $f(x) \in Q\downarrow u$ pour tout $x \in P\downarrow u$ et que la fonction $f\downarrow u$ est bien une fonction synchrone.

Définition 5.10 (Exponentiel synchrone)**.** Soient P et Q deux ensembles temporisés sur T. On définit l'exponentiel Q^P de Q par P comme étant l'ensemble des paires de la forme (u, h) avec $u \in T$ et $h : P\downarrow u \to Q\downarrow u$ synchrone. L'ensemble Q^P est équipé de la projection temporelle définie par $\pi(u, h) = u$ et de la relation d'ordre définie par $(u_1, h_1) \leq (u_2, h_2)$ lorsque $u_1 \leq u_2$ et $h_1 = h_2\downarrow u_1$ pour tout $(u, h), (u_1, h_1), (u_2, h_2) \in Q^P$.

Lemme 5.11. *Pour tous ensembles Q et R temporisés sur T, l'exponentiel R^Q de Q par R est bien un ensemble temporisé sur T. La fonction $eval_T : R^Q \otimes Q \to R$ définie pour tout $((u, h), y) \in R^Q \otimes Q$ par $eval_T((u, h), y) = h(y)$ est aussi une fonction synchrone. De plus, pour toute fonction synchrone $g : P \otimes Q \to R$, la fonction $g^* : P \to R^Q$ définie, pour tout $x \in P$, par $g^*(x) = (\pi(x), \lambda y.g(x\downarrow\pi(y), y))$, synchrone, est l'unique fonction telle que $g(x, y) = eval_T(g^*(x), y)$ pour tout $(x, y) \in P \otimes Q$.*

Remarque 5.12. Autrement dit, l'ensemble temporisé R^Q muni de la fonction $eval_T$ est bien l'*exponentiel* de R par Q dans la catégorie $TPoset(T)$. Les fonctions synchrones peuvent ainsi être codées comme des objets temporisés qui sont transmis de façon synchrone d'un site à un autre, la fonction synchrone $eval_T$ permettant d'*appliquer à la volée* une fonction temporisée à son argument.

Lemme 5.13 (Sous-objet temporisés)**.** *Soit $f : P \to Q$ une fonction synchrone. Alors on a $f(P) = \downarrow f(P)$. De plus, f est injective si et seulement si $P \simeq f(P)$.*

Remarque 5.14. Dans le cas des ensembles, on dispose de $\mathcal{P}(E)$, l'ensembles des parties d'un ensemble E, pour représenter tous les sous-objets de E dans la catégorie Set. Dans la catégorie des fonctions synchrones, il existe une construction analogue qui complète d'une certaine façon l'ensemble des sous-objets de P afin de pouvoir le modéliser lui-même comme un ensemble temporisé.

Définition 5.15 (Produit synchrone généralisé)**.** Soit P un ensemble temporisé sur T. Soit $\Omega^P = \{(u, X) \in T \times \mathcal{P}(P) : X = \downarrow X, \pi(X) \subseteq \downarrow u\}$ le *produit synchrone généralisé* de P, équipé de la première projection comme projection temporelle, et de l'ordre partiel définit par $(u, X) \leq (v, Y)$ lorsque $u \leq v$ et $X = Y\downarrow u$ pour tout $(u, X), (v, Y) \in \Omega^P$.

Lemme 5.16. *Le produit synchrone généralisé Ω^P de P est un ensemble temporisé.*

Remarque 5.17. L'interprétation de ce produit synchrone généralisé en terme de comportements temporisés est assez naturelle : il permet de représenter tous les comportements temporisés qu'on peut définir en lançant de façon synchrone un nombre arbitraire de calculs (même conflictuels) dont la trace est dans P. Autrement dit, avec Ω^P tous se passe comme si on dispose d'un ensemble de processeurs sur lesquels on peut faire tourner de façon synchrone n'importe

quel ensemble de calculs modélisés par P. Bien entendu, à chaque instant, deux calculs « identiques », c'est à dire deux calculs ayant même trace, n'apparaissent qu'une fois [1].

Remarquons de plus que dans une trace $(u, X) \in \Omega^P$, tout élement $x \in X$ tel que $\pi(x) < u$ représente la trace d'un sous-calcul passé. Ce dernier peut même être terminé dès lors qu'il n'existe pas d'élement $y \in X$ tel que $x < y$. Par exemple, l'élement $(u, \downarrow x) \in \Omega^P$ avec $u > \pi(x)$ représente la trace d'un calcul définie par x, terminé, dans lequel la seule information qui continue de changer est le temps qui passe, modélisé par l'instant u. En programmation temporisées il est parfois délicat de décrire des calculs qui son terminés bien que le temps continue de s'écouler. Le produit synchronisé généralisé semble apporter, entre autre, une réponse formelle à cette problématique.

Exemple 5.18 (Signaux temporisés)**.** On peut définir l'ensemble temporisés des impulsions $Imp(T, E) = T \times (E \uplus \{*\})$ équipé de la première projection comme projection temporelle et de la relation d'ordre définie comme (la cloture réflexive de) la relation d'ordre stricte définie pour tout $(t, e), (t', e') \in Imp(T, E)$ par $(t, e) < (t', e')$ lorsque $t < t'$ et $e = *$. On constante alors que l'ensemble $Sig(T, E)$ est isomorphe à l'objet puissance $\Omega^{Imp(T,E)}$ construit à partir de l'ensemble des impulsions temporisés.

Remarque 5.19. On peut démontrer que Ω^P est bien l'*objet puissance* associé à P dans la catégorie $TPoset(T)$. Mais même l'énoncé de ce fait nous entrainerait bien trop loin dans la théorie des topos. Plus simplement, l'existence de produits synchrones généralisés implique l'existence d'un ensemble temporisé qui caractérise en un certain sens la forme des sous-objets via la généralisation de la notion de fonction caractéristique. C'est le *classificateur* des sous-objets d'une catégorie. Dans le cas des ensembles, ce classificateur est simplement défini comme l'ensemble booléen $\mathbb{B} = \{t, f\}$ qui est le co-domaine de toute fonction caractéristique d'un sous-ensemble. Dans le cas des ensembles temporisés, il peut être défini de la façon suivante.

Définition 5.20 (Classificateur de sous-objets)**.** Le classificateur des sous-objets de $TPoset(P)$ est défini comme l'ensemble temporisé $\Omega^T = \{(u, V) \in T \times \mathcal{P}(T) : V = \downarrow V, V \subseteq \downarrow u\}$.

Remarque 5.21. Avec la fonction synchrone $true_T : T \to \Omega^T$ définie par $true_T(t) = (t, \downarrow t)$ pour tout $t \in T$, on peut démontrer que pour tout ensemble temporisé P, pour toute partie $X \subseteq P$ close par le bas, c'est à dire tout sous-objet X de P, il existe une unique fonction synchrone $\mathcal{X}_X : P \to \Omega^T$ telle que, en notant $inc_X : X \to P$ l'inclusion (synchrone) de X dans P on a $\mathcal{X}_X \circ inc_X = true_T \circ \pi$ et le schéma commutatif résultant est un *produit fibré*, i.e. pour toute fonction synchrone $f : Q \to P$ telle que $\mathcal{X}_X \circ f = true_T \circ \pi$ on a nécessairement $f(Q) = X$. La fonction $\mathcal{X}_X : P \to \Omega^T$ est appelé la *fonction caractéristique* du sous-objet X de P. D'une certaine façon, Ω^T capture la *structure* des sous-ensembles temporisés dans $TPoset(T)$, tout comme l'ensemble booléen $\{t, f\}$ capture la structure des sous-ensembles dans Set.

Le théorème suivant résume les résultats précédents. Il découle aussi de l'équivalence de la catégorie $TPoset(T)$ avec la catégorie $Psh(T)$ des pré-faisceaux sur T.

Théorème 5.22. *Pour toute échelle de temps T, la catégorie $TPoset(T)$ est un topos. En particulier, elle est cartésienne close et possède toutes les puissances temporisées.*

Remarque 5.23. En théorie des catégories, les topoi sont les catégories qui se comportent « essentiellement » comme la catégorie Set. En particulier, la présence des classificateurs de sous-objets permet en quelque sorte de raisonner, en logique du premier ordre ou plus, à l'intérieur des

1. De la même manière qu'un ensemble n'est pas un multi-ensemble puisqu'il ne contient qu'une « copie » de chacun des éléments qui le compose. La construction d'un analogue des multi-ensembles dans le cas temporisés est sans doute possible (avec quelle pondération lors de la différenciation de plusieurs trace qui était identique jusque là ?) quand bien même elle n'est pas explicitée ici. Une extension probabiliste doit aussi être envisageable. . .

topos ce qui, dans le cas qui nous interesse, pourrait se révéler utile afin d'analyser les propriétés des comportements de programmes temporisés modélisés dans le formalisme proposé.

6 Domaines temporisés

Nous développons dans cette section la notion de domaines temporisés : des ensembles ordonnés complets et temporisés sur des échelles de temps complètes.

Définition 6.1 (Continuité de Scott)**.** Soit P un ensemble partiellement ordonné. Une partie $X \subseteq P$ est dites *dirigée* lorsqu'elle est non vide et, pour tout $x, y \in X$, il existe $z \in X$ tel que $x, y \leq z$. Etant alors donné Q un autre ensemble partiellement ordonnée, une fonction $f : P \to Q$ est *continue* lorsque, pour toute partie dirigée de $X \subseteq P$ si la borne sup. $\bigvee X$ existe dans P alors la borne sup. $\bigvee f(X)$ existe dans Q est on a $\bigvee f(X) = f(\bigvee X)$.

Un ensemble partiellement ordonnée P est (dirigé) *complet* lorsque toute partie $X \subseteq P$ dirigée admet une borne sup. $\bigvee X \in P$. De plus, en notant $x \ll y$ la relation définie pour tout $x, y \in P$ lorsque pour tout $Z \subseteq P$ dirigée telle que $y \leq \bigvee Z$ il existe $z \in Z$ telle que $x \leq z$, on dit que P est *continu* lorsque pour tout $x \in P$, l'ensemble $\{y \in P : y \ll x\}$ est dirigé avec x comme borne sup.

Remarque 6.2. Toute fonction continue $f : P \to Q$ est nécessairement croissante. En effet, si f est continue, pour tout $x, y \in P$ tel que $x \leq y$, l'ensemble $\{x, y\}$ et dirigé avec $\bigvee\{x, y\} = y$ est donc, par continuité, $\bigvee\{f(x), f(y)\} = f(y)$ c'est à dire $f(x) \leq f(y)$.

Définition 6.3 (Ensembles temporisés pré-continus)**.** Soit P un ensemble temporisé sur T. On dit que P est *ensemble temporisé pré-continu* lorsque

(IN3) si X est dirigé dans P et $\bigvee \pi(X)$ existe alors $\bigvee X$ existe aussi,

pour tout $X \subseteq P$.

Exemple 6.4 (Signaux temporisés)**.** Un signal partiel temporisé $(u, X) \subseteq Sig(T, E)$ est dit *observable* à l'instant u lorsque pour tout $(t, e) \in X$ on a $t \ll u$. Autrement, un signal (u, X) est observable lorsque pour tout « chemin dirigée » d'observations temporelles « conduisant » à l'instant u, chaque évènement de X est observable à au moins l'un de ces instants. L'ensemble $SigC(T, E)$ des signaux observables muni de la coupure définie pour tout $(u, X) \in SigC(T, E)$ et $v \leq u$ par $(u, X)\downarrow v = (v, \{(t, e) \in X : t \ll v\})$ est un ensemble temporisés pré-continus. On vérifie que $\bigvee S = (\bigvee \pi(S), \bigcup \pi_2(S))$ avec $\pi_2(S) = \{X \subseteq T \times E : u \in T, (u, X) \in S\})$ pour toute partie $S \subseteq SigC(T, E)$ dirigée telle que $\bigvee \pi(S)$ existe.

Lemme 6.5. *Soit P un ensemble temporisé pré-continu sur T. Alors, pour tout $x \in P$, la coupure locale $cut_x : \downarrow \pi(x) \to P$ est continue.*

Remarque 6.6. En général, la continuité des coupures locales sur un ensemble temporisé n'implique pas la pré-continuité de l'ensemble temporisé lui-même comme le montre l'exemple $P_1 = \mathbb{N}$ temporisé sur $T_1 = \overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$ par la fonction d'inclusion de P_1 dans T_1. De même, la pré-continuité d'un ensemble temporisé n'implique pas la continuité de la projection temporelle comme le montre l'exemple $P_2 = \overline{\mathbb{N}}$ temporisé sur $T_2 = \overline{\mathbb{N}} \cup \{\infty'\}$, là encore par la fonction d'inclusion, avec ∞' un second majorant de $\mathbb{N}$ incomparable avec ∞.

Définition 6.7 (Domaine temporisé)**.** Un *domaine temporisé* est un ensemble temporisé, pré-continu, sur une échelle de temps T complète.

Théorème 6.8. *Soit P un domaine temporisé sur une échelle de temps (complète) T. Alors P est un ensemble partiellement ordonné complet et sa projection temporelle $\pi : P \to T$ est continue.*

Exemple 6.9. Lorsque T est complet, l'ensemble $SigC(T, E)$ des signaux temporisés observables 6.4 est donc un domaine temporisé complet. On peut montrer que si T est de plus continu au sens de Scott [21] alors $SigC(T, E)$ l'est aussi. Fait remarquable, cette condition sur T ne suffit pas, loin s'en faut, à faire de l'ensembles de tous les signaux temporisés $Sig(T, E)$ un domaine temporisé. La condition d'observabilité, apparement technique, semble ici essentielle.

Lemme 6.10 (Δ-synchrone vs continue)**.** *Soit P et, resp. Q, deux domaines temporisés sur U et, resp., V (dirigées) complets. Soit $\delta : U \to V$ une fonction continue. Soit $f : P \to Q$ une fonction Δ-synchrone de projection temporelle δ. Alors f est* continue.

Remarque 6.11. En particulier, toute fonction synchrones entre deux domaines temporisés est continue.

Théorème 6.12. *La categorie $TCpo(T)$ des domaines temporisés sur un cpo T et des fonctions synchrones (et continues) entre ces domaines temporisés est un topos.*

Eléments de preuve. La preuve, courte mais technique, est esquissée ici. Elle s'appuie sur l'équivalence de $TPoset(T)$ avec $Psh(T)$ et passe par la définition d'une topologie de Grothendieck J sur l'échelle de temps T qui caractérise les (pré-faisceaux représentant les) ensembles temporisés pré-continus. Techniquement, pour tout $t \in T$, cette topologie est définie par la famille de cribles $J(t)$ de toutes les parties $X \subseteq T$ majorée par t, close par le bas et dont la clôture de Scott est égale à $\downarrow t$. Pour mémoire, la clôture de Scott de X est, dans le cas où T est complet et X clos par le bas, l'ensemble de tous les sup. $\bigvee Y$ pour $Y \subseteq X$ dirigé. □

Remarque 6.13. Etant donnée la catégorie *Cpo* des ensembles partiellement ordonnés dirigés complets et des fonctions continues, on peut aussi considérer la catégorie $TCpo(Cpo)$ des domaines temporisés et des fonctions Δ-synchrones de projections temporelles continues. On montre facilement, tout comme dans le cas de $TPoset(Poset)$, que $TCpo(Cpo)$ est cartésienne close puisque *Cpo* l'est aussi.

7 Points-fixes dans les domaines temporisés avec minimums

Pour conclure notre étude des domaine temporisé, nous examinons maintenant les plus petit point-fixe des fonctions d'un domaine temporisé dans lui-même qui apparaissent dans sémantique aux définitions récursives de fonctions temporisées.

Cas général. Etant donnée une fonction monotone $f : P \to P$ on s'interesse donc au calcul du plus petit point-fixe de f, c'est à dire à l'élement $\mu(f) \in P$ qui, s'il existe, est telle que $f(\mu(f)) = \mu(f)$ et pour tout $x \in P$ tel que $x = f(x)$ on a $\mu(f) \leq x$. La théorie des domaines de Scott [21] nous garantie que si P est dirigé complet avec un plus petit élement $\bot \in P$ et si f est continue, alors on a $\mu(f) = \bigvee_{n \in \omega} f^n(\bot)$.

Soit $Cpo_\bot$ la catégorie des ensembles dirigées complets avec minimum et des fonctions continues (mais pas nécessairement strictes) entre ces ensembles. Soit $TCpo_\bot(Cpo_\bot)$ la sous-catégorie de $TCpo(Cpo)$ des domaines temporisés sur ces cpos avec minimum qui admettent eux-même un élement minimum $\bot$. Rappelons qu'on a nécessairement $\pi(\bot) = \bot$, en notant les minimaux de façon polymorphes.

Remarque 7.1. On vérifie aussi facilement que est cartésienne close avec l'objet terminal $\{\bot\}$ auto-temporisé, le produit et l'exponentiel étant définit tout comme dans $TPoset(Poset)$. Plus généralement, on peut montrer [15] qu'un foncteur $F : \mathcal{C} \to TCpo_\bot(Cpo_\bot)$ admet une limite (resp. une co-limite) dès lors que sa projection sur $Cpo_\bot$ admet une limite (resp. une co-limite).

Théorème 7.2. *Soit P un domaine temporisé sur T avec minimums respectifs $\bot_P$ et $\bot_T$, et T dirigé complet. Pour toute fonction Δ-synchrone de projection temporelle $\delta : T \to T$ continue, on a :*

$$\mu_P(f) = \bigvee_{n\in\omega} f^n(\bot_P) \quad et \quad \pi \circ \mu_P(f) = \mu_T(\delta) = \bigvee_{n\in\omega} \delta^n(\bot_T)$$

De plus, en notant P^P l'exponentiel de P par P dans la catégorie $TCpo_\bot(Cpo_\bot)$, c'est à dire l'ensemble des fonctions Δ-synchrones de P dans P, dont les projections temporelles sont des fonctions continues de T dans T continues, la fonction $\mu_P : P^P \to P$ est Δ-synchrone de projection temporelle $\mu_T : T^T \to T$.

Remarque 7.3. Autrement dit, le calcul du plus petit point-fixe d'une fonction peut être effectué à la volée, tout comme l'application de fonctions.

Cas des semi-treillis. Nous avons observé que les fonctions Δ-synchrones ne préservent pas forcément la cohérence. Vu le lemme 4.14 nous pouvons nous restreindre à la catégorie $TCpo_\bot(CSLat_\bot)$ des domaines avec minimum temporisés sur des semi-treillis continues, c'est à dire avec la borne inf. continue, et les fonctions Δ-synchrones de projection temporelle qui préservent ces inf.

On constate cependant que, restreint à la catégorie $TCpo_\bot(CSLat_\bot)$, le théorème 7.2 n'est plus satisfait. En effet, rien ne permet d'assurer que la fonction $\mu_T : T^T \to T$ préservent les inf.. Cette remarque nous conduit à considerer la classe, plus restreinte, des fonctions causales (voir 4.4).

La propriété de causalité semble tout d'abord être une restriction naturelle afin de mettre en place une sorte de *retro-action* qui nous permet effectivement de calculer les itérations $f^n(\bot)$ d'une fonction $f : P \to P$ avec P temporisé sur T afin d'en calculer le plus petit point-fixe $\mu_P(f)$. En effet, notre connaissance des sorties parties d'une fonction doit être postérieure à celle des entrées afin de pouvoir les réinjecter de façon temporellement cohérente. Le lemme suivant montre qu'on a plus encore.

Lemme 7.4. *Dans la catégorie $TCpo_\bot(CSLat_\bot)$, en notant $[P \to_C P]$ les fonctions Δ-synchrones causales de P dans lui-même, en notant $[T \to_C T]$ les fonctions auto-synchrones causales de T dans lui-même, la restriction de l'opérateur de point-fixe $\mu_P : [P \to_C P] \to P$ est bien une fonction Δ-synchrone dont la projection $\mu_T : [T \to_C T] \to T$ préserve les inf.*

Remarque 7.5. Autrement dit, de façon quelque peu inattendue, la notion de causalité, qui provient de l'interprétation temporelle qu'on peut avoir des fonctions Δ-synchrones d'un domaine temporisé sur lui-même, trouve ici une raison d'être mathématique : elle garantie que les calculs de points-fixes à la volée vont rester cohérents avec cette interprétation temporelle.

8 Conclusion

Nous avons proposé dans ces pages une extension temporisée de la catégorie $Cpo_\bot$ des domaines de Scott, pris ici au sens le plus général. Il s'agit bien d'une extension puisque la catégorie $Cpo_\bot$ est bien la sous-catégorie de $TCpo_\bot(Cpo_\bot)$ définie par les horloges (les ensembles auto-temporisés) et de leurs transformations continues. Nous avons aussi montré comment chaque

constructeur catégorique a une interprétation naturelle en termes de sémantique temporisée ; une interprétation synchrone sur les topos $TCpo_{\perp}(T) \subseteq TCpo_{\perp}(Cpo_{\perp})$, une interprétation asynchrone sur la catégorie $TCpo_{\perp}(Cpo_{\perp})$ elle-même. Dans la catégorie $TCpo_{\perp}(Cpo_{\perp})$ l'opérateur $\mu_P : P^P \to P$ de calcul de plus petit point-fixe des fonctions Δ-synchrones homogènes est lui-même un morphisme de cette catégorie avec comme projection temporelle $\mu_T : T^T \to T$. De plus, toute définition inductive (ou co-inductive) de domain temporisé admet une solution dès lors que sa projection temporelle sur $Cpo_{\perp}$ en admet une.

Cette extension des domaines aux cas temporisés explicite et généralise l'approche proposée par Colaço, Pouzet et al. [8, 9] qui proposent déjà un polymorphisme d'horloges et de l'ordre supérieure : les fonctions synchrones sont elles-mêmes traitées comme des objets temporisés pouvant être passés en paramètres de fonctions synchrones d'ordre supérieur. Les résultats cités ici, issues des travaux de recherche sur les langages synchrones [3, 4, 5], avaient déjà permis d'observer l'intérêt qu'on pouvait avoir à distinguer les échelles de temps (ou horloges) à l'entrée et à la sortie des programmes temporisés : une distinction que nous reprenons ici à notre compte. Comme dans [8], les transformations de ces échelles de temps participent aux types des fonctions temporisées.

On l'a vue, le choix d'une échelle de temps indique quelle est la structure spatio-temporelle des données et calculs modélisés. Les échelles de temps n'étant pas totalement ordonnées, on peut donc penser que le formalisme proposé ici est aussi applicable à la modélisation des types de sessions (voir par exemple [22]) qui fixent en quelque sorte les scénarios et protocoles de communications qui se dérouleront entre deux processus. Les liens possibles avec ce typage « concurrent » restent à développer. Il pourrait aussi être possible de revisiter à l'aide de la notion de domaine temporisé la sémantiques des propositions de langages fonctionnels réactifs existantes, tel que, notamment, ReactiveML [19] ou FRP [11, 12].

Remarquons enfin que toute sémantique temporisée porte en elle une sémantique opérationnelles puisqu'elle indique aussi « quand » les calculs sont effectués. Les liens possibles de notre propositions avec les formalismes de sémantiques opérationnelles issues, notamment, de la théorie des automates d'entrée-sorties temporisés [16] devront être étudiés.

Remerciements. L'auteur tient à remercier ici Gordon Plotkin et Phil Scott pour leur suggestion commune d'explorer la théorie des pré-faisceaux, Marek Zawadowski pour son explication personnalisées des notions de faisceaux et de topologies de Grothendieck, Simon Archipoff, Michail Raskin et Bernard Serpette pour de nombreuses discussions sur les sujets abordés ici, et les rapporteurs des JFLA pour leurs précieux commentaires.

Références

[1] G. Berry. Stable models of typed lambda-calculi. In *Int. Col. on Aut., Lang. and Programming (ICALP)*, volume 62 of *LNCS*, pages 72–89. Springer, 1978.

[2] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theor. Comp. Sci.*, 20 :265–321, 1982.

[3] G. Berry and G. Gonthier. The Esterel synchronous programming language : design, semantics, implementation. *Science of Computer Programming*, 19(2) :87 – 152, 1992.

[4] P. Caspi and M. Pouzet. Synchronous Kahn networks. In *Int. Conf. Func. Prog. (ICFP)*, pages 226–238, 1996.

[5] P. Caspi and M. Pouzet. A co-iterative characterization of synchronous stream functions. *Electr. Notes Theor. Comput. Sci.*, 11 :1–21, 1998.

[6] G. L. Cattani, I.Stark, and G. Winskel. Presheaf models for the π-calculus. In *Category Theory and Computer Science (CTCT)*. Springer, 1997.

[7] G. L. Cattani and G. Winskel. Presheaf models for CCS-like languages. *Theor. Comp. Sci.*, 300(1) :47 – 89, 2003.

[8] J.-L. Colaço, A. Girault, G. Hamon, and M. Pouzet. Towards a higher-order synchronous data-flow language. In *Int. Conf. On Embedded Software (EMSOFT)*, pages 230–239. ACM, 2004.

[9] J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In *Int. Conf. On Embedded Software (EMSOFT)*, pages 134–155. ACM, 2003.

[10] P. Cousot, R. Cousot, and L. Mauborgne. Logical abstract domains and interpretations. In S. Nanz, editor, *The Future of Software Engineering*, pages 48–71. Springer-Verlag, 2010.

[11] C. Elliott and P. Hudak. Functional reactive animation. In *Int. Conf. Func. Prog. (ICFP)*. ACM, 1997.

[12] C. M. Elliott. Push-pull functional reactive programming. In *Symp. on Haskell*, pages 25–36. ACM, 2009.

[13] J.-Y. Girard. Linear logic. *Theor. Comp. Sci.*, 50 :1–102, 1987.

[14] J. Hughes. Programming with arrows. In *Advanced Functional Programming (AFP)*, volume 3622 of *LNCS*, pages 73–129. Springer, 2005.

[15] D. Janin. Spatio-temporal domains : an overview. Technical report, LaBRI - Université de Bordeaux, 2017.

[16] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Morgan & Claypool Publishers, 2006.

[17] X. Liu and E. A. Lee. CPO semantics of timed interactive actor networks. *Theoretical Computer Science*, 409(1) :110 – 125, 2008.

[18] X. Liu, E. Matsikoudis, and E. A. Lee. Modeling timed concurrent systems using generalized ultrametrics. In *Conf. on Conc. Theory (CONCUR)*, volume 4137 of *LNCS*. Springer, 2006.

[19] L. Mandel and M. Pouzet. ReactiveML, a reactive extension to ML. In *Int. Symp. on Principles and Practice of Declarative Programming (PPDP)*. ACM, 2005.

[20] E. Matsikoudis and E. A. Lee. The fixed-point theory of strictly causal functions. *Theor. Comp. Sci.*, 574 :39–77, 2015.

[21] D. S. Scott. Data types as lattices. In *International Summer Institute and Logic Colloquium, Kiel*, volume 499 of *LNM*, pages 579–651. Springer, 1975.

[22] P. Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3) :384–418, 2014.

[23] G. Winskel. Events, causality and symmetry. In *International Conference on Visions of Computer Science*, pages 111–128. BCS Learning & Development Ltd., 2008.

HOcore en HOcore

Lionel Zoubritzky[1] and Alan Schmitt[2]

[1] ENS Paris
[2] Inria

Résumé

Nous présentons une machine abstraite pour HOcore, un calcul de processus d'ordre supérieur minimal. Nous prouvons la correction et la complétude de cette machine. Finalement, nous la raffinons en deux implémentations : une en OCaml et une en HOcore lui-même.

1 Introduction

Une étape intermédiaire classique entre la sémantique formelle d'un système et son implémentation est la définition d'une *machine abstraite*. Ces machines permettent de spécifier des aspects importants de l'implémentation d'un langage, comme la stratégie d'évaluation, tout en restant à un niveau d'abstraction suffisant pour non seulement pouvoir montrer leur correction, mais également pour autoriser plusieurs implémentations dans des langages très différents. Diehl, Hartel et Sestoft [5] présentent de nombreux exemples de machines abstraites.

Dans le cadre des langages concurrents, un des objectifs est de pouvoir simuler de façon réaliste la distribution, c'est-à-dire la capacité pour plusieurs unités de calcul indépendantes de pouvoir effectuer des tâches en interne, ou de les communiquer sur un réseau. Pour ce faire, plusieurs systèmes formels ont été créés pour les modéliser, qu'on appelle calculs de processus : ils sont en majorité inspirés du Calcul de Système de Communication (ou CCS, on pourra se référer à [13] pour une introduction complète sur ce calcul). Tous disposent d'un moyen d'émettre et de recevoir des messages, ce qui, avec une certaine notion de parallélisme, forme le cœur des calculs de processus.

Il existe une machine abstraite très générale pour les calculs de processus, la *Chemical Abstract Machine* [1] qui modélise ces émissions et réceptions de messages par des constituants d'une réaction chimique. Sa généralité lui permet de modéliser plusieurs calculs comme le π-calcul [6] ou le *Distributed Join-Calculus* [7]. Cependant, sa nature est très éloignée de l'implémentation qui est faite de ces calculs. D'autres machines abstraites existent, chacune spécifique à un calcul comme [2] pour le π-calcul, [16] pour *Safe Ambient* ou [8] pour le M-calcul par exemple.

Le M-calcul est en particulier un calcul d'ordre supérieur, c'est-à-dire que les messages transmis sont eux-mêmes des processus de M-calcul. Ceci permet de modéliser le transfert de code exécutable. Le π-calcul possède aussi un pendant d'ordre supérieur, HOπ [15]. HOπ ne dispose cependant pas encore de machine abstraite, alors qu'il s'agit d'un calcul de processus d'ordre supérieur de référence.

Notre travail se situe dans le cadre du développement d'une machine abstraite pour HOπ. Comme première étape, nous concevons une machine abstraite pour HOcore [11], un calcul de processus d'ordre supérieur minimal, inspiré de HOπ mais simplifié de façon à ne contenir que les constructions nécessaires à un calcul de processus d'ordre supérieur. Nous établissons ensuite une preuve que la machine abstraite simule de façon adéquate le calcul, selon des notions de correction et de complétude développées en 3.2.2. [1] Enfin, nous présentons deux implémentations

1. La preuve complète est disponible à l'adresse :
http://people.rennes.inria.fr/Alan.Schmitt/research/hocore/AbstractMachineHOCoreProofs.pdf

de la machine, une en OCaml afin de visualiser son action, et l'autre en HOcore afin de mieux comprendre ce calcul et pour montrer son expressivité. L'objectif à long terme est de pouvoir simplement fournir des implémentations pour des extensions de HOcore.

2 HOcore

Le calcul HOcore est un calcul de processus d'ordre supérieur : les messages échangés contiennent des processus, et non pas des noms comme dans le π-calcul. On peut le considérer comme un sous-calcul de HOπ, auquel on aurait enlevé la restriction de nom.

2.1 Définition

2.1.1 Syntaxe

La syntaxe d'un processus HOcore est définie par récurrence avec

$$P ::= \quad P \parallel P \mid \bar{a}\langle P \rangle \mid a(x).P \mid x \mid \star$$

— $P \parallel Q$ désigne la composition en parallèle des deux processus P et Q ;
— $\bar{a}\langle P \rangle$ désigne l'émission du processus P sur le canal a. Comme le message émis est lui-même un processus, le calcul est d'ordre supérieur ;
— $a(x).P$ désigne la réception d'un message sur le canal a avec pour continuation le processus P, dans lequel x désigne le message reçu.
— x est une variable ;
— $\star$ désigne le processus vide, qui n'agit pas.

L'opération $\parallel$ est commutative, associative et possède $\star$ comme élément neutre.

2.1.2 Réduction

HOcore dispose d'une unique règle de réduction, consistant en la communication d'un message entre une émission et une réception.

Formellement, cette règle s'écrit

$$\bar{a}\langle Q \rangle \parallel a(x).R \parallel P \;\rightarrow\; R\{x \leftarrow Q\} \parallel P$$

Le terme $R\{x \leftarrow Q\}$ correspond à R dans lequel toutes les occurrences libres de x ont été remplacées par Q.

Exemples :
— $\star$ n'admet pas de réduction.
— $\bar{a}\langle y \rangle \parallel a(x).x \rightarrow y$.
— Pour $P = \bar{a}\langle y \rangle \parallel \bar{a}\langle z \rangle \parallel a(x).x$, il y a deux réductions possibles : $P \rightarrow \bar{a}\langle y \rangle \parallel z$ et $P \rightarrow \bar{a}\langle z \rangle \parallel y$.
— Pour $\Omega = \bar{a}\langle a(x).(x \parallel \bar{a}\langle x \rangle) \rangle \parallel a(x).(x \parallel \bar{a}\langle x \rangle)$, on a : $\Omega \rightarrow \Omega$.

Nous considérons la réduction comme définition de la sémantique des processus. Cette notion permet de librement utiliser l'équivalence structurelle (commutativité et associativité de la composition parallèle) et est intuitivement plus simple. Elle permet également de réorganiser le processus évalué pour obtenir une représentation plus efficace, comme présenté en section 4.1.1. Une approche alternative est d'utiliser des *transitions étiquetées* qui conservent la structure

syntaxique du processus, ce qui est pratique pour leur manipulation avec des assistants de preuves. Les deux approches sont équivalentes, et nous préférons la première, plus intuitive, pour ce travail.

2.1.3 Indices de de Bruijn

La syntaxe initiale est sujette au problème de l'α-renommage, c'est-à-dire que, quitte à renommer entièrement les variables, un même processus peut s'écrire de plusieurs façons différentes. Ainsi, $a(x).x$ et $a(y).y$ désignent deux réceptions équivalentes, mais qui ne sont pas écrites de la même façon.

Pour y remédier, nous utilisons les variables de de Bruijn [4], c'est-à-dire qu'une variable ne sera plus nommée, mais sera représentée par le nombre de réceptions intermédiaires de la forme $a(x)$ se trouvant entre sa déclaration et sa position. Formellement, la syntaxe devient

$$P ::= \quad P \parallel P \mid \bar{a}\langle P\rangle \mid a.P \mid i \mid \star$$

avec i un indice de de Bruijn, c'est-à-dire un entier.

Exemples :

- $\star$ s'écrit de la même façon dans les deux syntaxes.
- $a(x).x$ se réécrit en $a.0$ car il n'y a aucun liant entre la position du x et sa déclaration dans $a(x)$.
- $a(x).b(y).x$ se réécrit $a.b.1$ car il y a un seul liant, $b(y)$, entre x et sa déclaration dans $a(x)$.
- $\Omega = \bar{a}\langle a(x).(x \parallel \bar{a}\langle x\rangle)\rangle \parallel a(x).(x \parallel \bar{a}\langle x\rangle)$ se réécrit $\bar{a}\langle a.(0 \parallel \bar{a}\langle 0\rangle)\rangle \parallel a.(0 \parallel \bar{a}\langle 0\rangle)$.

Un dernier problème se pose, pour traiter le cas des variables libres, c'est-à-dire des variables qui n'ont jamais été déclarées par un liant. Pour simplifier les opérations sur la machine, nous avons décidé d'utiliser la convention localement anonyme [3], consistant à dire qu'une variable liée est représentée par son indice de de Bruijn, tandis qu'une variable libre est nommée. La syntaxe utilisée est donc finalement

$$P ::= \quad P \parallel P \mid \bar{a}\langle P\rangle \mid a.P \mid i \mid x \mid \star$$

où x désigne une variable libre. Par exemple, $\bar{a}\langle y\rangle \parallel a(x).x$ se réécrit en $\bar{a}\langle y\rangle \parallel a.0$.

2.2 Propriétés du calcul

HOcore est un calcul non déterministe : un terme peut avoir plusieurs réductions différentes possibles en même temps. Il n'est pas confluent, c'est-à-dire que si un processus P admet deux réductions distinctes ayant pour résultat P_1 et P_2, il est possible que P_1 et P_2 ne puissent pas se réduire en un nombre quelconque d'étapes vers un même processus Q. C'est par exemple le cas pour $P = \bar{a}\langle y\rangle \parallel \bar{a}\langle z\rangle \parallel a(x).x$ avec $P_1 = \bar{a}\langle y\rangle \parallel z$ et $P_2 = \bar{a}\langle z\rangle \parallel y$. Cette propriété est similaire à la notion de course critique dans un programme concurrent, quand plusieurs processus légers écrivent simultanément dans la même case mémoire : deux exécutions d'un même processus peuvent ne pas avoir la même issue en fonction de l'entrelacement des sous-processus mis en parallèle.

La différence principale entre HOcore et HOπ dont il est issu est l'absence d'opérateur de restriction de nom. En effet, de nombreux calculs de processus disposent d'un moyen plus ou

moins direct de créer de nouveaux noms de canaux ou de variables, ce qui permet d'effectuer en parallèle un nombre arbitraire de tâches. Les opérations de ce genre sont cependant impossibles en HOcore. En particulier, le nombre de canaux différents qui peuvent exister au cours de l'exécution d'un processus est toujours majoré par le nombre de canaux initiaux.

Cette contrainte fait que HOcore dispose d'une expressivité moindre que beaucoup d'autres calculs. Il reste cependant Turing-complet [11].

3 Machine abstraite

3.1 De la machine de Krivine à HOcore

3.1.1 Machine de Krivine

Une machine abstraite classique est la machine de Krivine [10], qui sert à modéliser le λ-calcul. Elle a servi d'inspiration pour construire celle pour HOcore. Pour rappel, la syntaxe du λ-calcul en utilisant les indices de de Bruijn est la suivante :

$$u ::= \quad \lambda.u \mid u\,u \mid i$$

La machine de Krivine est alors définie comme un triplet formé d'un λ-terme, d'une pile π et d'un environnement e. La pile et l'environnement sont des listes dont les éléments sont définis récursivement comme étant des paires formée d'un λ-terme et d'un environnement. La pile contient les prochains termes à évaluer et l'environnement, les arguments successifs des λ-abstractions.

La machine possède plusieurs réductions possibles, en fonction de la forme du λ-terme :

$$\langle u\,v, \pi, e\rangle \to \langle u, (v, e) :: \pi, e\rangle \qquad \langle \lambda.u, p :: \pi, e\rangle \to \langle u, \pi, p :: e\rangle$$
$$\langle i+1, \pi, p :: e\rangle \to \langle i, \pi, e\rangle \qquad \langle 0, \pi, (u, e') :: e\rangle \to \langle u, \pi, e'\rangle$$

La machine de Krivine est déterministe puisqu'à un λ-terme ne peut correspondre qu'une seule réduction. Cela vient du fait qu'elle simule une stratégie d'évaluation particulière, l'appel par nom (ou stratégie d'évaluation externe gauche) : dans un terme de la forme $(\lambda.u)v$, c'est d'abord u qui est évalué puis v. Cela se justifie car le λ-calcul est confluent, et si un terme dispose d'une forme normale (c'est-à-dire s'il peut se réduire en un terme qui ne se réduit plus ensuite), alors cette stratégie d'évaluation mène à cette unique forme normale.

3.1.2 Machine abstraite pour HOcore

Contrairement au λ-calcul, HOcore n'est pas un calcul confluent. Nous avons donc choisi de ne pas implémenter de stratégie d'évaluation spécifique dans la machine abstraite. Par conséquent, celle-ci est non déterministe, puisqu'elle doit pouvoir simuler n'importe quelle réduction du processus HOcore initial.

La machine est inspirée de celle de Krivine par l'utilisation d'environnements e, récursivement définis comme des listes de paires formées d'un processus HOcore et d'un environnement. Le i-ème élément d'un environnement e, noté $e\,[i]$ est donc de la forme (P, e). Nous appelons de telles paires des *processus annotés*. La syntaxe de la machine est :

$$\mathbf{M} ::= \mathbf{M} + \mathbf{M} \mid (\bar{a}\,\langle P\rangle\,, e) \mid (a.P, e) \mid x \mid \star$$

— $+$ est le pendant de $\parallel$ pour les machines. Il est de même commutatif, associatif et admet $\star$ comme élément neutre ;
— $(\bar{a}\langle P\rangle, e)$ et $(a.P, e)$ sont des processus annotés d'un environnement, respectivement une émission et une réception ;
— x désigne une variable libre ;
— $\star$ est la machine vide, qui ne peut réaliser aucune action.

La définition de transitions pour une machine s'appuie sur une traduction des processus annotés (P, e) en machine, notée $[\![(P, e)]\!]_{\mathcal{M}}$ et récursivement définie comme suit.

$$\begin{aligned} [\![(P \parallel Q, e)]\!]_{\mathcal{M}} &= [\![(P, e)]\!]_{\mathcal{M}} + [\![(Q, e)]\!]_{\mathcal{M}} & [\![(\star, e)]\!]_{\mathcal{M}} &= \star \\ [\![(\bar{a}\langle P\rangle, e)]\!]_{\mathcal{M}} &= (\bar{a}\langle P\rangle, e) & [\![(x, e)]\!]_{\mathcal{M}} &= x \\ [\![(a.P, e)]\!]_{\mathcal{M}} &= (a.P, e) & [\![(i, e)]\!]_{\mathcal{M}} &= [\![e\,[i]]\!]_{\mathcal{M}} \end{aligned}$$

La transition d'une machine est alors

$$(\bar{a}\langle Q\rangle, e) + (a.R, f) + \mathbf{M} \;\to\; [\![(R, (Q, e) :: f)]\!]_{\mathcal{M}} + \mathbf{M}$$

Ceci signifie que le message émis est simplement mis dans l'environnement de la réception, sauf dans le cas où la réception est une variable liée, auquel cas sa valeur est évaluée.

Cette transition n'est pas tout à fait élémentaire. En effet, l'évaluation récursive de $[\![e\,[i]]\!]_{\mathcal{M}}$ dans le cas d'une variable n'est pas immédiate : on aurait pu définir à la place $[\![(i, e)]\!]_{\mathcal{M}} = (i, e)$ où (i, e) serait une nouvelle construction possible pour la machine, et rajouter la transition spontanée $(i, e) + \mathbf{M} \;\to\; [\![e\,[i]]\!]_{\mathcal{M}} + \mathbf{M}$. Les réductions possibles de cette alternative sont plus élémentaires, et on peut démontrer que toute transition de la première forme correspond à une série de réductions de la deuxième.

L'intérêt de la transition retenue par rapport à cette alternative est qu'une réduction du processus modélisé correspond exactement à une transition de la machine correspondante, ce qui simplifie grandement les preuves.

Exemples

— $\star$ n'admet pas de transition.
— $(\bar{a}\langle y\rangle, [\,]) + (a.0, [\,]) \;\to\; [\![(0, (y, [\,]) :: [\,])]\!]_{\mathcal{M}} = y$.
— $(\bar{a}\langle 0\rangle, [(x, [\,])]) + \big(a.(1 \parallel \bar{b}\langle\star\rangle), [(y, [\,])]\big) \;\to\; y + \big(\bar{b}\langle\star\rangle, [(x, [\,]) ; (y, [\,])]\big)$.

3.1.3 Bonne formation

L'approche localement anonyme permet l'existence de termes qui sont mal formés, si une variable de de Bruijn n'est pas liée. Par exemple, dans le processus $\bar{a}\langle 0\rangle$, la variable 0 est en réalité libre, et devrait donc être nommée et non pas écrite comme un indice de de Bruijn.

La correction de la machine est donc soumise à la contrainte que tous les indices de de Bruijn du processus initial se réfèrent à des liants valides, propriété dite de bonne formation. On vérifie simplement que si P est bien formé et si $P \to Q$, alors Q est aussi bien formé.

Une condition équivalente a été développée sur la machine de façon à ce que si $\mathbf{M} \to \mathbf{N}$ et $\mathbf{M}$ est bien formée, alors $\mathbf{N}$ aussi.

3.2 Correction et complétude

3.2.1 Traduction

L'essentiel de la preuve consiste à montrer qu'il existe une équivalence entre les transitions de la machine et les réductions des processus. Pour montrer ceci, il est nécessaire de disposer d'une traduction entre les machines et les processus.

Un processus P est simplement traduit en une machine abstraite qui le représente et notée $[\![P]\!]_{\mathcal{M}}$, définie par $[\![P]\!]_{\mathcal{M}} = [\![(P, [\,])]\!]_{\mathcal{M}}$.

La traduction inverse est plus complexe. Elle consiste à remplacer toutes les variables liées à une réception qui a déjà reçu un message par le message lui-même, que l'on peut trouver dans l'environnement. Pour définir formellement cette traduction inverse, on a besoin d'utiliser un paramètre annexe, la profondeur, qui sert à compter le nombre de liants sous lequel le terme à traduire est placé, afin de distinguer les variables liées par le terme des variables représentant un processus dans l'environnement. On définit la traduction partielle d'un processus annoté (P, e) avec profondeur d, notée $[\![(P, e)]\!]^d_{\mathcal{P}}$, définie par

$$[\![(P \parallel Q, e)]\!]^d_{\mathcal{P}} = [\![(P, e)]\!]^d_{\mathcal{P}} \parallel [\![(Q, e)]\!]^d_{\mathcal{P}} \qquad [\![(\star, e)]\!]^d_{\mathcal{P}} = \star$$

$$[\![(\bar{a}\langle P\rangle, e)]\!]^d_{\mathcal{P}} = \bar{a}\left\langle [\![(P, e)]\!]^d_{\mathcal{P}} \right\rangle \qquad [\![(x, e)]\!]^d_{\mathcal{P}} = x$$

$$[\![(a.P, e)]\!]^d_{\mathcal{P}} = a.\left([\![(P, e)]\!]^{d+1}_{\mathcal{P}}\right) \qquad [\![(i, e)]\!]^d_{\mathcal{P}} = \begin{cases} i & \text{si } i < d \\ [\![e\,[i-d]]\!]^0_{\mathcal{P}} & \text{sinon} \end{cases}$$

On peut alors définir la traduction inverse d'une machine, notée $[\![\mathbf{M}]\!]_{\mathcal{P}}$, simplement par

$$[\![\mathbf{M} + \mathbf{N}]\!]_{\mathcal{P}} = [\![\mathbf{M}]\!]_{\mathcal{P}} \parallel [\![\mathbf{N}]\!]_{\mathcal{P}} \qquad [\![(P, e)]\!]_{\mathcal{P}} = [\![(P, e)]\!]^0_{\mathcal{P}} \qquad [\![x]\!]_{\mathcal{P}} = x \qquad [\![\star]\!]_{\mathcal{P}} = \star$$

Une première propriété de ces traductions est que pour tout processus P bien formé, on a $[\![[\![P]\!]_{\mathcal{M}}]\!]_{\mathcal{P}} = P$.

L'autre égalité, $[\![[\![\mathbf{M}]\!]_{\mathcal{P}}]\!]_{\mathcal{M}} = \mathbf{M}$, est en revanche fausse dans le cas général d'une machine $\mathbf{M}$ bien formée. En effet, tous les environnements apparaissant dans la traduction directe $[\![P]\!]_{\mathcal{M}}$ sont vides, ce qui n'est pas nécessairement le cas chez la machine initiale $\mathbf{M}$.

Ce phénomène est une conséquence du fait qu'on peut obtenir un même processus HOcore à travers des suites de réductions différentes en partant de termes différents. En effet, la machine garde dans ses environnements la trace des différentes communications qui ont eu lieu, c'est-à-dire des réductions successives qui se sont produites. L'opération $[\![\cdot]\!]_{\mathcal{P}}$ applique ces substitutions retardées, et on peut considérer que $[\![[\![\cdot]\!]_{\mathcal{P}}]\!]_{\mathcal{M}}$ est une fonction de *nettoyage* de machines qui donne une machine équivalente dont tous les environnements sont vides (cette notion est utilisée à la section 3.2.3).

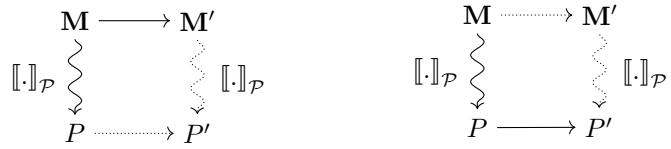
3.2.2 Propriétés requises

Deux propriétés sont requises pour la machine : la correction et la complétude.

La correction de la machine signifie que toute suite de transitions partant d'une machine bien formée correspond à une suite de réductions du processus en lequel elle est traduite : la machine simule le processus de manière correcte.

La complétude est la propriété duale, consistant à dire que toute suite de réductions d'un processus bien formé correspond à une suite de transitions des machines duquel il est traduit : la machine simule complètement le processus.

Ces deux propriétés se visualisent plus facilement à l'aide de diagrammes :



Sur ces diagrammes, les flèches pleines sont les hypothèses et celles en pointillées, les conséquences. Les flèches horizontales désignent généralement un nombre quelconque de réductions. Dans notre cas, une transition au sein des machines correspond exactement à une réduction pour les processus.

3.2.3 Résumé de la preuve

La preuve de la correction et de la complétude de la machine abstraite passe par de nombreuses étapes intermédiaires, nécessaires notamment pour la manipulation correcte des indices de de Bruijn. Elle consiste en trois grandes étapes :

1. Équivalence de machines

   Raisonner directement sur une machine abstraite quelconque pour établir la correction est délicat, car les environnements peuvent être de n'importe quelle forme. On réduit donc le problème à l'étude de certaines machines particulières.

   Deux machines $\mathbf{M}$ et $\mathbf{N}$ sont dites équivalentes lorsque $[\![\mathbf{M}]\!]_{\mathcal{P}} = [\![\mathbf{N}]\!]_{\mathcal{P}}$, ce qu'on note $\mathbf{M} \sim \mathbf{N}$. Le représentant standard de la classe d'équivalence de $\mathbf{M}$, que l'on note $\widehat{\mathbf{M}}$, est défini comme $\widehat{\mathbf{M}} = [\![[\![\mathbf{M}]\!]_{\mathcal{P}}]\!]_{\mathcal{M}}$.

   C'est ce $\widehat{\mathbf{M}}$ qui va servir à établir la correction de la machine, car tous ses environnements sont vides, ce qui le rend plus manipulable. Parmi les premières propriétés de $\widehat{\mathbf{M}}$ on trouve la stabilité par $+$, c'est-à-dire $\widehat{\mathbf{M} + \mathbf{N}} = \widehat{\mathbf{M}} + \widehat{\mathbf{N}}$ et la bonne définition, soit $\widehat{\widehat{\mathbf{M}}} = \widehat{\mathbf{M}}$. Cette dernière propriété est d'ailleurs un simple corollaire du fait que $[\![[\![P]\!]_{\mathcal{M}}]\!]_{\mathcal{P}} = P$, ce que l'on a déjà vu.

2. Lemme principal

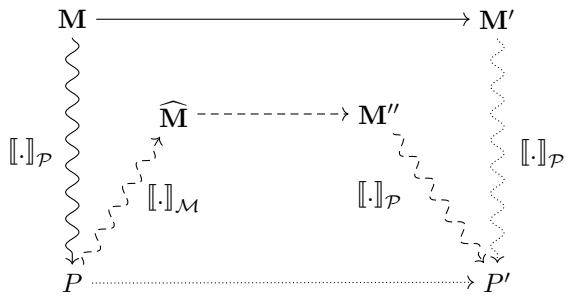
   Le lemme principal à établir correspond à montrer la correction de la machine lors d'une unique transition, en partant d'un représentant standard.

   Formellement, on montre que si $\mathbf{M} \to \mathbf{M}'$, alors $\widehat{\mathbf{M}} \to \mathbf{M}''$ avec $\mathbf{M}' \sim \mathbf{M}''$.

   Pour conclure la preuve de la correction, il faut enfin prouver que si $[\![P]\!]_{\mathcal{M}} \to \mathbf{M}''$ alors $P \to P'$ avec $[\![\mathbf{M}'']\!]_{\mathcal{P}} = P'$.

   Ces deux théorèmes sont décrits par les deux diagrammes :

   Mis bout à bout, on obtient donc la forme générale de la preuve :



3. De la correction à la complétude
   La complétude se déduit d'une propriété plus forte. On peut en effet étiqueter les transitions d'un processus : si P se réduit en P' alors P est de la forme $(a.R) \parallel \bar{a}\langle S\rangle \parallel Q$. On étiquette alors sa transition par le couple émission / réception : $P \xrightarrow{(\bar{a}\langle S\rangle,\, a.R)} P'$.
   On peut de même étiqueter les transitions au sein des machines : pour $\mathbf{M} = (\bar{a}\langle Q\rangle, e) + (a.R, f) + \mathbf{N} \to \mathbf{M}'$, on note $\mathbf{M} \xrightarrow{\left(\llbracket(\bar{a}\langle Q\rangle,e)\rrbracket_{\mathcal{P}},\, \llbracket(a.R,f)\rrbracket_{\mathcal{P}}\right)} \mathbf{M}'$.
   On a alors la nouvelle propriété de correction : Si $\mathbf{M} \xrightarrow{t} \mathbf{M}'$ alors $\llbracket\mathbf{M}\rrbracket_{\mathcal{P}} \xrightarrow{t} P'$ avec $\llbracket\mathbf{M}'\rrbracket_{\mathcal{P}} = P'$.
   On peut alors prouver la complétude en remarquant que si $P \xrightarrow{t} P'$, alors, du fait de la structure de P, pour tout $\mathbf{M}$ tel que $\llbracket\mathbf{M}\rrbracket_{\mathcal{P}} = P$, $\mathbf{M}$ admet une transition étiquetée par t vers un $\mathbf{M}'$. Or, par correction, P admet alors une transition étiquetée par t vers un P'' et $\llbracket\mathbf{M}'\rrbracket_{\mathcal{P}} = P''$. Enfin, comme $P \xrightarrow{t} P'$ et $P \xrightarrow{t} P''$, on obtient $P' = P''$ donc $\llbracket\mathbf{M}'\rrbracket_{\mathcal{P}} = P'$.

4 Implémentation

Nous avons implémenté notre machine abstraites de deux manières : en OCaml et en HOcore lui-même. Les implémentations sont disponibles en ligne. [2] Pour chaque implémentation nous avons été confrontés à la gestion du non-déterminisme, pour lequel nous proposons deux solutions. Une première solution est une implémentation interactive qui laisse l'utilisateur choisir quelle réduction a lieu. La deuxième solution est une implémentation qui explore toutes les réductions possibles et retourne une liste de processus.

4.1 En OCaml

4.1.1 Interpréteur

La première version, très naïve, modélise les processus comme des listes d'atomes, les atomes pouvant être des émissions, des réceptions ou des variables. Cette version convient pour une utilisation pas-à-pas, où la communication à choisir entre une émission et une réception est demandée à l'utilisateur à chaque réduction.

Elle n'est cependant pas suffisamment efficace pour une utilisation cherchant à explorer toutes les réductions possibles d'un processus initial donné. Dans ce cadre, nous avons amélioré l'interpréteur en raffinant le modèle de HOcore choisi, de façon à ce que l'exploration

2. https://people.rennes.inria.fr/Alan.Schmitt/research/hocore/

d'un terme possédant un nombre non borné de réductions prenne un temps raisonnable, même pour plusieurs milliers de réductions au maximum. Pour chaque version de l'interpréteur, nous indiquons entre parenthèses le nom sous lequel il est désigné dans la figure 1.

1. Regroupement par canal (Canaux)

   La première amélioration consiste à regrouper les émissions et les réceptions en fonction du canal sur lequel elles ont lieu. En utilisant des variables nommées, on obtient donc la structure suivante pour les processus :

   ```
   module SMap  = Map.Make(String)
   type process = Proc of ((send list * recv list) SMap.t) * var list
   and send = process
   and recv = var * process
   and var  = string
   ```

   L'intérêt de ce regroupement est de pouvoir identifier facilement les communications possibles, et de ne pas prendre en compte les canaux sur lesquels il n'y a pas de communication.

2. Compilation vers une fonction (Fonction)

   Une seconde amélioration possible consiste à transformer les réceptions en fonctions de la forme `type recv = process -> process`, les autres types étant conservés par ailleurs. En effet, la communication se fait alors très simplement en appliquant la réception (qui est une fonction) à l'émission (qui est un processus). Cependant, la création de telles fonctions ne peut pas se faire au sein du même programme qui fait l'analyse syntaxique du processus HOcore entré, car les différentes réceptions sont mutuellement récursives. Par exemple, la réception $a(x).b(y).x$ devra être transformé à terme en un objet dont la structure ressemblera à `fun x -> fun y -> x`, structure qui ne peut pas être créée lors d'un parcours de l'arbre de syntaxe abstrait du processus si l'on n'utilise pas de méta-programmation, une technique permettant de générer des programmes à l'exécution [9].

   La solution consiste donc à compiler le processus HOcore entré en un code OCaml contenant un objet de type `process`, qui sera ensuite compilé puis exécuté pour rechercher toutes ses réductions. On profite alors aussi de l'efficacité du compilateur OCaml pour que les fonctions constituant les réceptions soient optimisées.

   L'implémentation de cette méthode n'a cependant pas montré une amélioration significative des performances de l'interpréteur, sans doute à cause de la difficulté pour le compilateur d'optimiser les fonctions à travers l'appel au module `SMap`.

3. Structure de multiensemble (Multiensemble)

   Une dernière amélioration importante de l'interpréteur consiste à ne plus utiliser des listes, mais des multiensembles pour identifier plus facilement les processus identiques. L'intérêt de cet ajout réside dans l'exploration du graphe des réductions possibles, où l'on évite alors d'effectuer de nombreuses fois la même réduction. Cela requiert cependant l'utilisation de modules mutuellement récursifs pour pouvoir continuer à utiliser le foncteur `Map.Make`.

   Les performances sont effectivement nettement améliorées par cette approche.

4.1.2 Comparaison des performances

Le tableau de la figure 1 indique le temps d'exécution des quatre versions successives de l'interpréteur sur différents processus HOcore. Ces processus n'ont pas d'unique état

terminal : l'interpréteur cherche donc tous les états terminaux jusqu'à un certain nombre d'itérations au maximum.

Les processus servant aux tests sont :

— $\Omega = \bar{a}\langle\omega_a\rangle \parallel \omega_a$ où $\omega_a = a(x).(x \parallel \bar{a}\langle x\rangle)$.
On a $\Omega \to \Omega$.

— $P = p \parallel \bar{a}\langle p\rangle \parallel a(x).\bar{a}\langle *\rangle$ où $p = a(x).(x \parallel \bar{a}\langle x\rangle \parallel z)$ avec z une variable libre.
On a $P \to \begin{cases} z \parallel P \\ p \parallel \bar{a}\langle *\rangle \to z \parallel \bar{a}\langle *\rangle \end{cases}$.
L'arbre des réductions de P est donc un peigne de taille arbitraire, dont les feuilles sont de la forme $z \parallel z \parallel ... \parallel z \parallel \bar{a}\langle *\rangle$.

— C $= \bar{a}\langle c\rangle \parallel c$ où $c = a(x).(\bar{a}\langle x\rangle \parallel \bar{a}\langle x\rangle \parallel x)$.
On a $C \to \bar{a}\langle c\rangle \parallel C \to \bar{a}\langle c\rangle \parallel \bar{a}\langle c\rangle \parallel C \to ...$

— bi$\Omega = \bar{a}\langle\omega_a\rangle \parallel \omega_a \parallel \bar{b}\langle\omega_b\rangle \parallel \omega_b$. On a bi$\Omega \to$ biΩ.

Les performances sont données dans la figure 1, testées avec un processeur Intel® Core™ i5-3317U et OCaml v4.04.0. Les temps sont exprimés en secondes. TO correspond à un timeout, fixé à une minute. Le temps de la compilation, dans le cas de la version appelée "Fonction", est en moyenne de 0.036s.

Processus	Ω		P			C			$bi\Omega$		
Itérations	$\mathbf{2^4}$	$\mathbf{2^{21}}$	$\mathbf{2^{13}}$	$\mathbf{2^{15}}$	$\mathbf{2^{21}}$	$\mathbf{2^4}$	$\mathbf{2^9}$	$\mathbf{2^{10}}$	$\mathbf{2^4}$	$\mathbf{2^5}$	$\mathbf{2^{22}}$
Naïf	0	.3337	15.20	TO	TO	TO	TO	TO	.1145	TO	TO
Canaux	0	.7679	.0104	.0561	3.668	TO	TO	TO	.2343	TO	TO
Fonction	.036	.6370	.0442	.0695	1.753	TO	TO	TO	.2293	TO	TO
Multiensemble	0	.6822	0	0	0	0	8.052	TO	0	0	9.819

FIGURE 1 – Performance des interpréteurs

Ces temps sont cohérents avec les observations faites au-dessus. On peut remarquer quelques détails :

— Le processus C conduit à un timeout partout sauf pour la version avec multiensembles. En effet, on obtient $\underbrace{\bar{a}\langle c\rangle \parallel ... \parallel \bar{a}\langle c\rangle}_{n \text{ fois}} \parallel a(x).(\bar{a}\langle x\rangle \parallel \bar{a}\langle x\rangle \parallel x)$ après $n-1$ réductions à partir de C. Chacune des n émissions peut être reçue sur l'unique réception, ce qui conduit à n processus qui sont chacun traités séparément en l'absence de multiensembles. Le nombre de processus étudiés est donc factoriel en le nombre d'itérations sans multiensemble, alors qu'il n'y en a qu'un seul avec.

— Pour Ω, la version naïve est la plus efficace. Cela est simplement dû au fait qu'aucune des optimisations proposées ne s'applique sur un cas aussi simple que le processus Ω : il n'y a qu'un seul canal, une seule réception, et une seule transition possible à chaque fois. La vitesse vient donc de la légèreté de l'interpréteur.

— Cette dernière remarque s'applique aussi au processus biΩ, mais uniquement pour un très petit nombre d'itérations. En effet, en l'absence de multiensembles on retrouve un phénomène analogue à celui vu pour C : le nombre de processus à traiter est cette fois-ci exponentiel en le nombre d'itérations, donc l'optimisation sur les multiensembles a un effet très important.

4.1.3 Machine abstraite

L'implémentation de la machine abstraite se fait par-dessus celle de l'interpréteur. Il suffit d'y ajouter les types suivants :

```
type 'a annot = 'a * env
and env = Env of process annot list
type machine = ((send annot) list * (recv annot) list) SMap.t * var list
```

Les transitions de la machine se font alors comme définies, par insertion de l'émission dans l'environnement de la réception et évaluation du tout.

4.2 En HOcore

4.2.1 Motivations

Afin d'avoir une autre perspective sur le calcul, nous avons décidé d'implémenter la machine abstraite en HOcore.

L'intérêt d'une telle démarche est avant tout de mieux comprendre ce calcul. Plus spécifiquement, nous souhaitions voir comment manipuler des éléments de programmation élémentaires comme les boucles ou les nombres dans HOcore, qui est Turing-complet et devait donc pouvoir les exprimer. Cela permet par ailleurs de donner un aperçu de l'expressivité du langage, et en particulier de la façon d'échapper à la contrainte de n'utiliser qu'un nombre fixé de canaux, ainsi que la gestion du non-déterminisme.

Le programme à implémenter, en l'occurrence un interpréteur pour la machine abstraite, est suffisamment compliqué pour permettre d'explorer en détail ces aspects, mais reste réalisable dans le cadre très contraint de la programmation en HOcore. Enfin, cette implémentation nous a obligés à beaucoup simplifier la machine par rapport à ce qu'elle était à l'origine, ce qui a permis en retour de clarifier les preuves et rendre la machine sans doute mieux adaptable à HOπ.

4.2.2 Structure de l'implémentation

Pour écrire la machine, nous avons repris la syntaxe sans indice de de Bruijn pour HOcore. Nous y avons ajouté un peu de sucre syntaxique pour clarifier la présentation de la machine abstraite.

La syntaxe concrète d'une émission est `a<P>` et celle d'une réception `a(x).P`, où `a`, `x` et `P` sont des chaînes de caractères. On autorise la syntaxe `a.P` pour une réception n'utilisant pas le processus reçu. Ceci est particulièrement utile dans le cas où la réception sert juste à signaler la fin d'une action, ce qui intervient régulièrement lorsque l'on souhaite déterminiser l'exécution du processus HOcore. La composition parallèle s'écrit `P || P` et le processus vide est noté `*`.

D'autres ajouts au langage sont présentés au paragraphe suivant. De façon générale, ces ajouts sont écrits comme des macros dans la syntaxe du préprocesseur C : cela sert à pouvoir facilement transformer le code écrit avec ces ajouts en un code directement lisible par un interpréteur HOcore.

La machine, une fois traduite en HOcore pur, après expansion des macros, s'écrit en 1150 mots, un mot pouvant désigner un canal ou une variable. Il faut cependant y ajouter la taille du processus que l'on souhaite évaluer dans la machine abstraite.

Afin d'être interprété par la machine, le processus est traduit de façon à identifier les différents canaux et variables qu'il emploie en une autre représentation, de façon à ne pas confondre

ceux du processus évalué et ceux de la machine elle-même, et à pouvoir les manipuler correctement. Pour un processus tel que Ω, sa traduction s'écrit en 270 mots. Cette taille provient de l'utilisation de nombres, dont la syntaxe est présentée au paragraphe suivant, pour représenter les variables et les canaux du processus.

4.2.3 Éléments de programmation

Voici un aperçu des différents éléments de programmation en HOcore que nous avons développés, et qui nous ont été très utiles pour cette implémentation.

1. Entiers naturels

   Les entiers naturels sont représentés en HOcore d'une façon assez similaire à leur représentation en λ-calcul par les entiers de Church. En effet, on représente un nombre n comme une réception sur le canal `repeat`, qui attend un processus et le répète n fois, puis envoie le signal `zero<*>` pour signaler la fin de son exécution. On définit ainsi, dans la syntaxe du préprocesseur C :

   ```
   #define ZERO       (repeat.zero<*>)
   #define SUCC(n)    (repeat(P).(P || ACK_rep.(repeat<P> || n)))
   ```

   La convention choisie ici est que le processus `P` doit envoyer à la fin de son exécution un signal sur le canal `ACK_rep`.

   Parmi les opérations élémentaires, on peut ainsi définir l'addition par exemple :

   ```
   #define ADD(n,m)   temp<n> || m || zero.temp(x).RET_add<x> ||       \
                      repeat< temp(x).(temp<SUCC(x)> || ACK_rep<*>) >
   ```

   Le résultat de l'opération peut être obtenu sur le canal `RET_add`.

2. Listes

   Les listes chaînées ont une représentation intuitive, la tête de la liste étant dans le canal `hd` et la queue, dans le canal `tl`. On garde dans `len` la taille de la liste, afin de pouvoir la parcourir plus simplement :

   ```
   #define NIL        hd<*> || tl<*> || len<ZERO>
   #define push(x,l)  (l || hd.tl.len(n).(RET_push<hd<x> || tl<l> ||   \
                      len<SUCC(n)>>))
   ```

   L'accès au $\mathtt{i}^{\mathtt{e}}$ élément de la liste `l` se fait alors par :

   ```
   #define nth(i,l)  l || i || repeat<hd.len.tl(s).(s || ACK_rep<*>)>  \
                  || (zero.tl.len.hd(x).RET_nth<x>)
   ```

3. Booléens

   Les booléens sont représentés de la façon suivante :

   ```
   #define TRUE      (false.true(x).x)
   #define FALSE     (true.false(x).x)
   ```

   Une conditionnelle de la forme "*if* `e` *then* `x` *else* `y`" où `e` est un booléen s'écrit alors :

   ```
   e || true<x> || false<y>
   ```

   On peut remarquer que cette structure est en fait très adaptée à un raisonnement par cas avec un nombre quelconque de cas, en remplaçant `true` et `false` par `case_1`, `case_2`, ..., `case_n` avec un pseudo-booléen de la forme `(case_1. ... .case_{i-1}.case_{i+1}. ... .case_n.case_i(x).x)`.

4. Boucles

   Les boucles enfin s'écrivent par analogie avec le terme Ω vu plus haut. Une boucle de la forme "*while* `e` *do* `x`" où `x` se termine par une émission sur `ACK_loop` s'écrit :

```
INIT_loop< loop(o).(e || false<ACK_endloop<*>>
                     || true< x || ACK_loop.(o || loop<o>) >)
        > ||
INIT_loop(o).(o || loop<o>)
```

   Cette boucle se terminera alors par une émission sur `ACK_endloop`.

5 Travaux futurs et conclusion

Nous avons présenté une machine abstraite pour un calcul de processus d'ordre supérieur, HOcore. Cette machine est correcte, c'est-à-dire que toutes ses transitions correspondent à une réduction possible du processus qu'elle simule, et complète, c'est-à-dire qu'elle peut simuler toutes les suites de réductions du processus qui lui est associé. Nous avons également présenté plusieurs implémentations de la machine abstraite, dont une en HOcore illustrant la facilité de programmer dans ce calcul minimal.

La suite logique de ce travail est la généralisation de la machine abstraite à HOπ, en ajoutant les restrictions de nom et en adaptant la machine de façon adéquate. Cela pourrait encore s'étendre en ajoutant une notion de localité afin de simuler la distribution physique des opérations au sein d'un réseau.

Une deuxième suite à ce travail est la formalisation des preuves de correction et de complétude de la machine abstraite dans un assistant de preuve. Nous nous sommes astreints à ce que la preuve soit la plus détaillée possible, donc nous n'envisageons pas de difficulté majeure, la sémantique de HOcore étant déjà formalisée en Coq [12].

Par ailleurs, une simplification possible de la machine consisterait en une division de la transition en sous-transitions plus élémentaires, comme expliqué en 3.1.2. Les preuves de correction et complétude avec cette alternative peuvent se déduire de celles faites avec la transition retenue, donc l'essentiel du travail est déjà fait. Cela rendrait par ailleurs la machine plus flexible, et peut-être plus adaptée pour représenter des processus en cours d'exécution.

Sur un autre plan, il serait intéressant de définir une notion de bisimulation pour la machine abstraite qui corresponde à celle sur HOcore. Les bisimulations sont des relations d'équivalences adaptées à la comparaison de processus distribués [14]. HOcore dispose de plusieurs propriétés très fortes par rapport aux bisimulations, notamment la décidabilité de celles-ci et leur équivalence avec la congruence barbue [11]. La congruence barbue est la plus petite congruence fermée par réduction et qui préserve les barbes, c'est à dire le fait qu'un message soit envoyé sur un nom donné. La conception d'une notion de bisimulation simple sur la machine abstraite qui corresponde à celle des processus n'est cependant pas immédiate à cause des environnements de la machine qui n'ont pas de contrepartie dans le processus HOcore.

Références

[1] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1) :217 – 248, 1992.

[2] Philippe Bidinger and Adriana Compagnoni. Pict correctness revisited. *Theor. Comput. Sci.*, 410(2-3) :114–127, February 2009.

[3] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, 49(3) :363–408, October 2012.

[4] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5) :381–392, 1972.

[5] Stephan Diehl, Pieter Hartel, and Peter Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16(7) :739 – 751, 2000.

[6] Cédric Fournet and Georges Gonthier. The reflexive cham and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 372–385, New York, NY, USA, 1996. ACM.

[7] Cédric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier Rémy. *A calculus of mobile agents*, pages 406–421. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[8] Florence Germain, Marc Lacoste, and Jean-Bernard Stefani. An abstract machine for a higher-order distributed process calculus. *Electronic Notes in Theoretical Computer Science*, 66(3) :145 – 169, 2002. F-WAN, Foundations of Wide Area Network Computing (ICALP 2002 Satellite Workshop).

[9] Oleg Kiselyov. The design and implementation of BER MetaOCaml - system description. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 86–102. Springer, 2014.

[10] Jean-Louis Krivine. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation*, 20(3) :199–207, 2007.

[11] Ivan Lanese, Jorge A. Peréz, Davide Sangiorgi, and Alan Schmitt. On the Expressiveness and Decidability of Higher-Order Process Calculi. In *23rd Annual IEEE Symposium on Logic in Computer Science (LICS 2008)*, Proceedings of the 23rd Annual IEEE Symposium on Logic in Computer Science (LICS 2008), pages 145–155, Pittsburgh, Pennsylvania, United States, June 2008.

[12] Petar Maksimovic and Alan Schmitt. Hocore in Coq. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving - 6th International Conference, ITP 2015, Nanjing, China, August 24-27, 2015, Proceedings*, volume 9236 of *Lecture Notes in Computer Science*, pages 278–293. Springer, 2015.

[13] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[14] Damien Pous. Using bisimulation proof techniques for the analysis of distributed abstract machines, 2008.

[15] Davide Sangiorgi. *Expressing mobility in process algebras : first-order and higher-order paradigms*. 1993.

[16] Davide Sangiorgi and Andrea Valente. A distributed abstract machine for safe ambients. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming*, ICALP '01, pages 408–420, London, UK, UK, 2001. Springer-Verlag.

Hydra Ludica : Une preuve d'impossibilité de prouver simplement *

Pierre Castéran

Univ. Bordeaux, LaBRI, CNRS, INP Bordeaux,
pierre.casteran@u-bordeaux.fr

Résumé

Les jeux d'hydre (aussi appelés batailles d'hydre), dus aux mathématiciens L. Kirby et J. Paris, forment un exemple de système de transitions dont la terminaison est difficile à démontrer. Cette difficulté de prouver est exprimée par ces auteurs sous forme d'un méta-théorème de l'arithmétique de Peano. Nous en présentons une adaptation en Coq sous forme d'une preuve d'impossibilité d'utiliser des relations bien fondées trop simples. Ce développement s'appuie sur une représentation finitaire des ordinaux inférieurs à ϵ_0 sous forme normale de Cantor.

1 Introduction

Les preuves formelles d'impossibilité permettent de se pencher sur la structure des preuves, la pertinence d'hypothèses, et d'étudier la puissance d'expression d'un formalisme.

De nombreux exemples se trouvent en théorie des langages (application des lemmes d'itération [13]), en algorithmique distribuée [3, 1, 6], etc. Ces preuves partagent fréquemment la même structure : une preuve par l'absurde, où l'hypothèse d'une solution à un problème donné permet de construire un contre-exemple. Dans le cas d'un langage formel dont on veut prouver la non-reconnaissabilité, un lemme d'itération permet, à partir d'un hypothétique automate reconnaissant ce langage, de construire un mot accepté par l'automate mais en dehors du langage considéré. De la même façon, prouver qu'une classe de systèmes de calculs locaux ne peut pas résoudre le problème de l'élection se fait en construisant, pour tout algorithme de cette classe, un graphe et une exécution désignant deux sommets élus de ce graphe au lieu d'un seul. Nous nous intéressons dans cet article à une espèce particulière de preuve d'impossibilité : la non-existence de preuves simples d'un théorème donné.

En 1982, les mathématiciens Laurie Kirby et Jeff Paris ont publié un article [10] contenant entre autres :

1. La description d'un jeu mathématique : le combat d'Hercule contre l'hydre. La particularité d'une bataille d'hydre est que ce monstre a la capacité d'augmenter de taille à chaque round. À l'instar des suites de Goodstein, étudiées dans le même article, les batailles d'hydres peuvent avoir une durée qui dépasse notre intuition.
2. La preuve d'un théorème *a priori* surprenant : Hercule gagne toujours contre l'hydre. L'adaptation en Coq de cette preuve se trouve dans la contribution sur les notations d'ordinaux [5].
3. Un second théorème établissant que la preuve du premier théorème ne *peut pas* être simple. L'objet de cet article est la mécanisation de ce résultat.

*Ce travail a bénéficié de l'aide des projets ANR Impex et ESTATE

Dans une première partie, nous rappelons les règles des jeux d'hydre, ainsi que la preuve de leur terminaison. Puis nous montrons comment formaliser avec les constructions de Coq une version faible du second théorème de Kirby et Paris. Enfin, nous discutons des prolongements possibles de ce développement.

2 Hydres et batailles d'hydre

Définition 1. *Une* hydre *est un monstre mythologique en forme d'arbre à branchement fini. Nous appelons* tête *toute feuille de l'arbre, et* pied *la racine de cet arbre. La longueur de la suite d'arêtes menant du pied à un sommet est appelée* hauteur *de ce sommet.*

La partie gauche de la figure 1 représente une hydre de hauteur 4 et comportant 5 têtes.

2.1 Règles du jeu

Un jeu d'hydre se joue à deux joueurs, Hercule et l'hydre, sous la forme d'une suite de *tours*, *reprises*, ou encore *rounds* [1]. Le jeu se termine par la victoire d'Hercule lorsque l'hydre est réduite à une tête.

À chaque tour, Hercule coupe une tête de l'hydre :

1. Si cette tête est de hauteur 1, aucune réaction.
2. Si la tête est de hauteur ≥ 2 :
   (a) On considère le sous-arbre issu du grand-père de la tête coupée (sur les dessins, la partie de l'hydre dont les têtes sont tristes).
   (b) Soit i un entier naturel. On ajoute i copies de cette sous-hydre à la même place ; l'entier i est appelé *facteur de réplication* du tour considéré.

Remarquons, que, si un sommet de l'hydre a un seul fils réduit à une tête, et que celle-ci est coupée, alors ce sommet devient à son tour une tête.

Les figures de 1 à 3 représentent le début d'un combat. L'emplacement d'où une tête a été enlevée est représenté par une arête en pointillé. Au premier tour, une tête de hauteur 1 est coupée, et l'hydre ne réagit pas. Au deuxième tour, l'hydre ajoute 4 copies de la zone blessée. Le facteur de réplication au troisième tour est de 2. Il est clair que l'hydre n'augmente jamais de hauteur lors d'un combat ; en revanche, sa taille peut devenir très grande.

FIGURE 1 – Premier tour d'une bataille

1. Ce mot d'origine anglaise est tout à fait admis en français. Nous utiliserons indifféremment ces trois mots dans cet article.

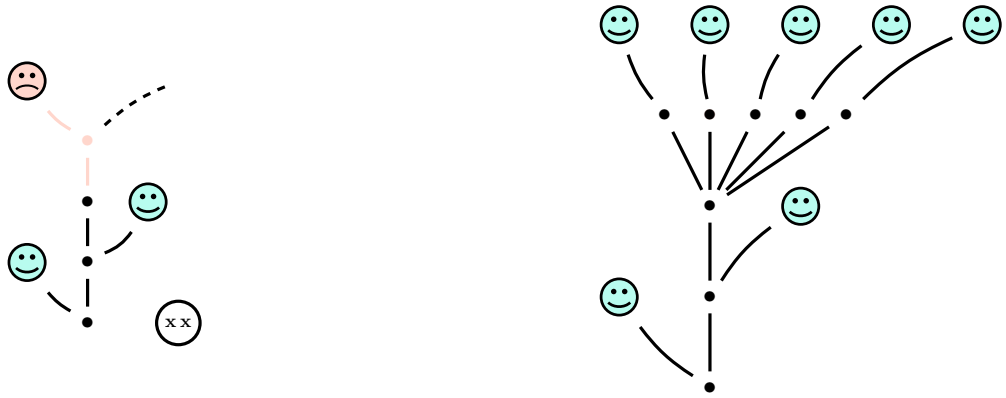


FIGURE 2 – Le deuxième tour

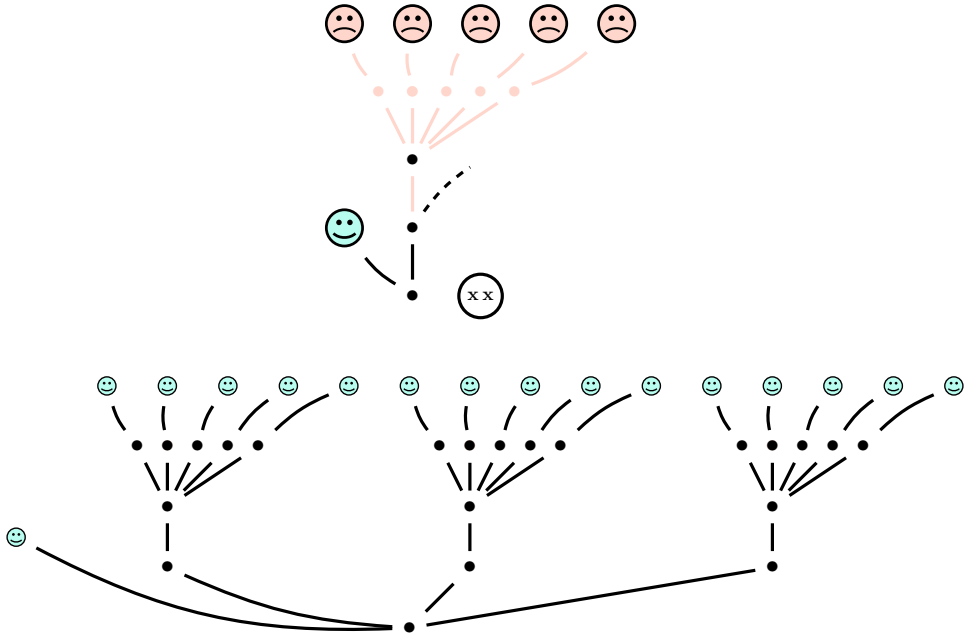


FIGURE 3 – Troisième reprise

Un combat peut durer *très* longtemps : considérons la petite hydre de la figure 4, et supposons qu'Hercule choisit toujours de couper la tête la plus à droite parmi les plus basses, et qu'au i-ème tour, le facteur de réplication soit de i. Alors, par comparaison avec la suite de Goodstein [10, 5] issue de 4, nous pouvons prouver que la bataille prendra plus de $3 \times 2^{402653211} - 1$ tours.

2.2 Le théorème de terminaison

L'article de Kirby et Paris de 1982 contient une preuve d'un théorème a priori étonnant.

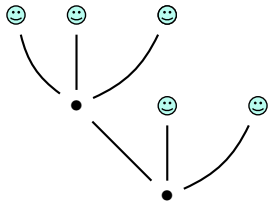


FIGURE 4 – L'hydre associée à la suite de Goodstein issue de 4

Théorème 1. *Quelles que soient les stratégies d'Hercule (choix d'une tête) et de l'hydre (choix du nombre de réplications), Hercule finit par gagner.*

2.2.1 Rappels sur les nombres ordinaux

La preuve du théorème 1 utilise les *nombres ordinaux*. Du point de vue mathématique, un ordinal est un représentant d'une classe d'équivalence (pour l'isomorphisme d'ordres) de relations d'ordre strict, totales et bien fondées. D'un point de vue ensembliste, un ordinal α est un ensemble dont les éléments sont les ordinaux strictement inférieurs à α. La relation $<$ sur les ordinaux est bien fondée, et l'ordre large $\leq$ associé est total.

Un ordinal α est un *ordinal successeur* s'il existe un ordinal β tel que α est le plus petit ordinal strictement supérieur à β. Un ordinal λ est un *ordinal limite* s'il est la borne supérieure d'une suite strictement croissante d'ordinaux.

Un ordinal est soit 0, soit un ordinal limite, soit un ordinal successeur. Cette division en cas, ainsi que la tactique de preuve par récurrence bien-fondée (aussi appelée *récurrence transfinie*) structurent de nombreuses preuves sur les ordinaux.

Par respect de la tradition, les ordinaux seront appelés α, β, γ, etc. On réservera la méta-variable λ aux ordinaux limites.

Le premier ordinal infini est l'ordinal limite ω. L'ordinal ϵ_0 est le premier ordinal α vérifiant l'égalité $\alpha = \omega^\alpha$. Certains ordinaux dénombrables : ω, ω^2, ω^ω, ϵ_0, Γ_0, ont une représentation sous forme de termes finis permettant le calcul et rendant la comparaison décidable. Dans cet article nous utilisons la *forme normale de Cantor* pour représenter l'ensemble des ordinaux inférieurs à ϵ_0. Sauf mention contraire, tous les ordinaux mentionnés dans cet article seront inférieurs à ϵ_0.

2.2.2 Structure de la preuve du théorème de terminaison

La preuve de terminaison de toute bataille d'hydre de [10] s'articule de la façon suivante :

1. On considère une *mesure* associant à toute hydre h un ordinal inférieur à ϵ_0. Cette mesure est définie récursivement de la façon suivante :
   — Si h est une tête, alors $m(h) = 0$
   — Si h est formée d'un pied et des sous-hydres $h_1, h_2, \ldots, h_n$, alors

   $$m(h) = \omega^{m(h_1)} \oplus \omega^{m(h_2)} \oplus \cdots \oplus \omega^{m(h_n)}$$

   où $\oplus$ est la somme commutative et strictement monotone d'ordinaux, appelée *somme d'Hessenberg* ou *somme naturelle*.
2. On prouve que, si h se transforme en h' en un round, alors $m(h') < m(h)$.

3. L'ordre $<$ sur les ordinaux étant bien fondé, on en déduit que toute bataille est finie. De par les règles vues en 2.1, toute bataille se termine forcément par la victoire d'Hercule.

L'article de N. Dershowitz et G. Moser « The Hydra Battle Revisited » [7] décrit de façon très complète les preuves à propos des batailles d'hydre. Il cite également une classe de batailles d'hydres plus vaste, où les hydres peuvent augmenter de hauteur, tout en garantissant la terminaison de toutes les batailles, prouvée grâce à l'ordinal Γ_0.

L'article de Will Sladek « The Termite and the Tower » [16] propose quelques explications très lisibles sur une preuve très similaire : la terminaison des suites de Goodstein, et l'utilisation des ordinaux dans cette preuve.

2.3 Preuve en Coq du théorème de terminaison

La preuve de terminaison de toutes les batailles d'hydre fait partie d'une contribution écrite avec Évelyne Contejean [5]. Cette preuve contient une partie générique sur les notations d'ordinaux, notamment la forme normale de Cantor. Tout ordinal inférieur à ϵ_0 est représenté par un terme fini d'un type inductif appelé `T1`, repris de la bibliothèque écrite par P. Manolios et D. Vroon pour ACL2 [11].

```
(** * Cantor "pre" Normal form
     After Manolios and Vroon's work on ACL2

T1 is the type of terms used for representing Cantor normal form.
The term (ocons a n b) is intended to represent the ordinal
omega^a * (S n)  + b
*)

Inductive T1 : Set  :=
| zero : T1
| ocons : T1 -> nat -> T1 -> T1.
```

Sur ce type, nous définissons un ordre bien fondé et les opérations usuelles : somme, produit, exponentiation de base ω. La somme d'Hessenberg ne fait pas partie de la contribution de 2006, mais a été ajoutée depuis pour simplifier nos preuves.

Les hydres sont représentées comme des arbres dont chaque sommet possède un nombre fini mais quelconque de fils. Dans la définition suivante, les types `Hydra` et `Hydrae` sont respectivement associés aux hydres et aux suites finies d'hydres :

```
Inductive Hydra : Set :=
| node :  Hydrae -> Hydra
with Hydrae : Set :=
| hnil : Hydrae
| hcons : Hydra -> Hydrae -> Hydrae.

Notation head := (node hnil).
```

2.3.1 Représentation des règles de combat

Notre formalisation des règles de bataille d'hydre se fait sous la forme d'une relation binaire sur le type `Hydra` exprimant la transformation associée à chaque reprise. Soit $i \geq 0$, on note $h \underset{i}{\longrightarrow} h'$ la transformation de h en h' telle que :

— soit la tête coupée est de hauteur 1 et la valeur de i est non pertinente,
— soit cette tête est de hauteur ≥ 2 et i est le facteur de réplication associé.

Dans nos preuves, nous utiliserons quelques relations définies à partir de $\underset{i}{\longrightarrow}$:

$\longrightarrow$ (`-1->` dans les scripts Coq) : oubli du paramètre i

$\overset{+}{\underset{i}{\longrightarrow}}$: fermeture transitive de $\underset{i}{\longrightarrow}$

$\overset{+}{\longrightarrow}$ (`-+->` dans les scripts Coq) : fermeture transitive de $\longrightarrow$

2.3.2 Preuve formelle de terminaison

La formalisation en Coq de la preuve de terminaison suit fidèlement la preuve mathématique de l'article de 1982. Nous en décrivons ci-dessous la structure principale :

— Bibliothèque sur les formes normales de Cantor. La bonne fondation de l'ordre sur les ordinaux est prouvée de façon directe (preuve d'accessibilité) mais aussi de façon beaucoup plus concise en appliquant un développement d'Évelyne Contejean sur l'ordre récursif des chemins (*r.p.o.*)
— Définition et propriétés de la somme d'Hessenberg.
— Description des jeux d'hydre et preuve de terminaison proprement dite.

Cette preuve prend plus de 4500 lignes de script Coq. Remarquons que les deux premières parties sont génériques et seule la troisième concerne les hydres. Il est cependant naturel de se demander si toute cette complexité est nécessaire pour prouver ce théorème de terminaison. L'article de Kirby et Paris apporte la réponse suivante :

Théorème 2. *La terminaison de toutes les batailles d'hydre ne peut pas se prouver dans l'arithmétique de Peano.*

3 Le théorème d'impossibilité

Le théorème 2 est un très beau résultat, mais dont l'énoncé pourrait ne pas trop parler à l'utilisateur de Coq, habitué à l'ordre supérieur et peu disposé à accepter des restrictions sur les règles logiques à utiliser. Nous proposons alors l'énoncé suivant, qui explicite la nécessité de considérer l'ordinal ϵ_0 dans la preuve de terminaison :

Théorème 3. *Soit α un ordinal strictement inférieur à ϵ_0 ; la terminaison de toutes les batailles d'hydre ne peut pas se prouver à l'aide d'une mesure des hydres vers l'intervalle $[0, \alpha[$.*

Notation Pour abréger les énoncés, nous noterons $P(\alpha)$ la propriété énoncée par le théorème 3.

Définition 2. *Par la suite, nous appellerons* variant *toute mesure vers un ensemble bien fondé qui décroit strictement à chaque reprise.*

3.1 Le cas de l'ordinal ω^2

La preuve du théorème 3 pour $\alpha < \epsilon_0$ quelconque est assez technique. Aussi nous présentons d'abord le cas $\alpha = \omega^2$, beaucoup plus simple, mais qui possède la même structure globale de preuve que le cas général, notamment la distinction entre ordinaux zéro, successeurs et limites.

L'ordinal ω^2, vu comme ensemble, est représenté par le produit cartésien $\mathbb{N} \times \mathbb{N}$ muni de l'ordre lexicographique usuel. L'ordinal $\omega \times i + j$ est alors représenté par le couple d'entiers (i, j). Les couples de la forme $(i, 0)$ $(i > 0)$ représentent les ordinaux limites, et les couples (i, j) $(j > 0)$ les ordinaux successeurs.

Notre preuve de $P(\omega^2)$ possède la structure suivante :

1. On plonge ω^2 dans l'ensemble des hydres par la fonction ι qui à tout couple (i, j) associe l'hydre composée de i tentacules de longueur 2 et j tentacules de longueur 1. Par exemple, la figure 5 représente l'hydre $\iota(3, 5)$.
2. On prouve, pour *n'importe quel variant m à valeur dans ω^2*, l'inégalité : $\forall \beta < \omega^2,\ \beta \leq m(\iota(\beta))$.
   La preuve se fait par récurrence bien-fondée sur ω^2 :
   Soit $\beta = (i, j)$ tel que l'inégalité ci-dessus est vérifiée pour tout couple strictement inférieur à β.
   — Si $\beta = (0, 0)$, l'inégalité est triviale.
   — Si $j > 0$, alors l'hydre $\iota(i, j)$ se tranforme en un round en $\iota(i, j - 1)$ (par perte d'une tête). Comme m est un variant, nous avons $m(\iota(i, j)) > m(\iota(i, j - 1))$, d'où $m(\iota(i, j)) > (i, j - 1)$ (par hypothèse de récurrence), et donc $m(\iota(i, j)) \geq (i, j)$.
   — Si $i > 0$ et $j = 0$, alors $(i, 0)$ est la limite de la suite $(i - 1, k)$ $(k \in \mathbb{N})$.
   Il suffit alors de montrer que $m(\iota(i, 0))$ est strictement supérieur à $(i-1, k)$ pour tout k. Or pour tout k, il suffit d'un round pour transformer l'hydre $\iota(i, 0)$ en $\iota(i - 1, k)$. Comme m est un variant, et par hypothèse de récurrence, nous avons $m(\iota(i, 0)) > m(\iota(i - 1, k)) \geq (i - 1, k)$ pour tout k. Par passage à la limite, nous obtenons enfin $m(\iota(i, 0)) \geq (i, 0)$.
3. Considérons l'hydre h_{ω^2} de la figure 6. Sa mesure $m(h_{\omega^2})$ est un couple (i, j). En deux reprises, h_{ω^2} se transforme en $\iota(i, j)$: dans un premier temps, nous obtenons une hydre avec $i + 1$ tentacules de longueur 2. Au round suivant, on remplace le plus à droite de ces tentacules par j tentacules de longueur 1. Pour tout variant m, on aura alors $m(h_{\omega^2}) > m(\iota(i, j)) = m(\iota(m(h_{\omega^2})))$.
4. En appliquant les deux étapes précédentes, nous obtenons l'inégalité suivante qui contredit l'irréflexivité de l'ordre strict $>$ sur ω^2 :

$$m(h_{\omega^2}) > m(\iota(m(h_{\omega^2}))) \geq m(h_{\omega^2})$$

3.2 Un schéma de preuve générique

Afin de mettre en valeur la structure de notre preuve d'impossibilité, indépendamment des propriétés spécifiques de tel ou tel ordinal α, nous abstrayons cette structure par une classe paramétrée par le type du variant considéré. En premier lieu, nous associons une classe à la notion de variant bien fondé. Soit une relation $<$ sur un type t. Toute instance de la classe suivante prouve la bonne fondation de $\longrightarrow^{-1}$, et donc la terminaison de toutes les batailles d'hydre.

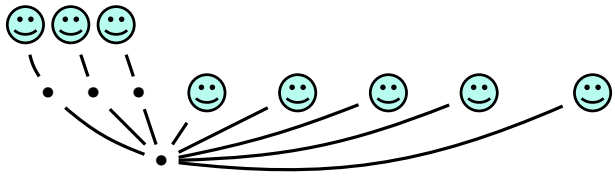


FIGURE 5 – L'hydre $\iota(3,5)$

FIGURE 6 – L'hydre h_{ω^2}

```
Class WfVariant (m: Hydra -> t) :=
  {
    wf : well_founded lt;
    decr : forall h h', h -1->  h' -> m h' < m h
  }.
```

Si l'on veut montrer qu'un ordre bien fondé $<$ ne permet pas de prouver la terminaison des batailles d'hydre, il suffit de montrer qu'il n'existe aucune instance de la classe `WfVariant` pour cet ordre. La classe suivante généralise la preuve de la section 3.1, et notamment l'usage de ses deux inégalités contradictoires.

```
Context (S: StrictOrder lt).
Infix "<" := lt.

Let le := clos_refl _ lt.
Infix "<=" := le.

Class TooSimple :=
 {
  iota : t -> Hydra;
  too_simple_1 :  forall m (V: WfVariant lt m) alpha,
                   alpha <= m (iota alpha);
  h: Hydra;
  too_simple_2 : forall m (V: WfVariant lt m),
                  h -+-> iota (m h)
  }.
```

Si l'on peut créer une instance de `TooSimple`, alors, pour tout variant m : `Hydra` $\to t$, on peut prouver $m(h) > m(\iota(m(h))) \geq m(h)$, et par conséquent `False`. La preuve de la section 3.1 est bien une instance de cette classe.

3.3 Preuve générale d'impossibilité

Nous avons alors à construire une instance de `TooSimple` pour le type $t = \{\beta : \texttt{T1} | \beta < \alpha\}$, où α est un ordinal quelconque strictement inférieur à ϵ_0. Nous donnons ci-dessous les étapes principales de cette preuve.

3.3.1 Plongement de $[0, \epsilon_0[$ dans `Hydra`

On définit récursivement une injection ι associant une hydre à tout ordinal $\beta < \epsilon_0$.

— $\iota(0) = \texttt{head}$

— Soit $\beta = \omega^{\beta_1} \times n_1 + \cdots + \omega^{\beta_p} \times n_p$ en forme normale de Cantor. Alors $\iota(\beta)$ est l'hydre formée d'un pied relié à n_1 copies de $\iota(\beta_1)$, ..., n_p copies de $\iota(\beta_p)$.

Par exemple, la figure 7 montre l'hydre associée à l'ordinal $\omega^{\omega+2} + \omega^\omega \times 2 + \omega + 1$.

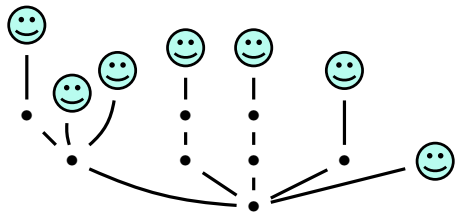


FIGURE 7 – L'hydre $\iota(\omega^{\omega+2} + \omega^\omega \times 2 + \omega + 1)$

3.3.2 La « machinerie » de Ketonen et Solovay

L'article de J. Ketonen et R. Solovay « Rapidly growing Ramsey functions » [9] est cité à l'appui des preuves de [10]. Une partie de cet article est consacrée à l'étude systématique de la relation $<$ sur l'intervalle $[0, \epsilon_0]$.

Les auteurs définissent la notion de « suite canonique ». Soit $\alpha < \epsilon_0$ un ordinal, et i un entier positif. On définit l'ordinal $\{\alpha\}(i)$ par récursion structurelle sur les formes normales de Cantor :

$$
\begin{array}{rcl}
\{0\}(i) & = & 0 \\
\{\alpha+1\}(i) & = & \alpha \\
\{\omega^{\beta+1}\}(i) & = & \omega^\beta \times i \\
\{\omega^\lambda\}(i) & = & \omega^{\{\lambda\}(i)} \quad (\lambda \textit{ ordinal limite}) \\
\{\omega^{\beta+1} \times n\}(i) & = & \omega^{\beta+1} \times (n-1) + \omega^\beta \times i \quad (n \geq 2) \\
\{\omega^\lambda \times n\}(i) & = & \omega^\lambda \times (n-1) + \omega^{\{\lambda\}(i)} \quad (n \geq 2,\ \lambda \textit{ ordinal limite}) \\
\{\omega^\alpha \times n + \beta\}(i) & = & \omega^\alpha \times n + \{\beta\}(i) \quad (n \geq 1,\ 0 < \beta)
\end{array}
$$

Notons que la définition ci-dessus diffère de celle, plus concise, de [9]. Notre définition, par sa récursion structurelle sur le type `T1`, simplifie le calcul des $\{\alpha\}(i)$ par l'évaluateur de Coq.

Par exemple, les égalités suivantes se prouvent par la simple tactique `reflexivity`.

$$\{\omega^\omega\}(4) = \omega^4 \qquad \{\omega^{\omega+2}\}(6) = \omega^{\omega+1} \times 6$$

.

L'intérêt des suites canoniques tient en les deux énoncés ci-dessous, prouvés dans [9].

Lemme 1. *Soit λ un ordinal limite. Alors λ est la limite de la suite strictement croissante $\{\lambda\}(i)$ $(i \in \mathbb{N})$.*

Théorème 4. *Soient deux ordinaux α et β tels que $\beta < \alpha < \epsilon_0$. Alors il existe un entier i et une suite d'ordinaux $\alpha = \alpha_0, \alpha_1, \ldots, \alpha_n = \beta$ telle que $\alpha_{k+1} = \{\alpha_k\}(i)$ pour tout $k < n$.*

3.3.3 Fin de la preuve

La relation entre les suites canoniques et les batailles d'hydre fait l'objet du lemme suivant :

Lemme 2. *Soit $0 < \alpha < \epsilon_0$ et $i \geq 1$ un entier. Alors $\iota(\alpha)$ se transforme en un round en $\iota(\{\alpha\}(i))$. Par conséquent, si $\beta < \alpha < \epsilon_0$, alors il existe une bataille transformant $\iota(\alpha)$ en $\iota(\beta)$.*

La première partie de ce lemme se prouve en construisant un round où Hercule coupe la plus à droite des têtes les plus basses de l'hydre, et l'hydre réagit avec un facteur de réplication de $i - 1$. En second lieu, le théorème 4 nous permet de finir la preuve.

Nous avons désormais tous les ingrédients pour construire une preuve de $P(\alpha)$, avec $\alpha < \epsilon_0$ quelconque. Soit m un variant quelconque à valeurs dans le type $\{\beta : \texttt{T1} \mid \beta < \alpha\}$.

On prouve d'abord l'inégalité $m(\beta) \geq \beta$ pour tout $\beta < \alpha$. Comme pour le cas de ω^2, on procède par récurrence bien fondée sur β, en considérant les trois cas $\beta = 0$, β ordinal successeur, et β ordinal limite. Seul ce dernier cas est non-trivial :

1. β est la limite (borne supérieure) de la suite canonique $\{\beta\}(i)$ $(i \in \mathbb{N})$.
   (a) Soit donc $\gamma < \beta$. Il existe un indice i tel que $\gamma < \{\beta\}(i)$.
   (b) Par le lemme 2, l'hydre $\iota(\beta)$ se transforme en un round en $\iota(\{\beta\}(i))$.
   (c) Comme m est un variant, nous avons $m(\iota(\beta)) > m(\iota(\{\beta\}(i)))$.
   (d) Par l'hypothèse de récurrence, nous avons $m(\iota(\{\beta\}(i))) \geq \{\beta\}(i) > \gamma$, donc $m(\iota(\beta)) > \gamma$.
2. L'ordinal $m(\iota(\beta))$ est donc strictement supérieur à tout ordinal γ strictement inférieur à β. Comme l'ordre $\leq$ est total, nous en déduisons l'inégalité $m(\iota(\beta)) \geq \beta$.

Considérons l'hydre $h = \iota(\alpha)$. Il reste à prouver l'inégalité $m(h) > m(\iota(m(h)))$.

1. Par hypothèse sur m, $m(h)$ est strictement inférieur à α.
2. En appliquant le théorème 4 de Ketonen et Solovay, on décompose l'inégalite $m(h) < \alpha$ en une suite d'applications de la fonction $\{\_\}(i)$ pour un certain i. Mais par le lemme 2, à cette suite s'associe une bataille transformant l'hydre $h = \iota(\alpha)$ en l'hydre $\iota(m(h))$.

Nous pouvons donc construire une instance de `TooSimple` pour l'ordre sur $\{\beta : T1 \mid \beta < \alpha\}$ et par conséquent avons prouvé $P(\alpha)$.

3.3.4 Remarques

L'analyse de cette preuve montre qu'aucune hypothèse supplémentaire sur le variant m n'est nécessaire. Seule la structure (alternance d'ordinaux successeurs et de limites) des ordinaux inférieurs à ϵ_0, ainsi que la relation entre suites canoniques et rounds de bataille, sont utilisées.

Remarquons également le caractère « constructif » de cette preuve : étant donnés un ordinal α et un hypothétique variant m à valeurs dans $[0, \alpha[$, on peut construire une bataille où l'hydre $\iota(m(\alpha))$ se transforme en elle-même au bout d'un nombre non nul de reprises.

Comme pour la preuve de la section 3.1, le choix d'un facteur de réplication dans une bataille est contrôlé par le traitement des ordinaux limites. Pour un ordinal β donné, la preuve commence par l'introduction d'un ordinal $\gamma < \beta$ quelconque, et introduit une dépendance entre γ et un approximant $\{\beta\}(i)$ strictement supérieur à γ. C'est cet indice i qui sera utilisé dans le round transformant $\iota(\beta)$ en $\iota(\{\beta\}(i))$. Notre preuve construit bien une bataille d'hydre, mais sans qu'on puisse ajouter une contrainte de la forme « le facteur de réplication est toujours 42 », ou « le facteur de réplication augmente de 1 à chaque tour ».

Dans notre développement, la preuve du lemme 1 et du théorème 4 de [9] a demandé 1803 lignes de vernaculaire Coq, pour 3 pages de discours mathématique. Nous savons gré à J. Ketonen et R. Solovay d'avoir proposé un découpage en lemmes très détaillé qu'il a suffi de suivre. Beaucoup de nos lemmes sont juste des reformulations directes des énoncés de [9]. L'application des théorèmes de Ketonen et Solovay au problème spécifique des hydres nous a demandé 740 lignes supplémentaires.

Ce ne sont cependant pas des traductions mot à mot. Les auteurs de cet article sont des mathématiciens, raisonnant en termes d'ensembles. En revanche, nous travaillons sur une représentation d'ordinaux par des termes finis, et avons privilégié l'écriture de fonctions calculables par `Compute` ou faciles à extraire. Citons à titre d'exemple les suites canoniques, et la fonction qui associe à un ordinal limite λ et un ordinal $\beta < \lambda$, un indice i tel que $\beta < \{\lambda\}(i)$. Le fait de pouvoir tester ces fonctions nous a permis d'avoir une vision plus « concrète » des objets mathématiques impliqués.

Certains lemmes et définitions de [9] ont disparu de notre adaptation. Par exemple, les auteurs définissent une relation « α concorde avec β » utilisée comme condition de certains lemmes intermédiaires, et définie par « α et β sont en forme normale de Cantor, et tous les termes de β sont inférieurs ou égaux aux termes de α ». En fait, les schémas de récurrence créés auparavant dans [5] génèrent des buts où cette condition est automatiquement remplie. Nous avons donc une preuve complète du lemme 1 et du théorème 4 fortement inspirée de [9], mais compatible avec « l'esprit Coq ».

4 Perspectives

4.1 Représentations des ordinaux dans un assistant de preuve

Les ordinaux sont utiles pour la construction de preuves formelles de totalité de fonctions récursives. Parmi les formalisations existantes, on peut considérer l'utiisation de *notations d'ordinaux* représentant des ordinaux dénombrables comme ϵ_0, Γ_0 sous forme de termes finis permettant le calcul effectif d'expressions arithmétiques et la comparaison d'ordinaux par une fonction booléenne [11, 5, 8].

L'implémentation de ces opérations peut alors être validée en prouvant leur correction par rapport à une définition mathématique : définition axiomatique [4, 8] ou définition des ordinaux comme quotients de bons ordres pour la relation d'isomorphismes d'ordres [12].

Il serait intéressant, comme cela a été fait pour l'arithmétique dans la bibliothèque standard, de rendre compatibles tous ces modules en unifiant les noms de théorèmes, et tant que possible, en définissant des fonctions de traduction d'un formalisme vers l'autre. Par exemple, nous avons défini une « coercion » des ordinaux en forme normale de Cantor vers l'ensemble des ordinaux dénombrables au sens de Schütte. On peut alors disposer de plusieurs vues d'un même objet.

4.2 Étude des stratégies de l'hydre

Rappelons que les preuves des sections 3.1 et 3.3 construisent des batailles d'hydre dans lesquelles le facteur de réplication est quelconque à chaque étape : l'hydre doit pouvoir choisir ce nombre à chaque tour. Or, la lecture de l'article de Kirby et Paris suggère que nous pouvons prouver le théorème 3 en considérant des batailles où le nombre de réplications de l'hydre est *contrôlé*, le cas classique étant une augmentation de 1 à chaque reprise du combat. Il reste donc à prouver que, dans ce dernier cas, l'ordinal ϵ_0 reste le co-domaine de variant le plus simple pour la preuve de terminaison.

L'article de Ketonen et Solovay propose des outils mathématiques permettant d'étudier systématiquement les facteurs de réplications utilisés dans une bataille. Soient un ordinal $\alpha < \epsilon_0$ et une suite finie d'entiers strictement croissante $s = i_1 < i_2 < \cdots < i_n$. On calcule $\alpha_1 = \{\alpha\}(i_1), \ldots, \alpha_n = \{\alpha_{n-1}\}(i_n)$. La suite s est dite α-*grande* si $\alpha_n = 0$. Autrement dit, si l'hydre $\iota(\alpha)$ réagit avec les facteurs de réplication de la suite s et Hercule coupe toujours la tête la plus à droite parmi les plus basses [2], la bataille est terminée au bout de n reprises.

Notons que Ketonen et Solovay définissent plutôt une notion d'*ensemble* α-grand, mais dont l'énumération se fait toujours dans l'ordre croissant des éléments. Notre implémentation (encore très partielle) de cette notion en Coq a pu bénéficier des définitions et lemmes de la bibliothèque standard de Coq sur les listes triées. Dans ce cas encore, nos définitions peuvent s'écarter de celles de [9], mais les notions principales restent les mêmes.

Toujours en considérant la même stratégie d'Hercule, et si l'hydre $\iota(\alpha)$ réagit avec un facteur de réplication de i, puis $i+1, i+2, \ldots$, la fin de la bataille sera prévue pour le plus petit entier k tel que l'intervalle $[i, k]$ soit α-grand. La fonction F_α qui à i associe l'entier k est une fonction *à croissance rapide*. Les propriétés de ces fonctions sont étudiées dans [9] et servent à prouver le théorème 2 de [10] pour toute stratégie récursive de l'hydre. Notre projet est de prouver en Coq une version du théorème 3 pour cette stratégie, avec pour effet collatéral l'écriture de modules sur les « grands » ensembles et les fonctions à croissance rapide. Ces fonctions d'une complexité monstrueuse échappent totalement au test ; seule la preuve permet d'en capter les propriétés. L'article de S. Schmitz [14] présente un ensemble d'applications de ces fonctions à l'étude de la complexité.

2. Des expérimentations en OCaML nous ont persuadé que, dans la mesure ou l'hydre augmente strictement son nombre de réplications à chaque reprise, cette stratégie est la pire pour Hercule. Négliger les têtes les plus hautes les rend beaucoup plus dangereuses au fur et à mesure que le nombre de réplications augmente.

5 Conclusion

Les suites de Goodstein et les batailles d'hydre ont pour intérêt de pallier le caractère trop abstrait des preuves d'incomplétude à la Gödel. Montrer qu'un théorème donné n'est pas prouvable en arithmétique de Peano, ou nécessite l'ordinal ϵ_0 est plus « simple » que considérer la prouvabilité en général.

L'énoncé des deux théorèmes de Kirby et Paris — surtout le premier — est suffisamment simple pour être compris d'un large public. Nous avons utilisé les jeux d'hydre dans des cadres divers : l'opération *Maths en Jeans* pour un public lycéeen, un cours de programmation fonctionnelle en Licence d'Informatique. Par ailleurs, nombreux sont les sites offrant des animations de batailles d'hydre ; citons parmi ceux-ci la page d'Andrej Bauer [2]. Le premier théorème est déjà surprenant, et le second fournit des éléments d'analyse sur la difficulté de prouver. Ce dernier pourrait cependant ne pas étonner ceux/celles pour qui *toute* démonstration est difficile.

L'utilisation d'un système de notation d'ordinaux (forme normale de Cantor) nous a permis de rendre « concrètes » certaines notions liées aux ordinaux : comparaison, opérations arithmétiques, suites canoniques, implémentées sous forme de fonctions que l'utilisateur peut appliquer. Par exemple, si $\alpha < \lambda$ et λ est un ordinal limite, on peut définir une fonction qui calcule le i-ème ordinal strictement compris entre α et λ, pour toute valeur de i. La notion d'*infini potentiel* est bien capturée par la notion de fonction.

Enfin, l'adaptation du discours mathématique : omissions d'étapes "faciles", utilisation d'ensembles, etc., à l'univers de Coq : tactiques, types inductifs, etc., est en soi un sujet d'intérêt.

L'intérêt du travail présenté est profondément lié aux caractéristiques de Coq que nous souhaitons mettre en valeur : preuves par calcul, classes de types, etc. Cette notion de classe nous a permis de construire une famille de preuves indexée par un ordre bien fondé quelconque, et de raisonner sur la nécessaire complexité de cet ordre.

État actuel du développement Les preuves décrites dans cet article sont disponibles dans leur état actuel à l'adresse suivante : `http://www.labri.fr/perso/casteran/hydra-ludica/`.

Ce développement a été mis au point sous la version `V8.7` de Coq. Nous avons inclus une version modernisée de la contribution [5] ainsi que la présentation axiomatique des ordinaux dénombrables d'après Schütte [15]. Nous avons également débuté une formalisation des ensembles et suites α-grands. Une documentation de ce développement est en cours de rédaction et est disponible à l'adresse ci-dessus.

Remerciements L'auteur remercie vivement les relecteurs anonymes pour leurs nombreux commentaires sur une première version de cet article, ainsi que les membres de l'équipe « Méthodes Formelles » du LaBRI pour leurs remarques lors de présentations détaillées de ce travail.

Références

[1] C. Auger, Z. Bouzid, P. Courtieu, S. Tixeuil, and X. Urbain. Certified Impossibility Results for Byzantine-Tolerant Mobile Robots. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems*, volume 8255 of *LNCS*, page 15, Osaka, Japon, November 2013.

[2] Andrej Bauer. The hydra game. math.andrej.com/2008/02/02/the-hydra-game.

[3] Pierre Castéran and Vincent Filou. Tasks, types and tactics for local computation systems. *Stud. Inform. Univ.*, 9(1) :39–86, 2011.

[4] Pierre Castéran. Utilisation en Coq de l'opérateur de description. In *Actes des Journées Francophones des Langages Applicatifs*, 2007. http ://jfla.inria.fr/2007/actes/index.html.

[5] Pierre Castéran and Évelyne Contéjean. On ordinal notations. User Contributions to the Coq Proof Assistant, 2006.

[6] Pierre Courtieu, Lionel Rieg, Sébastien Tixeuil, and Xavier Urbain. Impossibility of gathering, a certification. *Inf. Process. Lett.*, 115(3) :447–452, 2015.

[7] Nachum Dershowitz and Georg Moser. The hydra battle revisited. In Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof : Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, pages 1–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[8] José Grimm. Implementation of three types of ordinals in Coq. Research Report RR-8407, INRIA, 2013.

[9] Jussi Ketonen and Robert Solovay. Rapidly growing Ramsey functions. *Annals of Mathematics*, 113(2) :267–314, 1981.

[10] Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14 :725–731, 1982.

[11] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic : Algorithms and mechanization. *Journal of Automated Reasoning*, 34(4) :387–423, May 2005.

[12] Michael Norrish and Brian Huffman. Ordinals in HOL : Transfinite arithmetic up to (and beyond) ω_1. In Sandrine Blazy and Christine Paulin-Mohring and David Pichardie, editor, *International Conference on Interactive Theorem Proving*, pages 133–146, Rennes, France, July 2013. Springer.

[13] Marcus Vinícius Midena Ramos, Ruy J. G. B. de Queiroz, Nelma Moreira, and José Carlos Bacelar Almeida. Formalization of the pumping lemma for context-free languages. *CoRR*, abs/1510.04748, 2015.

[14] Sylvain Schmitz. Complexity hierarchies beyond elementary. *CoRR*, abs/1312.5686, 2013.

[15] Kurt Schutte. *Proof theory / Translation from the German by J. N. Crossley.* Springer-Verlag Berlin ; New York, 1977.

[16] Will Sladek. The Termite and the Tower : Goodstein sequences and provability in PA. http ://-citeseerx.ist.psu.edu/showciting ?cid=337283, 2007.

Liquidity: OCaml pour la Blockchain

Çagdas Bozman[1], Mohamed Iguernlala[1], Michael Laporte[1], Fabrice Le Fessant[1,2], and Alain Mebsout[1]

[1] OCamlPro SAS – Gif-sur-Yvette F-91190
[2] Inria Paris – 2 Rue Simone IFF, 75012 Paris

<prenom>.<nom>@ocamlpro.com

Résumé

Cet article présente le langage de smart-contract Liquidity, de sa conception à son implémentation. Les blockchains récentes comme Ethereum et Tezos offrent la possibilité aux utilisateurs d'écrire des contrats, le plus souvent financiers, sous forme de programmes exécutés directement par les nœuds de la blockchain. Ces programmes étant souvent utilisés pour gérer des sommes importantes d'argent, leur nature immuable les rend critiques. Liquidity est un langage lisible, purement fonctionnel, et dont la syntaxe est très proche de OCaml. Les programmes Liquidity sont compilés vers le langage à pile (Michelson) de la blockchain Tezos. Liquidity vient aussi avec une version en ligne, "Try Liquidity" [1] qui permet de développer interactivement ses propres contrats dans son navigateur, de les compiler et de les déployer sur le réseau alphanet de Tezos. En outre, Liquidity dispose aussi d'un décompilateur capable de traduire un programme Michelson en Liquidity.

1 Introduction

Bitcoin, apparue en 2008 [7], est la première crypto-monnaie basée sur l'utilisation d'une blockchain. La blockchain est une structure de données distribuée fonctionnelle : c'est une liste de blocs, à laquelle on ne peut qu'ajouter de nouveaux blocs. Les ajouts se font en concurrence, et les participants choisissent la chaîne qui maximise un critère, parmi les différentes chaînes qu'ils voient sur le réseau. Pour Bitcoin, le critère est la puissance de calcul consommée, par l'opération de minage, lors de la signature d'un bloc. La maximisation de ce critère permet de garantir qu'après un certain temps, tous les participants s'accordent sur le préfixe de la chaîne et qu'un bloc dans ce préfixe est devenu irrévocable. Bitcoin permet d'utiliser ce système pour enregistrer des transactions sur la chaîne, consistant quasi uniquement en des transferts de crypto-monnaie entre des adresses . Ethereum, apparue en 2014 [3], a pour sa part révolutionné l'univers des crypto-monnaies en introduisant la possibilité d'exécuter des programmes turing-complets, manipulant un stockage propre limité, pour effectuer les transferts entre adresses. Ces *smart contracts* ont éveillé l'intérêt des acteurs financiers et industriels pour les blockchains, par leur supposée capacité à remplacer, en une seule entité infalsifiable, à la fois les contrats juridiques et leurs exécutants. En effet, une fois un contrat légal traduit en smart contract déployé sur une blockchain, ce smart contract s'exécute sans possibilité pour les partenaires de modifier son comportement en dehors du code initial, ce qu'on traduit par l'avènement du "code is law" (ou *code fait loi*).

Tezos, dont l'idée est apparue en 2014 [6], propose de répondre à deux problèmes supposés des crypto-monnaies : d'une part, il permet de s'auto-modifier par référendum, pour répondre au problème du *fork* ; d'autre part, il met l'accent sur l'utilisation des méthodes formelles, pour clarifier et sécuriser les smarts contracts. Bien que peu connus à l'époque, ces deux problèmes

1. http://www.liquidity-lang.org/

vont montrer leur importance en 2016 avec l'accident de *TheDAO* : un pirate utilise une faille dans un smart contrat sur Ethereum pour récolter 50 millions de dollars en quelques heures. La communauté Ethereum décide alors de modifier la blockchain pour annuler ce piratage : refusant cette remise en question du "code is law", une petite partie de la communauté rejette cette modification et maintient l'ancienne version de la blockchain en fonctionnement, c'est le premier fork important des blockchains.

OCamlPro a développé en OCaml le prototype de Tezos de mars 2014 jusqu'à l'ICO (*Initial Coin Offering*) de juillet 2017. Le développement a ensuite été progressivement repris par la Fondation Tezos pour pouvoir lancer le réseau au cours de l'année 2018. Tezos est la seule blockchain à utiliser un langage de smart contract statiquement typé, fonctionnel, avec une sémantique formelle et un interprète validé par l'utilisation des GADTs. Ce langage à pile, appelé *Michelson*, est relativement difficile à utiliser directement, l'absence de variables nécessitant de manipuler directement la pile. Pour cette raison, nous avons développé, dès juin 2017, un langage de plus haut niveau, *Liquidity*, intégrant le système de types de Michelson dans un sous ensemble du langage OCaml.

Outre le compilateur qui permet de compiler le code Liquidity vers Michelson, nous avons aussi développé un décompilateur, permettant de retrouver, à partir du code Michelson, un contrat en Liquidity, beaucoup plus facile à interpréter. Cet outil revêt une certaine importance, dans la mesure où les smart contracts qui seront stockés sur la blockchain le seront en Michelson, et qu'il pourra être utile pour les utilisateurs de comprendre et vérifier leur fonctionnement.

Enfin, afin de permettre une utilisation aisée de Liquidity, nous avons développé une application Web permettant d'éditer des contrats, soit en Liquidity, soit en Michelson, puis de respectivement les compiler ou décompiler. L'application permet aussi d'exécuter le contrat sur la blockchain et d'obtenir un résultat. Elle devrait aussi permettre, dans les prochains mois, de directement déployer un contrat sur la blockchain.

Cet article présente l'ensemble de ces outils : la section 2 introduit les principes de base des smart contracts ; la section 3 décrit le langage cible, Michelson, tandis que la section 4 présente notre langage, Liquidity ; la section 5 décrit le processus de compilation de Liquidity vers Michelson, tandis que la section 6 décrit le processus inverse de décompilation depuis Michelson ; enfin, la section 7 présente l'application Web d'édition et d'exécution de contrats.

2 Principes des smart contracts

Les utilisateurs d'une blockchain possèdent une *clé privée* qui leur est propre. Celle-ci leur permet de créer des *comptes*, ou *adresses*, auxquels sont associés un *crédit*, ou *balance*, en cryptomonnaie (ou *jetons*). Seul l'utilisateur d'un compte peut l'utiliser, en signant les opérations dessus avec sa clé privée.

Une adresse peut être associée (ou non) à du code, exécuté à chaque transfert de jetons vers cette adresse. Lorsqu'une adresse a du code associé, on appelle le compte correspondant un *smart contract*. Un smart contract peut être vu comme un objet, au sens de la programmation objet, c'est à dire un programme accessible via des méthodes (points d'entrées) manipulant un état interne, son *stockage*. Un smart contract peut lui-même appeler les méthodes d'un autre smart contract. Dans une blockchain à smart contract, comme Ethereum et Tezos, l'appel de méthode est le seul moyen de transférer de l'argent entre comptes : à chaque appel de méthode est associé un *montant* à transférer depuis l'appelant vers l'appelé. Les adresses sans code sont considérées comme possédant une méthode vide, ne faisant rien.

La blockchain maintient un état persistent qui associe, à chaque adresse, sa balance et l'état interne (stockage) de son code. Pour *déployer* un nouveau smart contract, un utilisateur doit

fournir son code, sa balance initiale et initialiser son stockage. Un utilisateur peut ensuite appeler une méthode de ce smart contract en fournissant un montant et des arguments. L'évaluation de cette méthode (et transitivement des méthodes qu'elle appelle) s'effectue séquentiellement au sein d'une seule *transaction*, qui modifie atomiquement, si elle est acceptée, les balances et les stockages de tous les contrats exécutés. Pour être acceptées, les transactions sont placées dans un *bloc*, qui est ajouté sur la blockchain. A n'importe quel moment, l'état de la blockchain peut être recalculé en exécutant l'ensemble des transactions des blocs qui la constituent depuis sa génèse.

Généralement, le programme d'un *smart contract* implémente une logique particulière qui régit les interactions et échanges entre différentes entités. Pour cela, la plupart des langages de programmation de smart contracts fournissent des structures de contrôle classiques ainsi que certaines primitives liées à l'environnement d'exécution spécifique à une blockchain (*e.g.* primitives cryptographiques, introspection de la blockchain, appels de méthodes).

Pour garantir l'intégrité de la blockchain, l'ensemble des nœuds du réseau valident chaque bloc en exécutant chaque transaction. Le passage à l'échelle de la blockchain impose donc de limiter le temps d'exécution des transactions. Pour cela, l'exécution de chaque opération d'une méthode consomme du *carburant* (le *gas*), et l'appelant (l'utilisateur, ou un contrat en appelant un autre) doit fournir, en plus du montant de l'appel, une certaine quantité de ce carburant (en crypto-monnaie). L'épuisement du carburant provoque une erreur, qui annule l'exécution de la méthode (l'état interne de l'objet revient à l'état initial).

Sur Ethereum, la gestion d'une erreur (épuisement de carburant, débordement de pile, division par zéro, etc.) dépend du type d'appel effectué sur la méthode : certains appels négligent complètement les erreurs, et continuent l'exécution dans l'appelant, d'autres la propage comme une exception ou permettent de la traiter. La mauvaise gestion des erreurs (erreurs négligées, mais aussi non erreur lors des débordements sur les entiers, par exemple) est à l'origine de très nombreux bugs dans les smarts contracts, exploités par les pirates.

Dans le cas des blockchains *publiques* comme Ethereum ou Tezos, le code et stockage d'un smart contract déployé est accessible à tout un chacun. C'est bien entendu une forte garantie de transparence mais cela donne des motivations pécuniaires importantes aux acteurs malveillants pour chercher et exploiter les bogues des smart contracts. Il apparaît donc primordial pour un tel langage de programmation de fournir des garanties de sécurité et de sûreté les plus élevées possibles tout en restant accessible. C'est un des objectifs recherchés par Michelson et Liquidity.

3 Le langage cible, Michelson

Michelson est un langage à pile [8], fonctionnel et statiquement typé. Il fournit des types de base, comme les chaînes de caractères, les booléens, les entiers et les naturels non bornés, les listes, les paires, les types *option*, les type *union* (de deux types), les ensembles, et les tableaux associatifs (*map*), mais aussi des types domaines dépendants comme les types des montants en Tez, des clefs cryptographiques, des dates, *etc.* Un programme Michelson consiste en une séquence structurée d'instructions, chacune d'elles opérant sur la pile (par réécriture). Le programme reçoit en argument un paramètre ainsi qu'un stockage et renvoie un résultat et un nouveau stockage. Les programmes Michelson peuvent échouer à l'exécution par l'instruction `FAIL`, ou une autre erreur (appel d'un contrat échouant, épuisement du gas, *etc.*), mais la plupart des instructions qui pourraient échouer retournent une option à la place (`EDIV` par exemple).

L'exemple suivant est un smart contract qui implémente un système de vote sur la blockchain. Le stockage consiste en une map qui associe les options de vote (sous forme de chaînes de caractères) à un entier qui comptabilise le nombre de votes. Une transaction vers ce contrat doit

s'effectuer avec un montant (accessible avec l'instruction `AMOUNT`) supérieur ou égal à 5 tezzies et un paramètre qui correspond à une des options possibles de vote. Si une de ces conditions n'est pas respectée, l'exécution et donc la transaction échouent. Sinon le programme récupère l'ancien nombre de votes dans le stockage, et l'incrémente. Il renvoie la paire composée de la valeur `Unit` et de la nouvelle map mise à jour (son stockage).

```
parameter string;
storage (map string int);
return unit;
code
  {
    PUSH tez "5.00"; AMOUNT; COMPARE; LT; # Est-ce que amount < 5 tz ?
    IF
      { FAIL }
      {
        DUP; DUP; CAR; DIP { CDR }; GET; # GET parameter storage
        IF_NONE # Est-ce un vote invalide ?
          { FAIL }
          { # Some x, x dans la pile
            PUSH int 1; ADD; SOME; # Some (x + 1)
            DIP { DUP; CAR; DIP { CDR } }; SWAP; UPDATE;
            # UPDATE parameter (Some (x + 1)) storage
            PUSH unit Unit; PAIR; # Pair Unit nouveau_stockage
          }
      };
  }
```

Michelson possède plusieurs caractéristiques assez spécifiques :

— Le typage d'un programme Michelson s'effectue par propagation des types, et non à la Milner. Les types polymorphes sont interdits, et des contraintes de types sont obligatoires quand un type est ambigü (liste vide, etc.) ;

— Les fonctions (*lambda*) ne sont pas des fermetures, i.e. elles doivent avoir un environnement vide. Ainsi, une fonction passée à un autre contrat en argument agit de façon purement fonctionnelle, n'accédant qu'à l'environnement du nouveau contrat ;

— L'appel de méthode se fait par l'instruction `TRANSFER_TOKENS` : celle-ci requiert une pile vide au delà de ses arguments. Celle-ci prend en argument le stockage courant, le sauvegarde avant l'appel, puis retourne le stockage après l'appel, avec le résultat. Cela oblige le développeur à stocker l'état courant dans le contexte, tout en ayant conscience qu'un appel réentrant peut avoir lieu dans le contrat appelé.

Pour mieux comprendre la sémantique de Michelson, cette section présente quelques unes des règles de réécriture de la pile en figure 1. L'interprète de référence implémenté dans Tezos utilise des GADTs pour garantir que seuls les programmes Michelson bien typés sont acceptés et que toutes les transformations donnent des programmes bien typés. On utilise ici les notations $S : \Gamma$ pour dénoter une pile S de type Γ, $x : \tau$ pour dénoter un élément de la pile x de type τ. $x :: S$ est la pile S avec l'élément x ajouté en tête. La séquence est noté par { `A; B` } où `A` et `B` sont des instructions Michelson. On note respectivement, la réécriture conditionnelle de pile et la réécriture de programmes par

$$\mathtt{I} \mid S \quad \rightsquigarrow_c \quad S' \qquad \text{et} \qquad \mathtt{P} \mid S \quad \dashrightarrow \quad \mathtt{P}' \mid S'$$

où c est une condition booléenne. L'instruction `I` réécrit la pile S en S' ssi la condition c s'évalue à vrai, et un programme `P` dans une pile S est réécrit en un programme `P'` dans une pile S'.

$$
\begin{array}{rlll}
\texttt{PUSH}\ \tau\ x & \mid S & \rightsquigarrow & x :: S \\
\texttt{DROP} & \mid x :: S & \rightsquigarrow & S \\
\texttt{DUP} & \mid x :: S & \rightsquigarrow & x :: x :: S \\
\texttt{FAIL} & \mid S & \rightsquigarrow & [\mathit{fail}] \\
\texttt{I} & \mid [\mathit{fail}] & \rightsquigarrow & [\mathit{fail}] \\
\texttt{EQ} & \mid (0 : \mathit{int}) :: S & \rightsquigarrow & \texttt{True} :: S \\
\texttt{EQ} & \mid (x : \mathit{int}) :: S & \rightsquigarrow_{x \neq 0} & \texttt{False} :: S \\
\texttt{ADD} & \mid (x : \mathit{int}) :: (y : \mathit{int}) :: S & \rightsquigarrow & (x + y) :: S \\
\texttt{PAIR} & \mid x :: y :: S & \rightsquigarrow & (\texttt{Pair}\ x\ y) :: S \\
\texttt{CAR} & \mid (\texttt{Pair}\ x\ y) :: S & \rightsquigarrow & x :: S \\
\texttt{CDR} & \mid (\texttt{Pair}\ x\ y) :: S & \rightsquigarrow & y :: S \\
\texttt{EXEC} & \mid (f : \mathit{lambda}\ \tau\ \rho) :: (x : \tau) :: S & \rightsquigarrow & (f\ x) :: S \\
\texttt{SOME} & \mid x :: S & \rightsquigarrow & (\texttt{Some}\ x) :: S \\
\texttt{GET} & \mid (k : \tau) :: (m : \mathit{map}\ \tau\ \rho) :: S & \rightsquigarrow_{k \in m} & (\texttt{Some}\ m[k]) :: S \\
\texttt{GET} & \mid (k : \tau) :: (m : \mathit{map}\ \tau\ \rho) :: S & \rightsquigarrow_{k \notin m} & \texttt{None} :: S \\
\texttt{UPDATE} & \mid (k : \tau) :: (\texttt{Some}\ (x : \tau)) :: & & \\
 & \quad (m : \mathit{map}\ \tau\ \rho) :: S & \rightsquigarrow & m[k \leftarrow x] :: S \\
\texttt{UPDATE} & \mid (k : \tau) :: \texttt{None} :: (m : \mathit{map}\ \tau\ \rho) :: S & \rightsquigarrow & (m \backslash k) :: S \\
\texttt{TRANSFER\_TOKENS} & \mid (p : \alpha) :: (a :: \mathit{tez}) :: & & \\
 & \quad (c : \mathit{contract}\ \alpha\ \beta) :: (s : \sigma) :: [] & \rightsquigarrow & (r : \beta) :: (s' : \sigma) :: []
\end{array}
$$

où r est le résultat de l'appel du contrat c avec le paramètre p et le montant a, s le stockage courant, et s' le stockage courant potentiellement modifié par l'appel

$$
\begin{array}{rlll}
\{\ \texttt{I}\ \} \mid S & \texttt{-->}\ \{\} \mid S' & \text{ssi} & \texttt{I} \mid S \rightsquigarrow S' \\
\texttt{A ; B} \mid S & \texttt{-->}\ \texttt{B} \mid S' & \text{ssi} & \texttt{A} \mid S \texttt{ --> } \{\} \mid S' \\
\{\texttt{DIP P}\} \mid x :: S & \texttt{-->}\ \{\} \mid x :: S' & \text{ssi} & \texttt{P} \mid S \texttt{ --> } \{\} \mid S' \\
\{\texttt{IF T E}\} \mid \texttt{True} :: S & \texttt{-->}\ \texttt{T} \mid S & & \\
\{\texttt{IF T E}\} \mid \texttt{False} :: S & \texttt{-->}\ \texttt{E} \mid S & & \\
\{\texttt{IF\_NONE T E}\} \mid \texttt{None} :: S & \texttt{-->}\ \texttt{T} \mid S & & \\
\{\texttt{IF\_NONE T E}\} \mid (\texttt{Some}\ x) :: S & \texttt{-->}\ \texttt{E} \mid x :: S & & \\
\{\texttt{LAMBDA}\ \tau\ \rho\ \texttt{P}\} \mid S & \texttt{-->}\ \{\} \mid (\lambda x.\, p) :: S & \text{ssi} & \texttt{P} \mid [x : \tau] \texttt{ --> } \{\} \mid [p : \rho]
\end{array}
$$

FIGURE 1 – Interprétation d'un fragment de Michelson : règles de réécriture

4 Le langage Liquidity

Liquidity est un langage fonctionnel statiquement typé, qui compile vers le langage à pile Michelson. Sa syntaxe est un sous-ensemble de OCaml, et sa sémantique est donnée par son schéma de compilation vers le langage cible (*c.f.* Section 5). En faisant le choix de rester proche de Michelson et dans l'esprit, tout en offrant des constructions de plus haut niveau, Liquidity permet d'écrire facilement des smart contracts lisibles avec les mêmes garanties de sûreté que celles offertes par Michelson. En particulier, il nous a semblé important de conserver l'aspect purement fonctionnel du langage (à l'inverse des langages Lamtez [5] et Fi [1]) pour que la simple lecture d'un contrat ne soit pas déjà obscurcie par des effets et des états globaux. En outre, la syntaxe OCaml fait de Liquidity un outil *immédiatement* accessible à ceux qui connaissent déjà le langage, et son envergure limitée rend la phase d'apprentissage plutôt aisée.

L'exemple en figure 2 est une version Liquidity du contrat de vote présenté en Section 3. Son

fonctionnement apparaît clairement pour quiconque a déjà écrit dans un langage à la ML. Un

```
[%%version 0.11]
type votes = (string, int) map

let%entry main (parameter : string) (storage : votes) : unit * votes =
  let amount = Current.amount () in
  if amount < 5.00tz then
    Current.fail ()
  else
    match Map.find parameter storage with
    | None -> Current.fail ()
    | Some x ->
        (* incremente le nombre de votes pour l'option passee en parametre *)
        let storage = Map.add parameter (x + 1) storage in
        ( (), storage )
```

FIGURE 2 – Contrat de vote en Liquidity

contrat Liquidity commence par une méta-information optionnelle de version. Le compilateur peut rejeter le programme s'il est écrit dans une version trop vieille du langage ou si lui-même n'est pas assez récent. Viennent ensuite un ensemble de déclarations de types et de fonctions globales. Ici, on définit une abréviation de type `votes` pour une map des chaînes de caractères vers les entiers. C'est la structure qui nous sert à maintenir les comptes totaux des votes.

Le point d'entrée du programme est une fonction `main` définie avec une annotation spéciale `let%entry`. Elle prend en argument un paramètre d'appel (`parameter`) et un stockage (`storage`) et renvoie une paire dont le premier élément est le résultat de l'appel et le second élément est une version potentiellement modifiée du stockage.

Le programme ci-dessus définit une variable locale `amount` qui contient le montant de la transaction qui a généré l'appel au contrat. Il vérifie que le montant est bien supérieur à 5 tezzies. Si ce n'est pas le cas, il appelle la fonction prédéfinie `Current.fail` qui interrompt toute exécution et réinitialise les stockages et les crédits, à leurs valeurs respectives avant exécution de la transaction. Ensuite, le programme récupère le nombre de votes déjà comptabilisés pour l'option contenue en paramètre. Cette opération est effectuée en appelant `Map.find` sur la map contenue dans le stockage. Si l'option de vote n'est pas possible (*i.e.* il n'y a pas d'association dans la map), l'exécution échoue. Sinon, la valeur courante est liée au nom `x`. Le stockage est enfin mis à jour en incrémentant la valeur associée à l'option de vote du paramètre. La fonction prédéfinie `Map.add` effectue cette opération en renvoyant une nouvelle map modifiée. Le programme se termine (dans le cas normal) sur sa dernière expression. La valeur de retour est une paire contenant `()`, car le contrat ne fait que modifier son stockage, et le stockage lui même.

4.1 Spécificités

Liquidity supporte les mêmes types de base que Michelson, à savoir les booléens, les entiers non bornés, les listes, maps et ensembles, les types option, union, les montants, *etc.* Mais il fournit en outre des types enregistrements, des produits, des types somme ainsi que des alias de types. Il existe un type somme, `variant`, prédéfini ayant pour constructeurs `Left` et `Right`.

```
type ('a, 'b) variant = Left of 'a | Right of 'b
```

Par exemple, on peut écrire :

```
type montype = (int * string) list
type t = A of montype | B | C of (bool * tez * unit)
type r = { x : int; y : t }
```

Les types doivent être non récursifs, et les types imbriqués sont autorisés bien que pour l'instant le pattern matching ne se fasse que sur les constructeurs de tête.

Les constantes en Liquidity s'écrivent :

- `true` et `false` pour les booléens (`bool`)
- les entiers non bornés pour `int`, *e.g.* `123456899999`
- les entiers positifs non bornés suivis du suffixe `p` pour `nat`, *e.g.* `10p`
- des nombres à virgule fixe non bornés pour les montants suivis du suffixe `tz`, *e.g.* `1.00tz`, `99999.08tz`
- une notation identique à OCaml pour les listes constantes, *e.g.* `[1; 2; 3]`
- avec un constructeur `Set` particulier pour les ensembles, *e.g.* `Set ["a"; "c"; "b"]`
- avec un constructeur `Map` particulier pour les map, suivie d'une liste d'association *e.g.* `Map [1, "1"; 2, "2"; 3, "3"]`
- avec une annotation de type pour les listes, maps et ensembles vides, *e.g.* `([]: int list)`, `(Set: string set)`, `(Map: (int, string) map)`
- au format ISO8601 pour les dates, *e.g.* `2018-01-27T10:01:00+01:00`
- directement au format base 58, pour les clefs et les hashes de clefs, *e.g.* `tz1hxLtJnSYCVabeGio3M87Yp8ChLF9LFmCM`.

Un appel de contrat avec `Contract.call` crée une nouvelle portée, les variables définies avant l'appel ne sont plus accessibles après. Cette limitation, héritée de Michelson, pourrait être levée en s'autorisant à sauvegarder les variables visibles dans le stockage.

5 Compilation vers Michelson

5.1 Simplifications et encodages

Comme Liquidity est beaucoup plus riche que Michelson, certains types et certaines constructions doivent être simplifiées ou encodées. Les types *enregistrement* sont traduits en types tuples ayant autant de composants que de champs. Les types tuples sont eux mêmes traduits en types paires associées à droite. Dans l'exemple suivant le type `t1` est traduit en `t1'`. L'accès à une composante d'un tuple `x` en Liquidity se fait avec la notation `x.(`i`)`, où i est une constante entière positive. L'accès aux champs d'une expression d'un type enregistrement est traduit par l'accès aux composants du tuple correspondant. Les types *somme* sont traduits en types variant. Le type variant étant lui même traduit vers le type `or` de Michelson. Ci-dessous, le type `t2` est traduit en `t2'` par Liquidity. Le pattern matching sur les expressions de type somme est traduit vers un pattern matching sur des expressions de type variant. Un exemple de pattern matching et sa traduction est donné ci-dessous.

```
type t1 = { a: int; b: string; c: bool}

type t1' = (int * string * bool)
```

```
type t2 = A of int | B of string | C

type t2' = (int, (string, unit) variant) variant
```

```
match x with
| A i -> qqch1(i)
| B s -> qqch2(s)
| C -> qqch3

match x with
| Left i -> qqch1(i)
| Right r -> match r  with
             | Left s -> qqch2(s)
             | Right _ -> qqch3
```

Liquidity supporte les fermetures alors que Michelson n'autorise que les lambdas purs. Les fermetures sont *lambda-liftées*, *i.e.* encodées comme des paires, de premier élément un lambda et de second élément son environnement. Le lambda prend lui même en argument une paire composée de l'argument de la fermeture et de son environnement. Des transformations sont aussi effectuées sur les appels de fonctions prédéfinies qui prennent des lambdas en argument pour autoriser des fermetures.

5.2 Schéma de compilation

On note par $\Gamma, [\![x]\!]_d \vdash X \uparrow^t$, la compilation de l'instruction Liquidity x, dans l'environnement Γ. Γ est une map qui associe des noms de variables à une position dans la pile. L'algorithme de compilation maintient également la taille de la pile courante (à la compilation de l'instruction x), dénotée par d dans l'expression précédente. Une version non déterministe du schéma de compilation est présentée en figure 3, la version implémentée dans le compilateur Liquidity de référence étant, elle, une version déterminisée. Le résultat de la compilation de x, est une instruction (ou séquence d'instructions) Michelson X ainsi qu'une information booléenne de transfert t. En effet, l'instruction `Contract.call` (ou `TRANSFER_TOKENS` en Michelson) travaillant sur une pile vide, le compilateur vide la pile avant cet appel. Cependant, les différentes branches d'un programme Michelson doivent avoir le même type de pile. Pour cette raison, il est nécessaire de maintenir cette information de transfert afin de vider les piles dans certaines parties du programme.

Par exemple, la compilation d'une constante Liquidity consiste simplement à pousser cette constante sur la pile, avec l'instruction Michelson `PUSH`. Pour gérer les variables de manière simple, il suffit de regarder dans l'environnement Γ l'indice associé à cette dernière. Ensuite, l'instruction $\texttt{D(U)}^i\texttt{P}$, met une copie de l'élément se trouvant dans la pile à la profondeur i, au dessus de la pile. Les variables sont ajoutées à Γ avec l'instruction Liquidity `let ... in` ou bien par toute instruction qui lie un nouveau symbole, comme `fun` par exemple.

Certaines des règles ont des parties annotées avec $?_b$. Ce suffixe dénote une réinitialisation ou un effacement. En particulier :

— Pour les ensembles, $\Gamma?_b$ est $\emptyset$ si b s'évalue à $\bot$, et Γ sinon.
— Pour les entiers, $d?_b$ est 0 si b s'évalue à $\bot$, et d sinon.
— Pour les instructions, $(X)?_b$ est $\{\}$ si b s'évalue à $\bot$, et X sinon.

5.3 Optimisations

Des optimisations sont appliquées à différentes étapes de la compilation. Une analyse de dépendance, effectuée durant la phase de vérification et inférence des types, est utilisée pour

$$\textsc{Entry}\ \frac{\{\mathit{storage} \mapsto 0; \mathit{parameter} \mapsto 1\}, [\![e]\!]_d \vdash E \uparrow^t}{\emptyset, [\![\texttt{let\%entry main } \mathit{parameter}\ \mathit{storage} = e]\!]_0 \vdash \{\texttt{DUP}; \texttt{DIP}\{\texttt{CDR}\}; \texttt{CAR}; E\} \uparrow^t}$$

$$\textsc{Const}\ \frac{}{\Gamma, [\![c]\!]_d \vdash \{\texttt{PUSH } c.\tau\ c\} \uparrow^\bot} \qquad \textsc{Var}\ \frac{}{\Gamma + \{x \mapsto p\}, [\![x]\!]_d \vdash \{\texttt{D(U)}^{d-p}\texttt{P}\} \uparrow^\bot}$$

$$\textsc{Seq}\ \frac{\Gamma, [\![e_1]\!]_d \vdash E_1 \uparrow^{t_1} \qquad \Gamma, [\![e_2]\!]_d \vdash E_2 \uparrow^{t_2}}{\Gamma, [\![e_1; e_2]\!]_d \vdash \{E_1; \texttt{DROP}; E_2\} \uparrow^{t_1 \vee t_2}}$$

$$\textsc{Let}\ \frac{\Gamma, [\![e_1]\!]_d \vdash E_1 \uparrow^{t_1} \qquad \Gamma + \{x \mapsto d\}, [\![e_2]\!]_{d+1} \vdash E_2 \uparrow^{t_2}}{\Gamma, [\![\texttt{let } x\ =\ e_1\ \texttt{in } e_2]\!]_d \vdash \{E_1; E_2; (\texttt{DIP}\{\texttt{DROP}\})?_{t_2}\} \uparrow^{t_1 \vee t_2}}$$

$$\textsc{Lambda}\ \frac{\{x \mapsto 0\}, [\![e]\!]_1 \vdash E \uparrow^\bot}{\Gamma, [\![\texttt{fun } (x : \tau) : \rho\ \rightarrow\ e]\!]_d \vdash \{\texttt{LAMBDA } \tau\ \rho\ \{E; \texttt{DIP}\{\texttt{DROP}\}\}\} \uparrow^\bot}$$

$$\textsc{If}\ \frac{\Gamma, [\![c]\!]_d \vdash C \uparrow^{t_c} \qquad \Gamma?_{\neg t_c}, [\![e_1]\!]_{d?_{\neg t_c}} \vdash E_1 \uparrow^{t_1} \qquad \Gamma?_{\neg t_c}, [\![e_2]\!]_{d?_{\neg t_c}} \vdash E_2 \uparrow^{t_2}}{\Gamma, [\![\texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2]\!]_d \vdash \{C; \texttt{IF}\{e1; (\texttt{DIP}\{\texttt{DROP}\})?_{t_2}\}\{e2; (\texttt{DIP}\{\texttt{DROP}\})?_{t_1}\}\} \uparrow^{t_c \vee t_1 \vee t_2}}$$

$$\textsc{Apply}\ \frac{\Gamma, [\![f]\!]_d \vdash F \uparrow^\bot \qquad \Gamma, [\![x]\!]_{d+1} \vdash X \uparrow^\bot}{\Gamma, [\![f\ x]\!]_d \vdash \{F; X; \texttt{EXEC}\} \uparrow^\bot}$$

$$\textsc{Builtin}\ \frac{\Gamma, [\![x_1]\!]_{d+1} \vdash X_1 \uparrow^\bot \qquad \cdots \qquad \Gamma, [\![x_n]\!]_{d+n} \vdash X_n \uparrow^\bot}{\Gamma, [\![s\ x_1\ \ldots\ x_n]\!]_d \vdash \{X_n; \ldots; X_1; \mathit{prim}(s)\} \uparrow^\bot}$$

où $\mathit{prim}(s)$ est l'instruction Michelson correspondant au symbole Liquidity s

$$\textsc{MatchOption}\ \frac{\begin{array}{c}\Gamma, [\![a]\!]_d \vdash A \uparrow^{t_a} \qquad \Gamma?_{\neg t_a}, [\![e_1]\!]_{d?_{\neg t_a}} \vdash E_1 \uparrow^{t_1} \\ \Gamma?_{\neg t_a} + \{x \mapsto d?_{\neg t_a}\}, [\![e_2]\!]_{d?_{\neg t_a}+1} \vdash E_2 \uparrow^{t_2}\end{array}}{\begin{array}{l}\Gamma, [\![\texttt{match } a \texttt{ with None} \rightarrow e_1 \mid \texttt{Some } x \rightarrow e_2]\!]_d \vdash \\ \{A; \texttt{IF\_NONE}\{E_1; \mathit{none\_end}(t_1, t_2)\}\{E_2; \mathit{some\_end}(t_1, t_2)\}\} \uparrow^{t_a \vee t_1 \vee t_2}\end{array}}$$

$$\text{avec } \begin{cases} \mathit{some\_end}(\bot, \bot) = \{\texttt{DIP}\{\texttt{DROP}\}\} \\ \mathit{some\_end}(\top, \bot) = \{\texttt{DIP}\{(\texttt{DROP})^{d+1}\}\} \\ \mathit{some\_end}(\_, \top) = \{\} \end{cases} \text{ et } \begin{cases} \mathit{none\_end}(\bot, \top) = \{\texttt{DIP}\{(\texttt{DROP})^d\}\} \\ \mathit{none\_end}(\bot, \bot) = \{\} \\ \mathit{none\_end}(\top, \_) = \{\} \end{cases}$$

$$\textsc{Call}\ \frac{\begin{array}{c}\Gamma, [\![s]\!]_d \vdash S \uparrow^\bot \qquad \Gamma, [\![c]\!]_{d+1} \vdash C \uparrow^\bot \\ \Gamma, [\![a]\!]_{d+2} \vdash A \uparrow^\bot \qquad \Gamma, [\![p]\!]_{d+3} \vdash P \uparrow^\bot \qquad \{s' \mapsto 0, r \mapsto 1\}, [\![e]\!]_2 \vdash E \uparrow^t\end{array}}{\begin{array}{l}\Gamma, [\![\texttt{let } (r, s')\ =\ \texttt{Contract.call } c\ a\ s\ p \texttt{ in } e]\!]_d \vdash \\ \{S; C; A; P; \texttt{DIIIIP}\{\texttt{DROP}^d\}; \texttt{TRANSFER\_TOKENS}; E; (\texttt{DIP}\{\texttt{DROP}; \texttt{DROP}\})?_{\neg t}\} \uparrow^\top\end{array}}$$

FIGURE 3 – Règles de compilation pour un fragment de Liquidity vers Michelson

Code Michelson :

```
parameter bool;
return int;
storage int;
code {DUP; CAR;
      DIP { CDR; PUSH int 1 };  # stack is: parameter :: 1 :: storage
      IF # if parameter = true
         { DROP; DUP; }         # stack is storage :: storage
         { }                    # stack is 1 :: storage
         ;
      PAIR;
     }
```

Code Liquidity :

```
[%%version 0.12]
type storage = int
let%entry main (parameter : bool) (storage : storage) : (int * storage) =
    ((if parameter then storage else 1), storage)
```

FIGURE 4 – Un contrat travaillant par modification de la pile et sa décompilation en Liquidity.

inliner certaines variables (celles qui ne sont utilisées qu'une seule fois).

Après traduction vers Michelson, une seconde phase d'optimisation est appliquée pour tenter de réduire la taille du code généré. En effet, les nœuds Tezos n'acceptent pour l'instant que les smart contracts ayant une taille inférieure à une limite assez faible (< 16Ko).

6 Décompilation depuis Michelson

Bien que les types de Michelson soient de haut niveau par rapport à d'autres "bytecodes", il reste difficile pour un utilisateur de la blockchain de comprendre ce qu'un programme Michelson fait précisément. Or, selon l'idée que le "code is law", un utilisateur devrait pouvoir lire le programme pour comprendre la sémantique précise d'un contrat. Aussi, nous avons développé un décompilateur de Michelson vers Liquidity, qui permet d'obtenir une représentation plus facile à comprendre d'un programme de la blockchain.

La décompilation du code Michelson se décompose en plusieurs phases :

Chargement : Le Chargement consiste dans le parsing et la vérification du typage du code Michelson. Suite à cette phase, le code est sous la forme d'une S-expression.

Nettoyage : Le Nettoyage consiste à simplifier le code Michelson pour accélérer le traitement et simplifier les tâches suivantes. Cette phase consiste pour l'instant uniquement à supprimer les instructions dont la continuation est un échec (instruction `FAIL`).

Interprétation : L'interprétation consiste à exécuter symboliquement le code Michelson, en remplaçant chaque valeur placée dans la pile par le nœud d'un graphe contenant l'instruction qui l'a générée. Chaque nœud de ce graphe peut être vu comme une expression du programme cible, qui pourra être lié à un nom de variable, les arêtes vers ce nœud représentant les futures occurrences de cette variable.

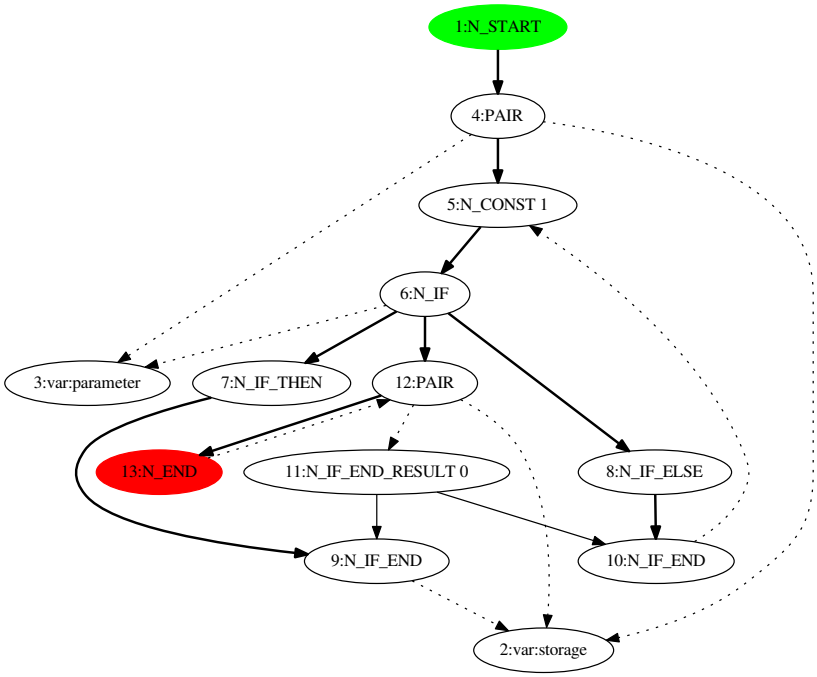


FIGURE 5 – Graphe du programme Michelson interprété. Les arêtes épaisses indiquent le chemin d'exécution, tandis que les arêtes en pointillés indiquent l'utilisation d'une valeur.

Décompilation : La Décompilation consiste à transformer le graphe issu de l'interprétation en un arbre de syntaxe Liquidity. Le programme obtenu est souvent assez peu lisible, toutes les expressions étant calculées dans des `let`.

Typage : Le Typage est la phase standard de typage d'un programme Liquidity. Cette phase permet à la fois de vérifier que le programme généré est correctement typé, et de calculer un certain nombre de propriétés sur l'arbre (nombre d'occurrences d'une variable, en particulier).

Simplification : La Simplification consiste à inliner les variables utilisées une seule fois, ce qui accroît considérablement la lisibilité du programme sur les petites expressions, et d'améliorer le nommage des variables (préfixe _ quand la variable n'est pas utilisée, etc.).

Génération : La Génération consiste à générer un AST OCaml à partir de l'AST Liquidity, puis à utiliser le joli-imprimeur du compilateur OCaml pour obtenir le source final.

Compiler puis décompiler un programme Liquidity donne rarement le même programme, mais peut converger rapidement. La principale différence est la suppression des types structures définis en Liquidity (enregistrements, types variants), le programme Michelson ne fournissant aucun moyen de retrouver les types d'origine. Les types enregistrements se retrouvent donc traduits sous forme de n-uplets. Les types variants sont traduits directement dans un type variant prédéfini (*c.f.* Section 4.1).

6.1 Exemple de décompilation

L'exemple de la Figure 4 illustre certaines difficultés du processus de décompilation : Liquidity est un langage purement fonctionnel, où chaque construction est une expression retournant une valeur ; Michelson travaille lui sur une pile, qu'il est impossible de concrétiser en Liquidity (les valeurs dans la pile n'ont pas le même type, contrairement à une liste). Dans cet exemple, en fonction du paramètre, le contrat retourne soit le contenu du storage, soit l'entier 1. Dans le code Michelson, le programmeur utilise l'instruction `IF`, mais chaque branche ne retourne pas une valeur, se contentant de modifier (ou pas) la pile.

La traduction en Liquidity contient elle aussi un `if`, mais celui-ci doit retourner une valeur. La Figure 5 présente le graphe obtenu lors de l'interprétation du programme Michelson. L'instruction `IF` est décomposée en plusieurs nœuds, mais ne contient au final aucune instruction restante : le résultat du `if` est en fait la différence entre les piles issues de l'interprétation de la branche `then` et de la branche `else`, et est indiqué par le nœud `N_IF_END_RESULT 0` (il pourrait y avoir plusieurs de ces nœuds avec des indices différents, le résultat du `if` serait alors un n-uplet correspondant à plusieurs emplacements modifiés sur la pile). Celui-ci pointe vers les deux nœuds `N_IF_END`, dont l'un pointe vers le stockage et l'autre vers la constante 1.

7 Try-Liquidity

Suite à notre expérience dans le domaine des applications Web en OCaml, dont les résultats les plus connus sont TryOCaml [2] et Learn OCaml [4], nous avons choisi de réaliser de la même manière, une plateforme totalement autonome, fonctionnant directement dans le navigateur grâce au compilateur js_of_ocaml [9].

Pour faciliter le développement en Liquidity, l'application contient deux éditeurs, l'un permettant d'éditer du code Liquidity et l'autre du code Michelson. Nous avons utilisé un binding de la bibliothèque JavaScript ACE [3], un éditeur de code générique, pour l'édition de code avec de la coloration syntaxique et de l'indentation, à l'aide de l'outil `ocp-indent`. Ces deux éditeurs sont mis à jour automatiquement dès que l'utilisateur compile ou décompile. Lorsqu'une des zones d'édition n'est plus synchronisée, ceci est signalé à l'utilisateur pour éviter une confusion entre les deux programmes.

Enfin, l'application permet d'exécuter le contrat Michelson via un nœud Tezos et d'obtenir un résultat. Elle permettra, dans les prochains mois, de directement déployer un contrat sur la blockchain. La vocation de Try-Liquidity est de fournir à terme un éditeur en ligne similaire à Remix pour Solidity [2].

7.1 Editeurs Liquidity et Michelson

L'interface d'édition de smart contract Liquidity présentée à la Figure 6, contient un menu latéral avec l'accès aux éditeurs Liquidity et Michelson, ainsi que les pages vers les documentations respectives des deux langages. Dans la zone centrale, les deux éditeurs sont deux instances de la bibliothèque JavaScript ACE personnalisés avec des caractéristiques propres à chacun des langages.

Pour Liquidity, nous utilisons `ocp-indent`, un outil d'indentation de code OCaml développé par OCamlPro. Tout comme pour la plateforme Learn OCaml, nous utilisons cet outil pour l'indentation automatique du code Liquidity. De plus, nous profitons de ce processus pour

2. https://try.ocamlpro.com/
3. https://ace.c9.io/

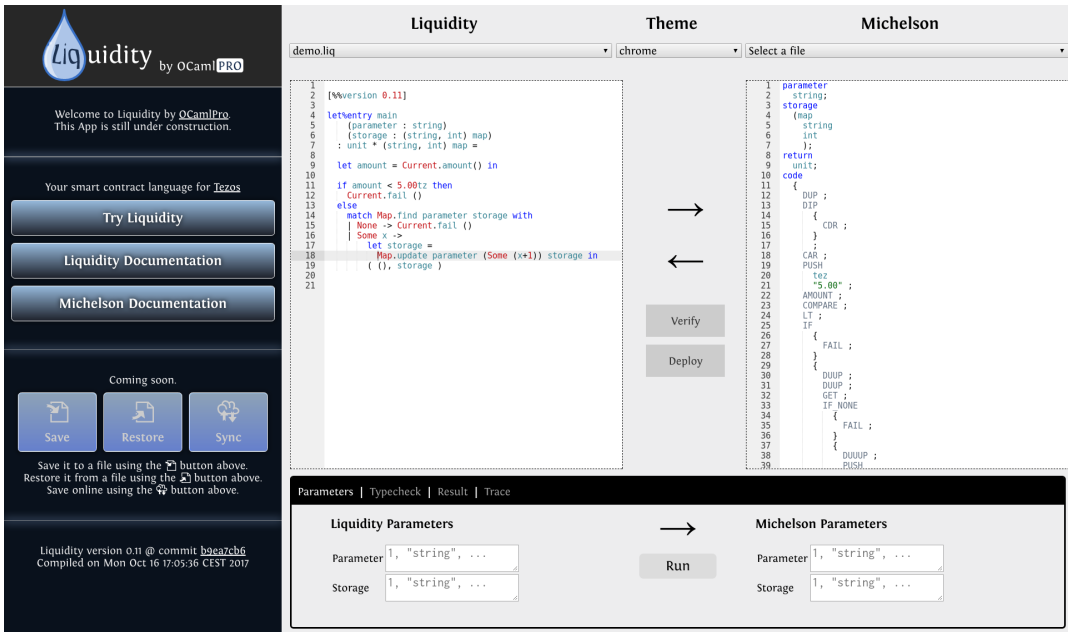


FIGURE 6 – Application Web d'édition de smart contract en Liquidity et Michelson.

personnaliser et colorer le code. La syntaxe de Liquidity étant très proche de celle d'OCaml, nous avons fait le choix de garder le même type de coloration (mots clés, variables, etc.) et d'indentation de code.

Enfin, les paramètres en bas de l'écran permettent d'exécuter le contrat. Pour cela, il est possible d'entrer directement les valeurs en Michelson, ou bien de les écrire en Liquidity et de les compiler vers des valeurs Michelson.

Le bouton *Deploy* permettra de déployer le contrat Michelson sur la blockchain Tezos.

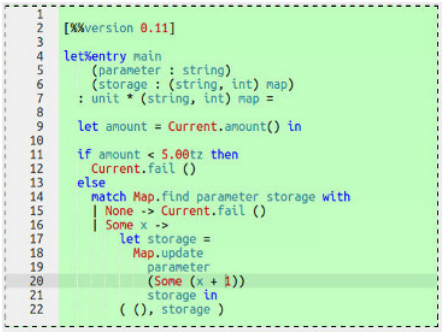
À la figure 7, nous présentons les différents effets visuels sur les éditeurs qui aident au développement. Lors d'une compilation ou décompilation, l'éditeur clignote en vert ou en rouge (figure 7a et 7b) pour signaler la réussite ou l'échec de ce processus. Lorsque la compilation réussit, le code dans l'autre éditeur est immédiatement modifié pour correspondre aux résultats de la compilation/décompilation. En cas d'échec, une erreur est signalée dans la marge de l'éditeur à la ligne correspondante (figures 7c et 7d).

7.2 Exécution et déploiement de contrat

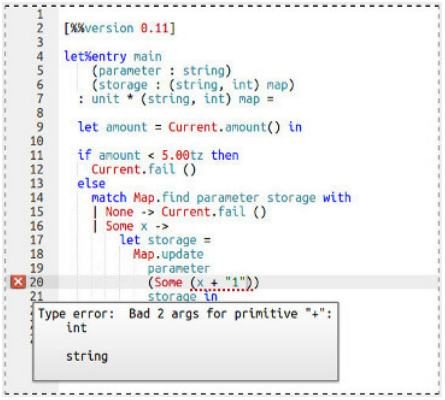
Comme pour tout programme, une fois le code compilé, il est important de pouvoir le tester. Il en est de même pour le code Michelson, généré à partir du code Liquidity. Pour cela, il est possible d'entrer des paramètres Michelson ou de les compiler directement depuis des paramètres Liquidity. Pour cela, nous avons mis en place un nœud Tezos et un serveur répondant aux requêtes de l'interface. Cela consiste entre autres à envoyer le code avec les paramètres au serveur qui fait à son tour une requête sur le nœud.

À la figure 8, on peut voir les différents champs décrits précédemment. Ce processus de compilation des paramètres utilise les informations de type du code Liquidity, pour vérifier que le paramètre et le storage ont le bon type. Dans le cas où ces valeurs n'ont pas le bon type, il

(a) Clignotement en rouge : erreur de compilation.



(b) Clignotement en vert : pas d'erreur.



(c) Affichage des erreurs de compilation Liquidity dans la marge de l'éditeur.

(d) Affichage des erreurs Michelson de compilation dans la marge de l'éditeur.

FIGURE 7 – Les différents effets visuels sur les éditeurs.

ne sera pas possible de les compiler vers les valeurs Michelson correspondantes.

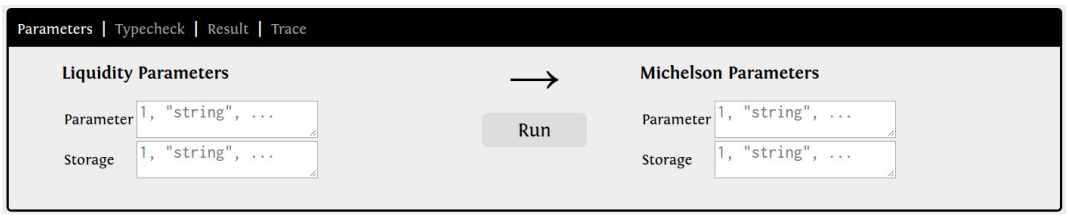


FIGURE 8 – Paramètres Liquidity et Michelson permettant l'exécution et le déploiement des contrats sur la blockchain.

Pour obtenir un résultat via un nœud Tezos, il faut donc compiler son code Liquidity en Michelson, ainsi que le paramètre et le storage. Une fois que cela est fait, on peut alors l'exécuter et obtenir un résultat. À la figure 9, on peut d'abord voir le résultat des types que nous renvoie le nœud. À la figure 10, nous pouvons voir le résultat après l'exécution. Cela se résume par la valeur du storage après exécution et la valeur de retour du contrat.

Comme indiqué précédemment, ce même procédé sera utilisé pour déployer le contrat sur la blockchain. La requête sera envoyée au serveur qui pourra alors faire la requête de déploiement avec en plus les informations sur l'identité cryptographique de l'utilisateur.

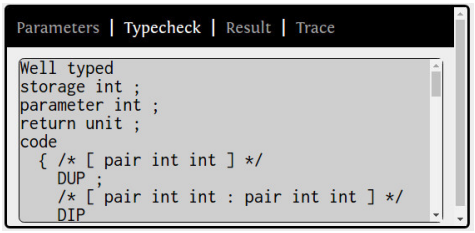


FIGURE 9 – Résultat du typage par le noeud Tezos.

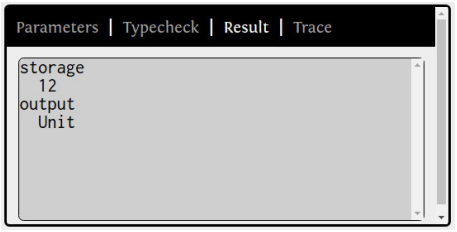


FIGURE 10 – Résultat de l'exécution d'un contrat.

8 Conclusion

Dans cet article, nous avons présenté Liquidity, un langage de smart contract plus lisible que Michelson, le langage de la blockchain Tezos. Liquidity utilise la syntaxe d'OCaml, mais avec un système de types différent, sans polymorphisme et avec certaines opérations surchargées (opérations arithmétiques). La distribution de Liquidity est diffusée en open-source sur Github, et contient un compilateur vers Michelson et un décompilateur depuis Michelson. Try-Liquidity est une version en ligne permettant d'éditer des contrats en Liquidity ou Michelson, de les compiler ou décompiler, puis les exécuter sur des exemples simples.

Plusieurs évolutions importantes de Michelson sont à venir, et Liquidity devra s'y adapter. La plus importante sera d'avoir plusieurs points d'entrée dans un contrat Michelson, comme c'est aujourd'hui le cas en Solidity (Ethereum). Une autre limitation qui pourra être levée est l'impossibilité d'effectuer des appels d'autres contrats depuis une fonction (lambda).

Un travail important pour la suite sera l'ajout d'un dévermineur (debugger), qui permettra d'exécuter un contrat Liquidity pas à pas pour mieux en comprendre le fonctionnement. L'automatisation du déploiement des smarts contracts en Liquidity est aussi un enjeu important. Une possibilité serait d'utiliser un système à la *eliom*, permettant de combiner dans un même source le code du smart contract, déployé sur la blockchain, et celui de l'application interagissant avec ce smart contract, en vérifiant la correction du typage des requêtes effectuées vers la blockchain.

Références

[1] fi - smart coding for smart contracts. https://tezrpc.me/fi/, 2017.
[2] Remix, solidity IDE. https://remix.ethereum.org, 2017.
[3] Vitalik Buterin. A next-generation smart contract and decentralized application platform. 2014.
[4] B. Canou, C. Bozman, and G. Henry. Sous le capot du MOOC OCaml. JFLA'2016.
[5] Fabien Fleutot. Lamtez : a typed lambda-calculus for tezos. https://github.com/fab13n/lamtez.
[6] L.M. Goodman. A self-amending crypto-ledger. tezos white paper. 2014.
[7] Satoshi Nakamoto. Bitcoin : A peer-to-peer electronic cash system. 2008.
[8] Jaanus Pöial. Algebraic specification of stack effects for forth programs. Janvier 1990.
[9] Jérôme Vouillon and Vincent Balat. From Bytecode to JavaScript : the Js_of_ocaml Compiler. 2013.

Towards Secure and Trusted-by-Design Smart Contracts

Zaynah Dargaye, Önder Gürcan, Florent Kirchner, and Sara Tucci Piergiovanni

CEA, LIST, Point Courrier 174, Gif-sur-Yvette, F-91191 France
lea-zaynah.dargaye@cea.fr, onder.gurcan@cea.fr, florent.kirchner@cea.fr,
sara.tucci@cea.fr

Abstract

Distributed immutable ledgers, or blockchains, allow the secure digitization of evidential transactions without relying on a trusted third-party. Evidential transactions involve the exchange of any form of physical evidence, such as money, birth certificate, visas, tickets, etc. Most of the time, evidential transactions occur in the context of complex procedures, called evidential protocols, among physical agents. The blockchain provides the mechanisms to transfer evidence, while smart contracts – programs executing within the blockchain in a decentralized and replicated fashion – allow encoding evidential protocols on top of a blockchain.

As a smart contract foregoes trusted third-parties and runs on several machines anonymously, it constitutes a highly critical program that has to be secure and trusted-by-design. While most of the current smart contract languages focus on easy programmability, they do not directly address the need of guaranteeing trust and accountability, which becomes a significant issue when evidential protocols are encoded as smart contracts.

1 Introduction

The rise of immutable distributed ledgers, or *blockchains*, extends the *"code is law"* vision to organizations and corporations through the Decentralized Autonomous Organization (*DAO*) concept. Indeed, blockchains allow the secure digitization of *evidential transactions* without relying on a trusted third-party [20] – where evidential transactions involve the exchange of any form of physical evidence, such as money, birth certificates, visas, tickets, etc. Most of the time, evidential transactions occur in the context of complex procedures encoded by specific programs, called *smart contracts*, executing within the blockchain in a decentralized and replicated fashion. A DAO, in particular, is an organization run by smart contracts encoding rules of governance. For those organizations, promises in terms of independence and savings are great advantages that smart contracts can enable, through the removal of both the trusted third-party and the need for repetitious (often manual) execution of contracts.

The smart contract concept thus plays a major role in a DAO. However, there is no official definition of smart contract: so far, no agreement has been established regarding this concept. Originally in 1994 in [20], Nick Szabo defined a smart contract as *a computerized transaction protocol that executes the terms of a contract*. The general objectives are to satisfy common contract conditions (such as payment terms, liens, confidentiality, and even enforcement) and minimize the need for trusted intermediaries. Related economic goals include lowering fraud loss, arbitration and enforcement costs, and other transaction costs. Nick Szabo believes that a contractual clause implemented in software and benefiting from cryptographical mechanisms would make the infringement of contracts very expensive – it is exactly this aspect that makes the contract "smart". Nowadays, a smart contract designates any piece of code running in the blockchain, losing the legal flavor but keeping its immutability properties: once in the blockchain the smart contract cannot be changed, becoming the "law" for the community using it.

Recently Josh Stark, Head of Operations & Legal at LedgerLabs, in the Ethereum community made an attempt to clarify these notions identifying two kinds of smart contracts (see [19]): *smart legal contracts* and *smart contract codes*. So far, most of the attention has focused on improving and designing smart contract code. Smart contract code languages, such as Solidity for Ethereum, lack both formal foundations and the expressiveness to program smart legal contracts in a secure way. To make things worse, a large number of vulnerabilities coming from the execution of the programs in a widely distributed network, i.e., the blockchain network, are fully exploitable today.

The lack of formalization and verification has already shown its impact: the famous attack on *The DAO* [8] was caused by a simple bug in the smart contract. The survey [3] shows that most attacks in Ethereum were caused by bugs or vulnerabilities of the execution platform (Ethereum Virtual Machine and blockchain network). Recently, the Tezos ledger has taken several steps to address these problems, with the use of OCaml for its implementation, and Michelson, a statically typed functional language for its smart contracts. However, the overall challenge of smart contract security remains unchanged. We advocate that to tackle this challenge there is the need (i) of a formal language specifically designed to capture the notion of trust (accountability, authentication, privacy) for the secure implementation of smart legal contracts in software, (ii) to capture and abstract the subtleties of the execution of smart contracts in a possible wide distributed environment and (iii) to provide full support for a compilation chain towards smart contract code language.

In this paper, we focus on the design of the smart legal contract language. We advocate that such a challenge could be addressed by extending so-called trust management frameworks to the blockchain. Trust management frameworks are formal frameworks allowing to formalize the specification of evidential protocols and to verify their correct implementation using a trusted-by-design approach. In particular, this paper presents a Coq implementation of a Cyberlogic framework that gathers the main features of smart legal contracts. Cyberlogic is a trust management framework, a first-order logic featuring an authority algebra for specifying protocols and their policies. Therefore, the Coq implementation enables to mechanically specify and verify Cyberlogic protocols, enabling to provide computable proofs. Moreover, as the Cyberlogic implementation is a shallow embedded one, a smart legal contract specified in Cyberlogic can be extracted to an executable language.

The paper is organised as follows: Section 2 presents the state of the art on the specification of evidential protocols detailing Cyberlogic and its implementation in Coq. Section 3 shows how to apply Cyberlogic to the design of trustworthy smart contracts. We corroborate our statement by specifying a smart contract in Cyberlogic and we present its implementation in a smart contract code. Finally, Section 4 presents our future work.

2 State-of-the-Art: Specifying Evidential Protocols

Evidential protocols by essence are based on trust between entities and particularly third-parties such as governments, banks or insurances. In an evidential protocol, those kind of entities exchange specific data, called evidences. A piece of evidence can be an official certificate, a license or a visa. It constitutes critical data, and is usually stated or claimed by a specific authority. Although accountability is endorsed by the authority which claims it, an evidence has to be formally specified as well as any other data in a protocol. Smart contracts have a lot in common with evidential protocols where a smart contract represents the interaction between the blockchain and the real world.

In this section, we detail the state-of-the-art of evidential protocols specification. First, we

present trust management frameworks which allow to specify trust and accountability. Second, we focus on the Cyberlogic framework which is an extended trust management framework allowing reasoning and verification. Finally, we detail our Coq implementation of Cyberlogic.

2.1 Trust Management

The underlying concepts of evidential protocols are the accountability of the evidence, its establishment and its usage. Therefore, the specification of evidential protocols requires being able to specify trust and accountability, which is the purpose of *trust management*.

PolicyMaker [6] is a trust management tool which associates to the management of the security (via public/private keys) a logic of predicates on keys. PolicyMaker is a language which features the ability to insert logic assertions. These assertions can express that a certain key permits (authorizes) a certain predicate. However, PolicyMaker does not allow formal reasoning for constructing a proof that validates the claim of an authority (ie., an abstraction of the entity that owns the key). The descendant of PolicyMaker is KeyNote [5], an other trust management framework is Fidelis [22]. This approach tackles the specification of policies in evidential transactions.

On the other hand, interesting contributions have been done about verification of trust in evidential protocols. In particular, a first alternative is to use the *a posteriori* validation of the authentication [2]. The proof accompanies the data and is verified at the reception by the authority. It has a cost and adds some work to the authority. [13] describes a delegation logic which considers the delegation of the authority. This logic permits expressing relation such as: "A claims P", "A delegates P to B with a delegation of k depth (a trusted chain of k authorities)", "A claims for B about P".

The Cyberlogic framework inherits the major features of trust management systems. In addition, Cyberlogic features native constructions to deal with distributed systems.

2.2 Cyberlogic

Cyberlogic [18] is designed for the transition from paper documents to electronic documents. In particular, Cyberlogic offers a framework for formalization and reasoning on attestation elements such as visas, certificates *etc*... Cyberlogic is a logic and a distributed program in which protocols can play several exchanges of attestations. As Cyberlogic features a first order logic, it allows to reason upon the actions of a Cyberlogic protocol. Cyberlogic also features an authorities' algebra allowing it to reason upon trust and accountability.

In addition to the expression of certificates and visas, we believe that the authorities' algebra can also be useful in complex systems verification, and whenever trust into an entity takes the form of a formal verification : acceptance.

This could, for instance, be the case of a development using a black-box library. The library editor claims that the library has been formally verified. Then, the user can suppose that each function of this library obeys its specification. In that configuration, Cyberlogic provides feature to naturally express some hypotheses usually categorized as informal hypotheses, or working hypotheses. If *Spec* is the specification of the function f of the library, then the user can use Cyberlogic to formalize its development. Hence, the specification of f becomes a formal formula: $E \ \triangleright \ Spec(f)$. If the editor has used Cyberlogic to formalize the library, then the library specification can be reused by the user. Finally, if f does not fill its specification and appears to bug in the development, the user can identify the editor as guilty and accountable.

An interpreter for Cyberlogic protocols and formulas has been developed at SRI International, using the PVS system [16]. Since 2013 at CEA List, thanks to two national projects

(SystemX MIC and FUI GeoTransMD) a Coq [14] implementation has been developed and used to verify authenticity properties of distributed protocols in communication systems.

In particular, the authors of [18] state that Cyberlogic is designed to enable evidential protocols implementation into agent-based frameworks where agents execute Cyberlogic protocols to carry out specific tasks that require the exchange of authorization and authentication information. Nowadays, this definition seems quite similar to blockchain protocols. In particular, we believe that Cyberlogic provides most of the properties required by a smart legal contract language.

2.3 Shallow Embedding Implementation of Cyberlogic in Coq

We developed a theorem prover for Cyberlogic as a shallow embedded implementation of Cyberlogic in the Coq theorem prover. The shallow embedding implementation allows to inherit all Gallina expressiveness and to use the standard libraries of Coq. Our implementation benefits the extraction mechanism as well. Moreover, it allows to focus only on the specific feature of Cyberlogic: the attestation. An attestation is a logic formula claimed by an authority. The other Cyberlogic constructors are those of the Coq language itself.

2.3.1 Authority and Attestations

Authorities are the owners of claims. A claim is a property of a data endorsed by an authority. The authority is accountable for the claimed property. An authority is the abstraction of a entity such as a person, a bank, a company or an organization, i.e. any actor of the system who can establish or claim a property. In Cyberlogic, an authority is uniquely identified and be can compare to other authorities via a decidable comparison.

2.3.2 Different Kinds of Attestations

An authority can claim a property either directly or indirectly. If an authority claims something directly then it could have established it itself and is accountable for this claim. If an authority claims something indirectly, it means that another authority transmits her(him) this claim. In other words, the chain of trust can be traced to find the accountable authority.

In the formalization we name the qualification direct or indirect through the access concept.

```
Inductive access := | Direct : access | Indirect : access.
```

There are different kinds of claims (named `authority` in the Coq development). In addition to the combination of an authority and an access mode, time can be taken in account in some claims. As in modal logic, it is possible to claim a fact before, at or after a date. Then, a claim defines an owner, an access and a timing for a claim.

```
Inductive authority :=
| Key : Authority → access → authority
| KatT : Authority → access → time → authority
| Kbef : Authority → access → time → authority
| Kaft : Authority → access → time → authority.
```

2.3.3 Attestations

An attestation is a property which is claimed by a authority.

```
Variable attestation : authority → Prop → Prop.
```

For the sake of clarity, let's define some notations as similar as possible from those of the original paper [4].

$k \mid > \ f$	$k \ \triangleright \ f$	k attests directly f
$k \ * \mid > \ f$	$k \ : \triangleright \ f$	k attests indirectly f
$k \mid >= t \ f$	$k \ \triangleright_{=t} \ f$	k attests directly f at t
$k \ * \mid >= t \ f$	$k \ : \triangleright_{=t} \ f$	k attests indirectly f at t
$k \mid >< t \ f$	$k \ \triangleright_{<t} \ f$	k attests directly f before t
$k \ * \mid >< t \ f$	$k \ : \triangleright_{<t} \ f$	k attests indirectly f before t
$k \mid >> \ t \ f$	$k \ \triangleright_{>t} \ f$	k attests directly f after t

2.3.4 Delegation of Attestation

Cyberlogic authorities' algebra features a delegation system which appears in the indirect mode. An indirect claim models that this claim has been obtained by delegation from another authority. The accountable is the originate claimer. $D1(A, B, P)$ says that A claims P and A knows P from B, B is accountable for P. $D2(A, B, P)$ says that A has been directly delegated by some other authority (C) to claim P. But C is not accountable for P, B is. $D2$ is a special delegation chain of depth 3. A delegation is indexed by the length of the chain of authority from the original claimer to the authority which actually claims.

```
Definition Dinf (k k': Authority) (A:Prop) := (k' *|> A)-> (k *|> A).
Definition D1 (k k': Authority) (A:Prop):= (k'|>A) → (k *|> A).
Definition D2 (k k': Authority) (A:Prop):=
 (∀ (k0 : Authority), (k0 |> A) ∧ (k' |>((k0 |> A)->A))) → (k *|> A).
```

2.3.5 Time Reasoning in the Cyberlogic

Cyberlogic integrates features of modal logic in the authority algebra. A specific authority is accountable of time: Kt. Kt establishes the current time by the predicate `curr`, date in the past by the predicate `has_been` and in future by the predicate `in_futur`.

```
Variable hasbeen: time → Prop.
Definition is_time (t:time):Prop:= Kt *|> (hasbeen t).
Definition curr (t:time):Prop:= is_time t ∧ (∀ (t':time), (is_time t') → t'≤ t ).
Definition in_future (t:time):= Kt *|> ~(hasbeen t).
```

3 Cyberlogic for Specifying Smart Legal Contracts

Cyberlogic allows reasoning both on (i) actions, thanks to a first-order logic and (ii) on trust and accountability, thanks to the authority algebra. Being specifically designed for evidential protocols and benefiting from a implementation in Coq, we advocate that Cyberlogic is, indeed, a perfect candidate to be at the core of a smart legal contract language. To support our claim, in this section, we show Cyberlogic at work by specifying the Schengen visa management process as a Cyberlogic protocol and present the corresponding smart contract code in Solidity[1].

[1]The availability of a formal and proved specification opens the way for a formal compilation chain targeting the smart contract code; the compilation chain, however, is not in the scope of this paper (see section 4 for a detailed future work discussion.

3.1 Schengen Visa Management

The Schengen visa management which is described in accordance with the European Commission, is an evidential protocol where the visa represents an evidence delivered by a country of the Schengen area. In order to obtain that evidence, it also requires to gather other evidence (in accordance to the requirements of the visa management process). The visa itself is an evidence stating that all the requirements are satisfied.

Nowadays, the execution of visa management protocol is time consuming, with latency experienced for each required evidence, and mostly manual (photocopies, forms, ...). The digitalisation of the entire protocol through smart legal contracts is indeed a good opportunity to improve the protocol and make it more secure. In the remainder of this section we provide a characterisation of the visa management as a Cyberlogic protocol, highlighting the kind of reasoning that can be made. In particular, the Cyberlogic specification allows the verifier of the visa (e.g. the customs officer) to roll-up the entire chain of trust to find the authority accountable for the possible error.

3.2 The Schengen Visa as a Cyberlogic Protocol

We propose to implement the Schengen visa management protocol as a smart legal contract. The result would be to alleviate centralization and possible related bottlenecks by digitizing the process in a secure and decentralized way [2]. Let us note, indeed, that the intermediary here is the smart legal contract itself that will be later encoded as a smart contract code running in a blockchain. The smart contract code will, indeed, (i) encode all the rules pertaining to the management of the Schengen's visa, including rules on authorities that have the right to deliver it, and (ii) will be executed in a secured and decentralised fashion among the blockchain participants.

Let us now to explicit our smart legal contract. First of all, to satisfy the autonomy and the authority of the real states of the Schengen area, the smart legal contract includes a consulate (or prefecture) role as official authority having the rights to deliver the visa. At delivery time, this official authority delegates the visa to the requester, where: (i) the official authority is still the unique accountable for the visa and (ii) the requester is allowed to provide directly the required pieces of evidence.

Let us note that the consulate can be viewed as a trusted third party we are still relying on. On the other hand, we propose to formally delegate the entire procedure to the requester and to exploit blockchain immutability properties to correctly compute proofs on executed scenarios. This approach without fully realising the vision of a decentralised and autonomous organisation, represents a first step in this direction [3]. Anyway, a smart legal contract to manage the Schengen visa will allow active controls, transparency and detection of contradictory requirements. A controller will be able to "go behind" the visa itself and consult the requirement it represents. In case of two requirements are contradictory, the controller will be able to observe it.

The management of the Schengen visa as a smart legal contract in Cyberlogic is composed of 4 functions which are: its demand, its delivery, its control and its indictment. Let's informally define these 4 functions.

To demand a Schengen visa is to write evidential requirements of the Visa's protocol in the

[2] We are aware that the adoption of such a smart legal contract is not only a technical issue, since legal compliance with the proposed framework should also be evaluated. However, since our goal is to show how CyberLogic works, we are not taking into consideration legal issues.

[3] Our approach is still compliant with the actual organisation of States and their rules, a different fully decentralized organisation would be possible using self-certification and new governance rules.

blockchain, once and for all thanks to immutability. The accountability is preserved thanks to the Cyberlogic protocol, where each evidential requirement is a claim from the appropriate authority.
To deliver a Schengen visa is to write the visa in the blockchain, the visa is claimed by the consulate (or official state organization) and is delegated to the requester, thanks to the Cyberlogic delegation. The chain of trust from the visa is digitized thanks to the delegation mechanism of Cyberlogic.
To control a Schengen visa is a read in the blockchain, accessing to the evidential requirements by rolling-up the trust chain.
To suspect a Schengen visa is to extract the evidential requirements from the blockchain, analysing the evidential requirements and identifying the potential suspicious claims, i.e. computing accountability.

3.3 The Cyberlogic Protocol

We present the smart legal contract of the Schengen visa as a Cyberlogic protocol that exchanges transactions that carrying a visa, its demand or its control.

```
Inductive transaction:=
| Demand: Schengen_demand → transaction
| Deliver : visa → transaction
| Control : visa → transaction.
```

Queries are the suspicions that an officer can make. When an officer suspects a visa, he will make queries about it.

```
Definition query := visa → Prop.
Definition queries := list query.
```

A smart contract answers to a querry either that the visa is valid or that there is a list of suspicious claims about the visa.

```
Inductive answer:=| Valid : answer| Suspects : list Prop → answer.
```

Action designates the interaction API with the blockchain. It consists in three operations on transactions: reading the ledger, writing in the ledger and questioning the ledger about properties on a specific data. The API presented here is light but sufficient for highlighting the Cyberlogic properties.

```
Variable action:Type.
Variable write: Authority → transaction → action.
Variable read : Authority → transaction → action.
Variable verify : Authority → visa → queries → answer.
```

The functions of the Schengen visa smart contract in the Cyberlogic protocol becomes:

```
Definition demand (Requester: Authority)(C:country) (f:schengen_form) (pic:photo)
(pport:passport)(trvls:travel_itinerary)(ins_policy:travel_health)(accs:accommodations)
(suff:sufficient_means)(t:time):=
    write Requester (Demand (mkDemand f pic pport trvls ins_policy accs suff t)).
Definition deliver (cons:Authority)(v:visa):= write cons (Deliver v).
Definition control (officer:Authority)(v:visa):= read officer (Control v).
Definition suspect (officer:Authority)(v:visa) (ev:queries):= verify officer v ev.
```

3.4 The Protocol Policies

The policy of demanding a visa is a time-stamp verification. The requester is accountable of the demand.

```
Definition demanding (r:Authority) (t:time) (c:country) (d:Schengen_demand):=
  time_stamp d = t ∧ (r |>=t make (demand r c (form d) (picture d) (pass d) (travels d)
 (insurance d) (lodgings d) (sufficient d) (time_stamp d))).
```

A consulate delivers a visa if the visa demand is valid. The appendix (see A) details the validity of a demand: the seven requirements are satisfied and claimed by the appropriate authority. For example, the validity of the passport of a citizen of a country C, is a claim of C, i.e. C is accountable for the validity of the passport.

```
Definition delivering_validation (cons req:Authority) (v:visa) (t:time) :=
 ∃ d, visa_of_demand d v ∧ demanding req (time_stamp d) (country_of cons) d ∧
   schengen_demand_validation req d ∧ (cons |>=t (make (deliver cons v))).
Definition delivering (cons req:Authority) (v:visa) (t:time) :=
 (D1 req cons (cons |><t delivering_validation cons req v t) ).
```

Of course only specific persons can control a visa, this officer has to be one of the Schengen area. If an officer has some suspicions regarding specific properties of the visa, two cases can happen. First case, this was a false alert and the officer claims a proof that all queries are satisfied. Second case, this was a real alert and the officer claims a list of suspicious claims that have to be checked.

```
Definition controlling (officer:Authority) (v:visa) (t:time):=
   schengen_officer officer ∧ curr t ∧ (officer |>=t (make (control officer v))).
Definition false_alert (officer:Authority) (v:visa)(ev:queries) (t:time):=
 schengen_officer officer ∧ (officer |>=t (make_answer (suspect officer v ev))) ∧
 suspect officer v ev= Valid.
Definition raise_alert (officer:Authority)(v:visa)(ev:queries) (evidences: list Prop) (t:time):=
 schengen_officer officer ∧ (officer |>=t (make_answer (suspect officer v ev))) ∧
 suspect officer v ev= Suspects evidences.
```

3.5 The Cyberlogic Protocols at Work

In this section, we present a scenario specified in the Cyberlogic implementation to highlight the accountability computation, its role in conflict detection and the fact that it allows to raise an alert. Today, this facility can only be applied at design time while playing scenarios to improve Cyberlogic protocol. We plan to transpose this facility at runtime via a monitoring smart contract or service. To do this, the Cyberlogic framework has to be extended to consider transactions as first class citizens and to express accountability on transaction instead than only on properties. The scenario is as the following:

Jon Snow requests a Schengen visa to the French consulate for 3 months. He plans to stay in Paris from the 1st June 2018 to 31st August 2018 in the Icy Wall.

```
Definition JSaccs := mkAcc IcyWall FirstJune2018 ThirtyFirstAugust2018.
Definition JSaccs := JSacc::nil.
```

His flight tickets are:

Drogo airline	from Winterfell (Essos) 1st June 2018	to Paris (France) departure: 3am	on the flight 3 arrival: 3:30am
Drogo airline	from Paris (France) 31st August 2018	to Winterfell (Essos) departure: 4pm	on flight 10 arrival 4:30pm

```
Definition JSoutward :=
 (mkFlight Drogo 3 JonSnow (Winterfell, FirstJune2018+Wdep_t)
           (France, FirstJune2018+Farr_t) W_IATA F_IATA 100).
Definition JSreturn :=
 (mkFlight Drogo 10 JonSnow (France, ThirtyFirstAugust2018+Fdep_t)
           (Winterfell, ThirtyFirstAugust2018+Warr_t) F_IATA W_IATA 100).
Definition JStravels := JSoutward::JSreturn::nil.
```

Thanks to his new job as King of The North, he provides his employment contract as a mean of sufficient.

```
Definition was_KingOfTheNorth:= KingOfTheNorth JonSnow demand_t.
Definition JSsuff:= Employment Cwinterfell was_KingOfTheNorth.
```

He also provides its passport number (delivered by Winterfell) and a reference of its travel health insurance delivered by Three-eyed crow &cie.). We have axiomatized the no significant part of the requirements.

```
Definition JSdemand:= (mkDemand JSform JSpic JSpassport JStravels JSinsurance JSaccs JSsuff demand_t).
```

He obtains his visa, `JSvisa`. The 1st July 2018, a police officer controls his visa. The officer recognizes him and knows, thanks to the Winterfell Times that Lady Sansa Stark. It reveales that she took his job since 4 months. So he suspects his visas and asks for evidence of sufficient means.

```
Definition suspicious_clue := WinterfellTime |> (KingOfTheNorth SansaStark (demand_t - 5)).
Definition JSsuff_query (v:visa):=
 ∀ d, visa_of_demand d v → (sufficient d) = JSsuff → KingOfTheNorth JonSnow demand_t.
```

The protocol returns the claim of the employment contract, which is a claim of the Winterfell kingdom. At this stage, the smart contract waits for physical world intervention and raises an alert.

```
Axiom suspicious_sufficient_means:
demanding JonSnow demand_t France JSdemand → delivering CFrance JonSnow JSvisa deliver_t
→
raise_alert JaimeL JSvisa (JSsuff_query::nil) ((Cwinterfell |> was_KingOfTheNorth)::nil) suspicious_t.
```

While the expressiveness of Cyberlogic allows to specify and reason on smart legal contracts and execution scenarios, a lot has to be done to provide mechanisms to monitor accountability at runtime in a blockchain-based execution environment [4]. Moreover, the authorities algebra allows to claim properties while in a smart legal contract language, this is the object/data of a transaction that has to be claimed. The chosen example is an evidential protocol for which Cyberlogic is particularly suitable. It also highlights the trust and accountability issues that DAO has to manage. However, to cover the whole kind of DAO the Cyberlogic has to be extended to handle contractual relationship (see section 4). The scenario we had unfolded highlight the detection of conflict and active control. To treat more complex scenarios, with several demands, the write, read and query operations have semantics concording with the distributed and immutable characteristics of the ledger. Therefore, those operations have to be first-class citizen of the smart legal contract language. In the next section the blockchain-based smart contract, i.e. the code counterpart of the Cyberlogic protocol is presented.

[4] In section 4 this issue is discussed as work-in-progress.

3.6 A solidity smart contract code

We implemented the Schengen Visa smart contract given in Section 3.2 in the Solidity language (Algorithm 1). Solidity is a contract-oriented, high-level language whose syntax is similar to that of JavaScript and it is designed to target the Ethereum blockchain framework [9]. It is statically-typed, supports inheritance, libraries and user-defined abstract data types. Solidity contracts bundle data with the functions operating on that data and have mechanisms for restricting direct access to some of their components.

The Schengen Visa smart contract functions are implemented as demand(), deliver(), control() and suspect() functions respectively in Algorithm 1. It is assumed that the identity (address) of the consulate and the officer are already known by the contract, and they are used by the contract to restrict access to its functions. External calls to the contract functions bring a msg.sender parameter that returns the address of the caller. The contract can then use these information to verify the identities when needed using the modifier construct of Solidity together with the msg.sender parameter. Modifiers are used to amend the semantics of functions in a declarative way. The modifier onlyConsulate() is used for verifying that msg.sender is a valid consulate (Algorithm 1 line 6) and the modifier onlyOfficer() used for verifying that msg.sender is an officer of schengen aera (Algorithm 1 line 7). This way, it is assured that the deliver() function can only be called by the consulate (Algorithm 1 line 18) and the control() and suspect() function can only be called by the officer (Algorithm 1 line 30 and 34).

4 On-going and Future Research Perspectives

Our aim is to provide a framework to design and implement smart contracts in a secure and trusted manner. Our point of view is that a smart legal contract has to be compiled in a smart contract code. We advocate that an extension of Cyberlogic is a good candidate to specify the smart legal contract. As transparency and trust are at core of our vision, we plan to equip our framework with formal analyzers as well as a formal verification of the compiler. In this section, we discuss the different current works in progress and futures research perspectives that we intend to explore.

4.1 Specification of Legal Artefacts

Smart contracts aim at digitized parts of legal contracts between entities. Specifying law is a recurrent holy Grail in formal methods [17]. However, deontic logic [15] with some restriction to avoid paradoxes have been embedded in formal frameworks. In particular, the Contract Language [10] is an action-based language featuring concepts for permission, obligation and prohibition as first-class citizen. The tool CLAN allows to analyze specification in CL and detect conflicts. On the other hand, CL does not allow to reason about trust and accountability as Cyberlogic does. We aim at extending Cyberlogic with CL-like mechanisms to provide more specific expressiveness dedicated to smart legal contracts. The open project Legalese also aims at providing such a language based on *Contract Language*.

4.2 Targeting Secure Smart Contract Code

Cyberlogic protocols in our Coq implementation are Gallinea programs. Therefore, they can be extracted to Ocaml code and then be compiled into a smart contract code language. In particular, we target Ocaml based languages as they might provide a formal base such as Michelson for the smart contract language of Tezos [12]. Another possibility is to target a

Algorithm 1 The solidity implementation of the Schengen Visa smart contract. It is assumed that the identity (address) of the consulate and the officer are already known by the contract. Due to the space limitation the abstract data types Demand and Visa are not shown.

```
1: contract SchengenVisa {
2:
3: address public consulate;
4: address public officer;
5:
6: modifier onlyConsulate() { require(msg.sender == consulate); _; }
7: modifier onlyOfficer() { require(msg.sender == officer); _; }
8:
9: mapping (address => Demand) demands;
10: mapping (uint => Visa) visas;
11: ...
12:
13: function demand(uint sch_form_id, uint photo_id, string passport_id, uint[] travel_ids,
    uint travel_health, uint[] accommodation_ids, uint sufficient_means, uint time_stamp)
    public {
14:     address visaDemander = msg.sender;
15:     demands[visaDemander] = Demand(sch_form_id, photo_id, passport_id, travel_ids,
16:         travel_health, accommodation_ids, sufficient_means, time_stamp); }
17:
18: function deliver(Demand demand, string country, string duration) public returns (uint
    visaId) onlyConsulate {
19:     if (isValid(demand)) {
20:         visas[++visaId] = Visa(visaId, consulate, country, duration);
21:         return visa.id;
22:     } else return -1; }
23:
24: function isValid(Demand demand) returns (bool valid) private {
25:     valid = validateTravels(demand.travel_ids, demand.accommodation_ids);
26:     valid &= (demand.sufficient_means >= 5000);
27:     return valid;
28: }
29:
30: function control(uint visaId) public returns (Visa visa) onlyOfficer {
31:     var visa = visas[visaId];
32:     return visa; }
33:
34: function suspect(uint visaId, string reqField) public returns (var field) onlyOfficer {
35:     var visa = visas[visaId];
36:     var field = visa[reqField];
37:     return field; }
38:
39: } // contract SchengenVisa
```

subset of Solidity [9] of the Ethereum Virtual Machine EVM [21]. The execution infrastructure should also include a monitoring mechanism for trace/scenarios analysis which is linked to the Cyberlogic formalisation and allowing reasoning.

4.3 Specification of the Blockchain at Low-Level

The inherent non-determinism caused by the execution of the programs in a widely distributed environment subject to Byzantine failures and network partitions exposes the programs to several vulnerabilities, which are often misunderstood by programmers. The survey [3] shows, indeed, that most attacks in Ethereum were caused by vulnerabilities of the execution platform (Ethereum Virtual Machine and blockchain protocols).

Available formal analyses of security of the blockchain [11] have several limitations, assuming a perfect message diffusion mechanism, instantaneous communication and a fixed number of participants. These assumptions are far from being realistic so security thresholds as the famous "majority assumption" for Bitcoin (the system is secure if the majority the hashing power is in the hands of honest nodes) falls short in more realistic settings. A formal definition of blockchain (first attempts in [1, 7]) is indeed needed to (i) allow the secure design of blockchain protocols and (ii) to gain trust in the smart contract execution by defining formal semantics to specify the properties of the execution context.

5 Conclusion

We advocate that since the final goal of smart contracts is to obviate the use of trusted third-parties, a smart contract specification must allow reasoning about trust and accountability. By considering smart contracts as evidential protocols we take advantage of Cyberlogic to specify and verify them. In this paper we have shown the use of Cyberlogic through an illustrative example and we have described the overall approach along with the elements needed to provide a trustworthy framework to design and implement secure and trusted-by-design smart contracts. Finally, we are currently exploring extensions of Cyberlogic with deontic supports and dedicated features to interact with the underling immutable distributed ledger.

References

[1] E. Anceaume, R. Ludinard, M. Potop-Butucaru, and F. Tronel. Bitcoin a distributed shared register. In *Stabilization, Safety, and Security of Distributed Systems - 19th International Symposium, SSS 2017, Boston, MA, USA, November 5-8, 2017, Proceedings*, pages 456–468, 2017.

[2] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, CCS '99, pages 52–62, New York, NY, USA, 1999. ACM.

[3] N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on ethereum smart contracts sok. In *Conference on Principles of Security and Trust - Volume 10204*, New York, USA, 2017.

[4] V. Bernat, H. Ruess, and N. Shankar. First-order cyberlogic. Technical report, SRI International, 2005.

[5] M. Blaze, J. Feigenbaum, and A. D. Keromytis. Keynote: Trust management for public-key infrastructures (position paper). In *Security Protocols, 6th International Workshop, Cambridge, UK, April 15-17, 1998, Proceedings*, pages 59–63, 1998.

[6] M. Blaze, J. Feigenbaum, and M. Strauss. Compliance checking in the policymaker trust management system. pages 254–274. Springer, 1998.

[7] T. Crain, V. Gramoli, M. Larrea, and M. Raynal. (leader/randomization/signature)-free byzantine consensus for consortium blockchains. *CoRR*, abs/1702.03068, 2017.

[8] P. Daian. Analysis of the dao exploit. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/.

[9] C. Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, CA, USA, 1st edition, 2017.

[10] S. Fenech, G. J. Pace, and G. Schneider. *CLAN: A Tool for Contract Analysis and Conflict Discovery*, pages 90–96. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[11] J. A. Garay, A. Kiayias, and N. Leonardos. The bitcoin backbone protocol: Analysis and applications. In *EUROCRYPT (2)*, pages 281–310, 2015.

[12] L. Goodman. A self-amending crypto-ledger. tezos white paper. 2014.

[13] N. Li, B. N. Grosof, and J. Feigenbaum. A practically implementable and tractable Delegation Logic. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 27–42. IEEE Computer Society Press, May 2000.

[14] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Ver. 8.0.

[15] P. McNamara. Deontic logic. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2014 edition, 2014.

[16] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In *Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 748–752, London, UK, UK, 1992. Springer-Verlag.

[17] H. Prakken and G. Sartor. *The Role of Logic in Computational Models of Legal Argument: A Critical Survey*, pages 342–381. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[18] Rueß and N. Shankar. Introducing cyberlogic. In B. Martin, editor, *HCSS'03—High Confidence Software and Systems Conference*, Baltimore, MD, 1-3 April 2003.

[19] J. Stark. Making sense of blockchain smart contracts. https://www.coindesk.com/making-sense-smart-contracts/.

[20] D. Tapscott and A. Tapscott. *The blockchain revolution:how the technology behind bitcoin is changing Money,Business and the World*, pages 72,88,101,127. TNew York, New York : Portfolio / Penguin, 2016. ISBN-13: 978-1101980132.

[21] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. http://bitcoinaffiliatelist.com/wp-content/uploads/ethereum.pdf, 2014. Accessed: 2016-08-22.

[22] W. Yao. Trust management for widely distributed systems. Technical Report UCAM-CL-TR-608, University of Cambridge, Computer Laboratory, Nov. 2004.

A Specification of the Verification of Schengen Visa Requirements

A smart contract for Schengen visa requests a generic specification of visa. A visa is delivered by a specific authority, has a duration or an expiry date, is an evidence that allow someone to circulate in a country.

Variable *visa*: Type.
Variable *visa_delivered_by* : *visa* → *cyber.Authority*.
Variable *visa_duration*: *visa* → cyber.time.
Variable *visa_kind* : *visa* → Prop.
Variable *visa_country*: *visa* → country.

1-The Visa Application form must be fully completed and signed in the corresponding blanks.

```
Variable schengen_form:Type.
Variable schengen_from:schengen_form → time.
Variable schengen_to: schengen_form → time.
Variable schengen_requester:schengen_form → Authority.
Variable schengen_country : schengen_form → country.
Variable schengen_form_requirement: schengen_form → Prop.
```

2- A passport is delivered by an authority, has a expiry date and contains a list of visas.

```
Variable photo: Type.
Variable passport_photo: photo → Prop.
Variable passport:Type.
Variable passport_delivered_by: passport → Authority.
Variable visas_of_passport : list visa.
Variable passport_expericy_date: passport → time.
```

Passport_of is a mapping that associates a passport to its owner.

```
Variable passport_of: Authority → passport.
```

3- The passport as well as all the copies of your previous visas, valid for at least 3 months prior to your departure is required. The passport must have at least two blank pages.

```
Definition valid_passport_at (p:passport) (departure_time: time):=
 (passport_expericy_date p) ≤ (departure_time - (months 3)).

Variable valid_passport: passport → time → Prop.
Definition valid_at_time (p:passport)(departure_time: time):=
 (valid_passport p departure_time) → (valid_passport_at p departure_time).
```

4- Round trip reservation or itinerary with dates and flight numbers specifying entry and exit from the Schengen area. You can use the visa consultation services like this one. These guys can handle most of your visa requirements such as flight itineraries, hotel reservations along with free consultation over email.

```
Record flight := mkFlight
{ fl_airline: Authority; fl_id: nat; fl_for: Authority; fl_departure: country × time;
  fl_arrival: country × time; fl_dep_aipport: IATA; fl_arr_airport:IATA; fl_price: nat; }.

Definition trav_itinerary := list flight.
```

trav_valid holds is a flight that is valid recording the requirement of the Schengen visa. That would be a property claimed by an authority.

```
Variable trav_valid: flight → Prop.
Definition travs_valid (l:trav_itinerary) := ∀ fl, List.In fl l → ((fl_airline fl) |> trav_valid fl).
```

travs_consistency is a verified property that is not a claim.

```
Fixpoint travs_consistency (tvls : trav_itinerary) (tfrom tto:time):Prop:=match tvls with
| a::m ⇒ (snd (fl_departure a)) < (snd (fl_arrival a)) ∧
         (match m with
          | nil ⇒ (snd (fl_departure a)) = tfrom ∧ (snd (fl_arrival a)) = tto
          | m ⇒ (snd (fl_departure a))=tfrom ∧ (travs_consistency m (snd(fl_arrival a)) tto)
          end)
| nil ⇒ True end.
```

5- The travel health insurance policy is to be secured, covering any medical emergency with hospital care and travel back to ones native country due to medical motives. This health insurance policy has to cover expenses up to 30,000 euros, the sum depending on the residing days, and also it has to be valid in all Schengen countries.

The health insurance policy must be purchased before picking up the visa and if your visa is refused you can cancel it!

```
Variable trav_health :Type.
Variable trav_health_of: trav_health → Authority.
Variable trav_health_emitter: trav_health → Authority.
Variable trav_health_valid : trav_health → Prop.
```

6-Proof of accommodation for the whole duration of the intended stay in the Schengen area.

```
Record accd:= mkAcc{ shelter_at: Authority; from : time; to:time;}.
Definition accds:= list accd.
Variable accd_valid: accd → Prop.
Definition accds_valid (acc:accds):= ∀ ac, List.In ac acc → ((shelter_at ac) |> accd_valid ac).

Fixpoint accds_consistency (accs:accds) (tfrom tto:time):Prop:=match accs with
| a::m ⇒ (from a) < (to a) ∧
         (match m with
          | nil ⇒ (from a) = tfrom ∧ (to a) = tto
          | m ⇒ (from a)=tfrom ∧ (accds_consistency m (to a) tto)
          end)
| nil ⇒ True end.
```

7- Proof of sufficient means of subsistence during the intended stay in the Schengen area. Varies from country to country. To complete Supporting document to attest sponsor's readiness to cover your expenses during your stay Proof of prepaid accommodation Document about accommodation in private Proof of prepaid transport.

```
Inductive sufficient_means:=
| Bank_statement : (Authority → Prop) → sufficient_means
| Credit_card : (Authority → nat → Prop)-> sufficient_means
| Cash : (nat → Prop) → sufficient_means
| Employment : (Authority → Prop) → sufficient_means.

Variable means_of_sufficiency : country → sufficient_means → Prop.

Record Schengen_demand := mkDemand
{ form: schengen_form; picture: photo;pass: passport; travs: trav_itinerary; insurance: trav_health;
  lodgings: accds;sufficient: sufficient_means; time_stamp:time; }.

Definition schengen_demand_valid (requester:Authority)(d_form:Schengen_demand):=
 let demand := form (d_form) in let C := schengen_country demand in
 let Consul:= consulat_of C in let tfrom := schengen_from demand in
  let tto:= schengen_to demand in
 (Consul |> (schengen_form_requirement demand)) ∧
 (requester |> passport_photo (picture d_form)) ∧
 (travs_valid (travs d_form) ∧ travs_consistency (travs d_form) tfrom tto)/\
 ((trav_health_emitter (insurance (d_form))) |> ( trav_health_valid (insurance (d_form)))) ∧
 ((accds_valid (lodgings d_form))/\ (accds_consistency (lodgings d_form) tfrom tto)) ∧
 ((means_of_sufficiency C) (sufficient d_form)).
```

Un cadre pour la preuve de programmes probabilistes

Florian Faissole[1] et Bas Spitters[2]

[1] Inria, Université Paris-Saclay, F-91120 Palaiseau
LRI, CNRS & Université Paris-Sud, F-91405 Orsay
florian.faissole@inria.fr
[2] Aarhus University
spitters@cs.au.dk

Résumé

L'usage d'assistants de preuves dans la certification de programmes probabilistes est de plus en plus répandu. La bibliothèque Coq ALEA formalise une théorie des probabilités discrètes via une approche monadique (variante de la monade de Giry). Cette formalisation est basée sur une axiomatisation de l'intervalle [0,1] et l'usage de sétoïdes. Dans cet article, nous proposons une alternative constructive à cette bibliothèque. Nous utilisons la bibliothèque HoTT pour Coq, une nouvelle interprétation de la théorie des types dans la théorie de l'homotopie et nous nous basons sur les principes de la topologie synthétique. La topologie synthétique est un formalisme mathématique dans lequel les ouverts sont au premier rang et dans lequel les notions topologiques ont une définition concise. Nous formalisons des notions mathématiques (nombres réels, ouverts, valuations, intégrales, etc) dans le cadre de ce formalisme puis les appliquons à l'interprétation de programmes fonctionnels probabilistes. Nous discutons par ailleurs les avantages de HoTT pour une telle formalisation.

1 Introduction

Les algorithmes probabilistes sont de plus en plus répandus. Ils permettent en effet de donner des approximations raisonnables pour certains problèmes et d'introduire de l'aléatoire dans les processus. La modélisation de programmes probabilistes est connue pour être un problème difficile, reposant sur des fondements mathématiques conséquents. On peut citer les travaux de Kozen [18], qui propose un langage de programmation probabiliste impératif. Plus récemment, des approches fonctionnelles sont apparues, reposant généralement sur des constructions monadiques [11, 17, 25].

Il est devenu courant d'utiliser des assistants de preuves pour fournir une connexion formelle entre un langage de programmation et sa sémantique. Des travaux récents se sont concentrés sur la programmation probabiliste. En Isabelle/HOL, Holzl utilise une construction monadique pour la formalisation de procédés de Markov [14]. Hasan et Tahar formalisent une théorie des probabilités continues dans l'assistant de preuves HOL [13] : leur formalisation est plus "analytique" au sens où elle utilise les notions de variables aléatoires et de fonctions de distribution. Le travail que nous présentons se rapproche davantage d'ALEA, une bibliothèque Coq pour la preuve de programmes probabilistes [2]. Il s'agit d'une formalisation de la théorie discrète de la mesure qui utilise une variante de la monade de Giry [11] comme sous-monade de la monade CPS $(A \to [0,1]) \to [0,1]$. Audebaud et Paulin interprètent le langage *Rml*, un mini-langage fonctionnel probabiliste. Cependant, ALEA repose sur la bibliothèque réelle standard de Coq, qui fournit une axiomatisation des nombres réels [22]. De plus, ALEA repose sur une axiomatisation de l'intervalle $[0,1]$ (qui est plus riche que l'axiomatison des réels) et dans lequel sont valuées les mesures. En outre, de par l'absence de quotients et d'extensionnalité fonctionnelle par défaut dans Coq, la bibliothèque utilise des sétoïdes qui peuvent rendre certains raisonnements fastidieux.

Nous proposons une interprétation constructive de programmes probabilistes. Notre approche repose sur la combinaison de la théorie homotopique des types [30] (voir Section 2.1) et de la topologie synthétique [8] (voir Section 2.2). La théorie homotopique des types nous permet d'utiliser les types quotients et l'extensionnalité fonctionnelle, ce qui simplifie les raisonnements. La topologie synthétique nous permet de fonder une variante constructive de la théorie de la mesure de manière concise. D'ailleurs, une part importante de notre formalisation concerne les concepts fondamentaux de la topologie synthétique. En aval, nous développons une théorie des valuations et des intégrales inférieures, qui sert de base à l'interprétation de programmes probabilistes. Nous nous basons sur la bibliothèque HoTT [4] et sur une branche expérimentale de Coq, qui implémente le mécanisme d'induction-récursion. Nous utilisons également la bibliothèque HoTTClasses, une adaptation de MathClasses pour HoTT [10].

La Section 2 décrit les formalismes mathématiques sur lesquels nous nous basons, à savoir la théorie homotopique des types et la topologie synthétique. La Section 3 est consacrée à la formalisation de concepts de base de la topologie synthétique dans HoTT, comme les nombres réels inférieurs. Dans la Section 4, nous formalisons une théorie des valuations et des intégrales dans le cadre de la topologie synthétique (semblables à la théorie de Lebesgue) puis appliquons notre développement à l'interprétation de programmes probabilistes dans la Section 5. Enfin, la Section **??** conclut et fournit des perspectives pour ce travail.

2 Fondements mathématiques

Nous proposons une approche conjuguant les formalismes de topologie synthétique et de théorie homotopique des types. Dans cette section, nous présentons brièvement les bases de ces deux formalismes mathématiques, et donnons les bénéfices de leur utilisation pour l'interprétation de programmes probabilistes.

2.1 Théorie homotopique des types

La théorie homotopique des types connecte la théorie des types et la théorie de l'homotopie en topologie algébrique [30]. Ce formalisme a pour but de fonder les mathématiques sur de nouvelles bases mêlant théorie des types et axiome d'univalence. Cet axiome relâche la notion d'égalité, qui, en théorie des ensembles, est une notion très stricte. En effet, l'axiome d'univalence postule qu'il y a une équivalence entre la notion d'équivalence et la notion d'égalité en théorie des types. Cette connexion permet par ailleurs de mieux relier mathématique et informatique, plaçant ainsi les assistants de preuves au coeur de la fondation des mathématiques. De surcroît, la théorie homotopique des types n'autorise pas de fonder des concepts de façon axiomatique et les mathématiques qui y sont développées sont constructives.

Coq repose sur une logique intuitionniste dans laquelle les types quotients et l'extensionnalité fonctionnelle ne sont pas disponibles par défaut. La bibliothèque ALEA requiert de telles constructions et utilise la machinerie des sétoïdes pour obtenir des constructions similaires [2]. Les sétoïdes rendent la bibliothèque plus difficile à utiliser, et requièrent une bonne connaissance de leur fonctionnement, puisqu'il est nécessaire de prouver que chaque fonction préserve la relation qu'ils décrivent. La théorie homotopique des types permet d'ajouter les quotients et l'extensionnalité fonctionnelle de façon consistante. Bien que leur intégration dans Coq reste aujourd'hui une question difficile, la bibliothèque HoTT [4] axiomatise ces fonctionnalités.

Tous les types de Coq ne peuvent pas être vus comme des "ensembles". Par exemple, un univers est un type mais n'est pas un ensemble (au sens usuel de la théorie des ensembles, correspondant à `Type_0` dans Coq). Ils ne correspondent pas au même niveau dans la théorie

des types de Coq. Dans HoTT, la distinction est plus précise, et il y a différents niveaux d'homotopie. Ainsi, les types de niveau -1 sont les propositions (*hProp*), les types de niveau 0 sont les ensembles (*hSet*, parmi lesquels on peut citer les entiers, les booléens, ...) et les types de niveau $n > 1$ (univers) correspondent aux n-groupoïdes en théorie des catégories. En fait, l'axiome d'univalence de Voevodsky correspond à l'extensionnalité pour les univers, dont le niveau d'homotopie est strictement supérieur à 1. Dans ce travail, nous identifions les espaces topologiques aux ensembles (*hSet*) de la théorie homotopique des types. Rijke et Spitters [26] décrivent précisément les propriétés de ces ensembles en utilisant la théorie des ΠW-pretopos.

2.2 Topologie synthétique

En mathématiques, on distingue l'approche analytique et l'approche synthétique. Tandis que l'approche analytique décrit les objets mathématiques par les points qu'ils contiennent (par exemple, la géométrique analytique fait appel aux coordonnées de points et aux équations de droites), l'approche synthétique est purement logique. Les concepts mathématiques sont déterminés par les axiomes qu'ils vérifient, ce qui offre un meilleur niveau d'abstraction.

La topologie synthétique est fortement inspirée de la théorie des domaines synthétique [16], un outil utilisé en sémantique. Elle en reprend donc la philosophie et les techniques. Nous utilisons un langage spécifique aux domaines (DSL) pour la topologie. Il s'agit, d'un point de vue catégorique, d'interpréter un langage dans une catégorie proche de celles des espaces topologiques [8, 5].

En topologie synthétique, les ouverts sont des citoyens de première classe. On utilise un objet $\mathbb{S}$ pour classifier les ouverts : l'espace de Sierpiński. L'espace $\mathbb{S}$ est l'espace topologique $\{\top, \bot\}$ muni de la topologie $\{\emptyset, \{\top\}, \{\bot, \top\}\}$. Les applications de type $A \to \mathbb{S}$ ($A : hSet$ espace topologique) sont les fonctions caractéristiques de A. Chacune d'elles caractérise un sous-ensemble B de A et peut être notée χ_B. Ainsi, $\chi_B^{-1}(\top) = B$. Or, par définition de la continuité de fonctions, χ_B est continue si et seulement si $\chi_B^{-1}(\top)$ est un ouvert de A, *i.e.* si et seulement B est un ouvert de A. Les applications de type $A \to \mathbb{S}$ caractérisent donc les ouverts de A.

Cette correspondance est centrale car elle permet d'interpréter d'autres propriétés topologiques de manière originale et concise [20]. Par exemple, si on considère $\widetilde{\mathbb{S}}$ l'espace $\{\top, \bot\}$ muni de la topologie $\{\emptyset, \{\bot\}, \{\bot, \top\}\}$ (qui inverse les ouverts et les fermés par rapport à $\mathbb{S}$), les applications de type $A \to \widetilde{\mathbb{S}}$ caractérisent les fermés de A. En topologie synthétique, un espace topologique A est de Haussdorff si et seulement si l'égalité (fonction définie dans $A \times A$) est un fermé de A. Pour définir l'espace de Sierpiński en théorie des types, la philosophie suivie est celle de la calculabilité synthétique [3]. D'après Bauer, les valeurs de vérité semi-décidables sont caractérisées par l'ensemble $\Sigma = \{p \in \Omega | \exists d : \mathbb{N} \to bool, p = \exists n, d(n)\}$ (Ω est l'ensemble des valeurs de vérité, soit $\mathcal{P}(\top) : \{\top\}$ correspond à "vrai" et $\emptyset$ correspond à "faux"). Une valeur de vérité U sur un type $A : hSet$ est semi-décidable si χ_A est valué dans Σ, *i.e.* de type $A \to \Sigma$. Σ a donc le même comportement avec les valeurs de vérité que $\mathbb{S}$ avec les ouverts. On peut donc définir $\mathbb{S}$ comme l'ensemble des valeurs de vérité semi-décidables.

Pour avoir un modèle pour la topologie synthétique, certains points doivent être satisfaits. L'implémentation de l'espace de Sierpiński comme ensemble des valeurs de vérité semi-décidables doit notamment vérifier la propriété de dominance [20, 9], ce qui signifie qu'un sous-ensemble ouvert X d'un sous-ensemble ouvert U de $A : hSet$ est aussi un sous-ensemble ouvert de A. Nous vérifions le modèle dans la Section 3. Il n'y pas eu d'extension des modèles de la topologie synthétique à la théorie des types. Néanmoins, il y a eu des travaux pour étendre d'autres modèles de réalisabilité (voir Shulman pour plus de détails sur ces problématiques [28])

à la théorie homotopique des types, comme le modèle cubique de HoTT développé dans l'assistant de preuves NuPrl [1]. Enfin, bien que les modèles de réalisabilité de la topologie synthétique vérifient l'axiome du choix dénombrable, nous prenons le parti de ne pas l'ajouter. Il n'est en effet pas disponible dans HoTT et n'est pas utile pour les applications de cet article.

Gilbert propose une implémentation de l'espace de Sierpiński basée sur la monade de partialité d'Altenkirch [1, 10]. La théorie homotopique des types et l'axiome d'univalence permettent de faciliter les raisonnements algébriques et catégoriques [30]. Ils facilitent notamment la formalisation de structures algébriques, notamment des structures quotients. Par exemple, la monade de partialité $A_\perp$ sur un ensemble A est l'ω-cpo complétion de A (voir détails dans Section 3.2) et est définie comme un type inductif-inductif quotient (QIIT) [1].

$A_\perp$ est défini par trois constructeurs $\eta, \perp, \bigcup$ et une relation $\subseteq$ (dont les propriétés attendues sont décrites dans l'article d'Altenkirch [1]) :

$$\begin{array}{c} A_\perp : hSet \qquad \subseteq_{A_\perp} : A_\perp \to A_\perp \to hProp. \\ \eta : A \to A_\perp \qquad \perp : A_\perp \\ \bigcup : \prod_{f:\mathbb{N}\to A_\perp} (\prod_{n:\mathbb{N}} f(n) \subseteq_{A_\perp} f(n+1)) \to A_\perp \end{array}$$

La partialité a été définie pour la bibliothèque HoTT [10] par des types inductifs-inductifs quotients. Ce développement est basé sur une branche expérimentale de Coq dans laquelle l'induction-récursion est disponible. Les valeurs de vérité semi-décidables ont été définies comme l'ensemble $\mathbb{S} = \mathbf{1}_\perp$, avec $\mathbf{1}$ le type *unit*. Gilbert montre que $\mathbb{S}$ est un treillis distributif. Nous devrons en outre prouver que cet ensemble est une dominance (voir Section 3.1).

Gilbert [10] formalise également les nombres réels de Cauchy et de Dedekind compatibles avec HoTT en se basant sur une adaptation de MathClasses [19, 29]. Cette adaptation contient notamment les propriétés algébriques et les propriétés d'ordre des nombres réels. Les réels de Dedekind y sont définis comme paires de prédicats de type $\mathbb{Q} \to \mathbb{S}$ enrichis des propriétés standard des coupures. Nous réutilisons cette approche pour définir les réels inférieurs (Section 3.3), qui se comportent comme des "semi-coupures" de Dedekind et permettent de définir des applications semi-continues inférieurement.

3 Formalisation de la topologie synthétique

Cette section présente une formalisation des fondements de la topologie synthétique utiles au développement d'une théorie des valuations. Nous vérifions que notre formalisation satisfait bien les principes attendus de la topologie synthétique et construisons une théorie des réels (inférieurs) compatible avec celle-ci.

3.1 Espace de Sierpiński et dominance

La notion de dominance est centrale en topologie synthétique. Une dominance est un sous-ensemble de *hProp* vérifiant certaines propriétés. Dans notre développement, nous plongeons toute dominance S vers *hProp* en la munissant d'une fonction .

Définition 1. *Un type S équipé d'une fonction $\texttt{IsTop} : S \to hProp$ telle que $\texttt{IsTop}(s) := (s = \top)$ est une* ***dominance*** *ssi $\top \in S$ et $\forall u : hProp, \forall s : S, (\texttt{IsTop}(s) \Rightarrow \exists z \in S, \texttt{IsTop}(z) = u) \Rightarrow \exists m \in S, \texttt{IsTop}(m) = u \wedge \texttt{IsTop}(s)$.*

1. http://www.nuprl.org/wip/Mathematics/cubical!type!theory/

Il existe une reformulation de cette définition plus proche de la théorie des types [9]. Il s'agit de considérer un prédicat $d(x) := \exists z \in \mathbb{S}.x = z$. En topologie synthétique, les ouverts d'un type A sont définis comme les objets de type $A \to S$ où S est une dominance. Nous souhaitons utiliser l'espace de Sierpiński formalisé par Gilbert [10].

Lemme 1. $\mathbb{S}$ *est une dominance.*

Démonstration. Comme $\mathbb{S}$ est égal à $\mathbf{1}_\perp$, nous procédons par induction sur $\mathbb{S}$. Les cas $u = \top$ et $u = \perp$ sont triviaux. Lorsque u est le supremum d'une suite croissante u_n de $\mathbb{S}$, nous devons construire une suite de témoins m_n. A priori, définir une telle séquence requiert l'axiome du choix dénombrable. Cependant, chaque m_n est unique (déterminé de façon unique par u_n). Or, dans HoTT, la propriété du choix unique est démontrable (le choix unique est une version constructive de l'axiome *iota* de Hilbert). □

3.2 Partialité et ω-cpos

Nous formalisons une théorie des ω-cpos (ensembles partiellement ordonnés avec un plus petit élément et dont toutes les chaînes croissantes d'éléments ont une borne supérieure) et les connectons à la monade de partialité [1], sur laquelle est fondée la formalisation des espaces de Sierpiński que nous utilisons [10] (voir Section 2.2). Altenkirch, Danielsson et Krauss [1] définissent le type $A_\perp$ de manière à enrichir A d'une structure d'ω-cpo. Cela signifie que, pour tout ω-cpo C et toute fonction $f : A \to C$, il existe une unique fonction $f_\perp : A_\perp \to C$ telle que $f_\perp \circ \eta = f$. Nous prouvons formellement ce résultat.

En Coq, toutes les fonctions terminent, ce qui constitue une garantie très forte et facilite la définition de sémantiques. Cependant, l'utilisateur souhaite occasionnellement écrire des fonctions qui ne terminent pas. Se pose alors la question de les ajouter à la théorie des types de façon consistante. La monade de partialité [1] est une solution possible. Notre sémantique est semi-décidable, au sens où elle décrit un langage récursivement énumérable (voir Section 2.2). Nous considérons donc les moyens d'évaluer des valeurs de vérité semi-décidables. Nous utilisons une tactique qui essaie de prouver qu'une valeur de vérité semi-décidable est vraie. La tactique prend en entrée un entier naturel m. `S_compute m` s'applique à des buts de la forme `IsTop(s)` : si `s` est égal à $\top$ alors le but est trivialement prouvé ; si `s` $= \sup(\lambda n.(f(n))$, la tactique essaie de prouver $f(m)$. Si elle échoue, elle réécrit $\sup(\lambda n.(f(n)) = \sup(\lambda n.(f(n+m))$ et réitère le même procédé jusqu'à obtenir un but qui se prouve par réflexivité. Nous simulons ainsi des fonctions possiblement non-terminantes. La technique est courante, l'entier m pris en entrée étant généralement appelé "fuel". Le procédé est comparable à la réflection des types décidables vers les booléens, caractéristiques de ssreflect [12]. Ici la reflection se fait vers les valeurs de vérité semi-decidables. Une tactique similaire a été utilisée en Coq pour semi-décider les nombres réels par OConnor [24, 5.5]. Cette tactique n'utilise cependant pas le mécanisme de réflection vers $\mathbb{S}$.

3.3 Réels inférieurs

À partir du type $\mathbb{S}$, nous formalisons les réels inférieurs $\mathbb{R}_l$. Ce sont des semi-coupures inférieures ouvertes de $\mathbb{Q}$, c'est-à-dire des objets de type $\mathbb{Q} \to \mathbb{S}$. Un réel inférieur envoie tout nombre rationnel qui lui est strictement inférieur vers l'élément $\top$. Ces nombres permettent de modéliser les fonctions de type $A \to \mathbb{R}_l$, qui sont les fonctions semi-continues inférieurement au sens de la topologie synthétique. De façon analogue, nous pouvons définir les réels supérieurs.

Les coupures de Dedekind correspondent à une paire valide contenant un réel inférieur et un réel supérieur. Constructivement, les nombres réels inférieurs (ou supérieurs) ne sont pas des

nombres réels [2]. Il n'est par exemple pas possible de construire l'opposé d'un réel inférieur sans perdre les propriétés de la structure des réels inférieurs.

Définir les nombres réels (inférieurs ou supérieurs) en utilisant le type $\mathbb{S}$ présente l'avantage de faire hériter les réels du même niveau d'homotopie que $\mathbb{S}$ et présentent par conséquent les avantages des ensembles de type *hSet*.

Pour gagner en généralité, on construit une théorie générale de semi-coupures ouvertes sur un ensemble A quelconque (le mot-clé `merely` spécifie que la proposition est dans *hProp*) :

```
Variables (A : hSet) (elt : A → Sier).
Class IsRC : Type := { is_inhab : merely (exists q, elt q);
      is_rounded : forall q, (elt q) ↔ (merely (exists r, q < r ∧ elt r)) }.
...
Class RC := { elt :> A → Sier; elt_RC : IsRC elt }.
```

Ainsi, nous définissons $\mathbb{R}_l$ comme instance de `RC` avec `A` $= \mathbb{Q}$ et `Rlt` $= <_{\mathbb{Q}}$. De même, les réels supérieurs sont une instance de `RC` avec `A` $= \mathbb{Q}$ et `Rlt` $= >_{\mathbb{Q}}$.

Nous formalisons ces nombres de façon constructive, et les munissons d'une relation d'ordre partiel $\leq_{\mathbb{R}_l}$ et d'un ordre strict $<_{\mathbb{R}_l}$. Nous sommes en mesure, pour toute chaîne croissante de réels inférieurs, de construire un supremum $sup_{\mathbb{R}_l}$ avec les propriétés attendues. En outre, on peut se restreindre aux réels inférieurs positifs $\mathbb{R}_l^+$ (ensemble d'arrivée de nos valuations et intégrales) et obtenons une structure d'ω-cpo, héritée de l'espace de Sierpiński $\mathbb{S}$:

Lemme 2. *$\mathbb{R}_l^+$ est un ω-cpo avec $sup = sup_{\mathbb{R}_l}$ et $\bot = 0_{\mathbb{R}_l^+}$.*

Ainsi, on peut injecter $\mathbb{S}$ dans les réels inférieurs. Si $\bot$ et $\top$ sont simplement envoyés vers $0_{\mathbb{R}_l}$ et $1_{\mathbb{R}_l}$, $\mathbb{S}$ comporte un troisième constructeur *sup* qui construit un élément de $\mathbb{S}$ à partir d'une suite croissante d'éléments de $\mathbb{S}$. Pour injecter le *sup* d'une suite de $\mathbb{S}$ dans les réels, il faut alors injecter les éléments de la suite dans $\mathbb{R}_l$, ce qui donne une suite de réels, puis en prendre le supremum grâce à l'opération $sup_{\mathbb{R}_l}$.

Nous munissons les réels inférieurs de plusieurs opérations élémentaires. Parmi les plus simples, on peut citer l'addition de réels inférieurs et l'injection des nombres rationnels dans les réels inférieurs. Nous vérifions que ces opérations conservent les relations d'ordre large et stricte comme sur des ensembles plus standards de nombres réels. Nous définissons également la multiplication d'un réel inférieur par un nombre rationnel strictement positif q (s'il ne l'est pas, la multiplication d'un réel inférieur par q est un réel supérieur). Enfin, nous définissons la soustraction d'un nombre rationnel positif à un réel inférieur.

Ensuite, nous avons vérifié la distributivité de ces opérations les unes par rapport aux autres, que ce soit entre l'addition et la multiplication par un rationnel, entre l'addition et l'injection depuis $\mathbb{Q}$ ou entre la multiplication par un rationnel et l'injection depuis $\mathbb{Q}$.

Enfin, pour obtenir une structure de treillis, nous définissons deux opérateurs $\cap_{\mathbb{R}_l}$ et $\cup_{\mathbb{R}_l}$ sur les réels inférieurs puis vérifions leur distributivité vis-à-vis des autres opérations. Dans le cas de $\mathbb{R}_l^+$, on obtient une propriété très forte :

Lemme 3. *$\mathbb{R}_l^+$ est un treillis distributif et un semi-groupe ordonné.*

Comme souvent en mathématiques constructives, la manipulation de ces réels inférieurs est plus difficile qu'avec une axiomatisation classique des réels, comme dans la bibliothèque réelle standard de Coq [22]. Cela se répercute dans notre formalisation de programmes probabilistes par rapport à ceux d'ALEA, qui se base sur les réels de la bibliothèque standard et sur une axiomatisation de $[0, 1]$. La difficulté principale est l'absence de développement conséquent sur

2. Au contraire, en logique classique, ce sont des nombres réels.

$\mathbb{Q}$ qui soit compatible avec HoTT. Bien que les nombres rationnels de la théorie homotopique des types soient décidables et donc isomorphes à ceux de la bibliothèque standard, ils ne sont pas facilement portables vers HoTT, qui ne repose pas sur la biliothèque standard.

Nous utilisons une adaptation pour HoTT des nombres rationnels de MathClasses (dans HoTTClasses [10]). Elle n'est cependant pas complète et ne permet pas aisément de raisonner automatiquement sur $\mathbb{Q}$. HoTTClasses comporte une tactique `ring`, mais pas de tactique `field`. Ces difficultés soulèvent un problème plus général, que sont les problèmes de portabilité entre bibliothèques dans la communauté des assistants de preuves. Le développement autour des réels inférieurs est substantiel (environ 2 000 lignes de code Coq) et pourrait être utilisé pour d'autres applications faisant intervenir des fonctions semi-continues.

4 Intégrales inférieures et valuations

Dans cette section, nous souhaitons développer une alternative à la théorie de la mesure et de l'intégration constructive et compatible avec le formalisme de la topologie synthétique.

4.1 Ouverts d'un *hSet*

En topologie synthétique, les ouverts d'un type sont les objets centraux, qui permettent d'isoler les sous-ensembles d'un type qui peuvent être valués (classiquement, les ensembles mesurables). Nous définissons les ouverts d'un type $A : hSet$ comme les objets de type $\mathcal{O}_A = A \to \mathbb{S}$. En effet, comme expliqué en Section 2.2, ces fonctions sont en bijection avec les ouverts de A.

Nous définissons des opérations $\cup_{\mathcal{O}_A}$ et $\cap_{\mathcal{O}_A}$ et prouvons que $\mathcal{O}_A$ est un treillis distributif. Par ailleurs, la structure d'ω-cpo de l'ensemble d'arrivée $\mathbb{S}$ permet de prouver que $\mathcal{O}_A$ est un ω-cpo.

4.2 Fonctions positives sur un *hSet*

Indépendamment, nous définissons également les fonctions de $mf(A) = A \to \mathbb{R}_l^+$ et les munissons d'une addition. Nous obtenons un semi-groupe. Ensuite, nous munissons ces fonctions de la structure d'ω-cpo, qui est une conséquence de la structure d'ω-cpo de l'ensemble d'arrivée $\mathbb{R}_l^+$.

4.3 Valuations

En théorie des domaines, il existe une notion analogue aux mesures en analyse classique : la notion de valuations. Une valuation sur $A : hSet$ assigne une "quantité" aux ouverts de A. Contrairement à une mesure, une valuation est semi-continue inférieurement (et non continue). Il est donc naturel que son ensemble d'arrivée soit les réels inférieurs positifs $\mathbb{R}_l^+$. Une valuation de probabilité est par ailleurs inférieure à $1_{\mathbb{R}_l^+}$, de façon analogue aux mesures de probabilité.

Définition 2. *Une **valuation** de probabilité sur $A : hSet$ est une fonction $\mu : \mathcal{O}(A) \to \mathbb{R}_l^+$ qui est :*

- *Définie :* $\mu(\emptyset) = 0$;
- *Modulaire :* $\forall U, V \in \mathcal{O}(A), \mu(U) + \mu(V) = \mu(U \cup V) + \mu(U \cap V)$;
- *Monotone :* $\forall U, V \in \mathcal{O}(A), U \subseteq V \Rightarrow \mu(U) \leq \mu(V)$;
- *(Sous)-probabiliste :* $\mu(A) \leq 1$;

- *Continue :* $\forall f : \mathbb{N} \to \mathcal{O}(A), \forall n, f_n \leq f_{n+1} \Rightarrow \mu(\sup(\lambda n.f(n)) \leq sup_{\mathbb{R}_l^+}(\lambda n.\mu(f(n)))$.

On vérifie que le type des valuations est bien un *hSet*, tout comme les type des réels inférieurs qui en est l'ensemble d'arrivée. En découlent, sans axiomes, des propriétés d'extensionnalité utiles pour les preuves.

4.4 Intégrales inférieures positives

Indépendamment, nous définissons la théorie des intégrales inférieures, l'équivalent semi-continu inférieurement des intégrales en théorie de la mesure de Lebesgue. Vickers [31] définit ces intégrales inférieures sur des locales, espaces topologiques abstraits en théorie des catégories vérifiant certaines propriétés. Notre construction sur *hSet* est plus générale, car la catégorie *hSet* vérifie des hypothèses plus faibles.

Définition 3. *Une* ***intégrale inférieure positive*** *sur A est une fonction $\mathcal{I} : mf(A) \to \mathbb{R}_l^+$ qui est :*

- *Définie :* $\mathcal{I}(0_{mf(A)}) = 0$;
- *Additive :* $\forall f, g \in mf(A), \mathcal{I}(f) + \mathcal{I}(g) = \mathcal{I}(f + g)$;
- *Monotone :* $\forall f, g \in mf(A), f \leq g \Rightarrow \mathcal{I}(f) \leq \mathcal{I}(g)$;
- *Sous-probabiliste :* $\mathcal{I}(1_{mf(A)}) \leq 1$;
- *Continue :* $\forall f : \mathbb{N} \to mf(A), \forall n, f_n \leq f_{n+1} \Rightarrow \mu(\sup(\lambda n.f(n)) \leq \sup(\lambda n.\mu(f(n)))$.

Sur les intégrales comme sur les valuations, on exhibe des propriétés intéressantes. Intégrales et valuations sont munies d'une structure d'ω-cpo, ce qui permet d'interpréter des valuations ou des intégrales définies récursivement (voir Section 5.2). L'extensionnalité fonctionnelle disponible dans HoTT permet également de prouver que l'égalité sur les valuations et les intégrales est un prédicat de *hProp* (niveau -1 d'homotopie). Dans la suite de l'article, on notera $IntPos(A)$ le type des intégrales inférieures positives sur A.

4.5 Plongement des intégrales dans les valuations

Le théorème de Riesz, standard en théorie de la mesure, donne une bijection entre les mesures et les intégrales. Généralement, ce résultat est démontré via l'axiome du choix, comme beaucoup de résultats d'analyse fonctionnelle et de théorie de la mesure. Constructivement, la preuve est plus difficile. Coquand et Spitters proposent une preuve constructive de ce théorème [7] mais leurs intégrales sont à valeurs dans les coupures de Dedekind. Cependant, les types sur lesquels ils raisonnent ont des hypothèses plus fortes, ce qui permet d'exhiber un isomorphisme entre des intégrales (et non des intégrales inférieures) et des valuations (à valeur dans les réels inférieurs). Pour obtenir une construction valable sur tous les *hSets*, on doit se restreindre aux intégrales inférieures. Vickers [31] prouve une variante du théorème de Riesz qui donne la bijection entre intégrales inférieures et valuations, mais dans un contexte plus abstrait, utilisant des locales. C'est donc plutôt de sa preuve que nous nous inspirons pour prouver une partie du théorème de Riesz dans le cadre de la topologie synthétique, mais avec des ensembles *hSet* et non des locales.

Prouver formellement le théorème de Riesz reviendrait à construire une intégrale "à la Lebesgue", ce qui constituerait un travail intéressant de formalisation. Cependant, dans notre cas de figure, ce n'est pas nécessaire. Nous devons construire nos programmes probabilistes comme valuations de probabilité. Ceci dit, la manipulation des valuations est plus difficile que la manipulation des intégrales. En effet, prouver la modularité est plus fastidieux que prouver l'additivité. Par ailleurs, les intégrales inférieures sont à valeurs dans $\mathbb{R}_l^+$ comme les fonctions

qu'elles mesurent, ce qui facilite leur composition. Nous prouvons donc l'une des implications du théorème de Riesz, à savoir qu'à partir de toute intégrale inférieure $\mathcal{I}$, on peut construire une valuation $\mu_{\mathcal{I}}$. Il faut définir l'objet en question et prouver que c'est bien une valuation.

Définition 4. *La valuation μ_I est définie comme $\mu_{\mathcal{I}}(U) := \mathcal{I}(\mathbb{1}_U)$.*

En mathématiques constructives, la construction de la fonction caractéristique $\mathbb{1}_U$ de type $mf(A)$ à partir d'un ouvert $U : \mathcal{O}(A)$ n'est pas immédiate (contrairement au cas classique) car on ne peut pas décider si un élément de A est dans U ou non. D'abord, on construit inductivement un plongement de $\mathbb{S}$ dans $\mathbb{R}_l^+$. C'est un argument supplémentaire en faveur de l'utilisation de $\mathbb{S}$ dans la construction de $\mathbb{R}_l$. Ensuite, pour $U : \mathcal{O}(A)$ donné, on obtient $\mathbb{1}_U : A \to \mathbb{R}_l^+$ à partir de cette construction. On doit prouver que les propriétés des valuations sont vérifiées. La modularité est la plus difficile, et nécessite un résultat intermédiaire :

Lemme 4. *Soit $U : \mathcal{O}(A)$. La fonction $\mathbb{1}_U$ vérifie :*

1. *(1) $\forall U, V \in \mathcal{O}(A), \mathbb{1}_{U\cap V} = \mathbb{1}_U \cap \mathbb{1}_V$;*
2. *(2) $\forall U, V \in \mathcal{O}(A), \mathbb{1}_{U\cup V} = \mathbb{1}_U \cup \mathbb{1}_V$;*
3. *(3) $\forall U, V \in \mathcal{O}(A), \mathbb{1}_{U\cup V} + \mathbb{1}_{U\cap V} = \mathbb{1}_U + \mathbb{1}_V$*

Les propriétés (1) et (2) sont prouvées en montrant que le plongement de $\mathbb{S}$ dans $\mathbb{R}_l^+$ préserve les intersections et les unions. Pour cela, on utilise les propriétés de $\mathbb{S}$ comme la distributivité des unions et des intersections par rapport aux supremums. La propriété (3) est ensuite dérivée de (1) et (2). On prouve par ailleurs le lemme de caractérisation suivante :

Lemme 5. *Supposons $U \in \mathcal{O}(A)$, $x \in A$. Alors : $\mathbb{1}_U(x) = 1_{\mathbb{R}_l^+} \Leftrightarrow U(x) = \top$.*

Enfin, on prouve le plongement des intégrales dans les valuations :

Thoreme 1. *Soit $\mathcal{I}$ une intégrale inférieure positive. Alors μ_I est une valuation.*

Démonstration. On vérifie que les différentes propriétés des valuations sont satisfaites :

- **Definition** : comme $\mathbb{1}$ est définie, $\mu_{\mathcal{I}}(\emptyset) = \mathcal{I}(\mathbb{1}_\emptyset) = \mathcal{I}(0_{mf(A)})$.
  Donc, comme $\mathcal{I}$ est une intégrale, $\mu_I(\emptyset) = 0$;
- **Monotonicité** : supposons $U \subseteq V$. On sait que $\mu_{\mathcal{I}}(U) = \mathcal{I}(\mathbb{1}_U)$ et $\mu_{\mathcal{I}}(V) = \mathcal{I}(\mathbb{1}_V)$.
  On prouve que $U \subseteq V \Rightarrow \mathbb{1}_U \leq \mathbb{1}_V$.
  Donc par monotonicité de $\mathcal{I}$, $\mathcal{I}(\mathbb{1}_U) \leq \mathcal{I}(\mathbb{1}_V)$ *i.e.* $\mu_{\mathcal{I}}(U) \leq \mu_{\mathcal{I}}(V)$;
- **Modularité** : $\mu_{\mathcal{I}}(U \cap V) + \mu_{\mathcal{I}}(U \cup V) =$
  $\mathcal{I}(\mathbb{1}_{U\cap V}) + \mathcal{I}(\mathbb{1}_{U\cup V}) = \mathcal{I}(\mathbb{1}_{U\cap V} + \mathbb{1}_{U\cup V})$ (car $\mathcal{I}$ est additive).
  Donc, par le Lemme 4, $\mu_{\mathcal{I}}(U \cap V) + \mu_{\mathcal{I}}(U \cup V) = \mathcal{I}(\mathbb{1}_U + \mathbb{1}_V)$
  Donc, comme $\mathcal{I}$ est additive, $\mu_{\mathcal{I}}(U \cap V) + \mu_I(U \cup V) = \mu_{\mathcal{I}}(U) + \mu_{\mathcal{I}}(V)$;
- **Probabilité** : comme $\mathcal{I}$ est une intégrale, $\mu_{\mathcal{I}}(A) = \mathcal{I}(\mathbb{1}_A) \leq \mathcal{I}(1_{mf(A)}) \leq 1$;
- **Continuité** : Soit $f : \mathbb{N} \to mf(A)$ une suite croissante.
  Par définition, $\mu_{\mathcal{I}}(\sup(\lambda n.f(n))) = \mathcal{I}(\mathbb{1}_{\sup(\lambda n.f(n))})$.
  On prouve que : $\mathbb{1}_{\sup(\lambda n.f(n))} = \sup(\lambda n.\mathbb{1}_{f(n)})$ et par continuité de $\mathcal{I}$:
  $\mathcal{I}(\sup(\lambda n.\mathbb{1}_{f(n)})) \leq \sup(\lambda n.\mathcal{I}(\lambda n.\mathbb{1}_{f(n)})) = \sup(\lambda n.\mu_{\mathcal{I}}(f(n)))$

□

Remarque 1. *On définit un ouvert $\mathcal{D}(f, q)$: Soit $f \in mf(A)$, $q \in \mathbb{Q}_+$. On définit l'ouvert $\mathcal{D}(f, q) \in \mathcal{O}(A)$ comme :*

$$\mathcal{D}(f, q) := \lambda x.(q < f(x)).$$

On peut alors définir $\mathcal{I}_\mu(f)$ en utilisant une subdivision du codomaine de f par des rationnels :

$$\mathcal{I}_\mu(f) = \sup_{n\in\mathbb{N}} \left(\frac{1}{n} \sum_{i=0}^{n} \mu \left(\mathcal{D}(f, \frac{i}{n}) \right) \right).$$

Prouver que c'est une intégrale inférieure complèterait le théorème de Riesz à condition de vérifier également la bijection suivante :

$$(\forall \mathcal{J}, \mathcal{I}_{\mu_\mathcal{J}} = \mathcal{J}) \wedge (\forall \nu, \mu_{\mathcal{I}_\nu} = \nu).$$

5 Interprétation d'un langage probabiliste

La topologie synthétique constitue une approche adaptée aux raisonnements mathématiques constructifs et peut servir de base à des applications diverses. Dans cette section, nous utilisons une construction monadique proche de celle de la bibliothèque ALEA afin d'obtenir des fonctionnalités similaires [2]. Ainsi pouvons-nous utiliser notre formalisme constructif pour interpréter un mini-langage probabiliste.

5.1 Monade de Giry sur les intégrales inférieures

La construction monadique de Giry consiste à envoyer un objet sur l'ensemble des valuations sur cet objet [11]. La classe d'espaces sur laquelle on définit cette monade peut être vue comme une catégorie adaptée. Généralement, la monade de Giry est définie sur des catégories comme les espaces polonais (espaces métriques enrichis de propriétés). Dans notre cas, la catégorie choisie est le type *hSet* de HoTT. En outre, comme nous avons prouvé le plongement des valuations dans les intégrales inférieures (voir Section 4.5), nous définissons une variante de la monade de Giry qui envoie un espace de type *hSet* sur l'ensemble de ses intégrales inférieures. Ce choix facilite la formalisation car l'ensemble des intégrales inférieures correspond au type $((A \to \mathbb{R}_l^+) \to \mathbb{R}_l^+)$ alors que celui des valuations correspond au type $((A \to \mathbb{S}) \to \mathbb{R}_l^+)$. En effet, le fait d'avoir le même espace $\mathbb{R}_l^+$ pour l'ensemble d'arrivée des fonctions évaluées et pour l'ensemble d'arrivée des intégrales facilite les compositions d'intégrales, et permet plus simplement de prouver qu'on a bien une monade.

Définition 5. *Nous définissons la monade des intégrales inférieures comme le triplet $(\mathcal{M}, unit, bind)$ tel que :*

- $\mathcal{M}(A : hSet) := IntPos(A)$;
- $unit(A : hSet) : A \to IntPosA := \lambda\ x : A.\ \lambda\ f : mf(A) \Rightarrow f(x)$;
- $bind(A : hSet)(B : hSet) : IntPos(A) \to (A \to IntPosB) \to IntPosB :=$
  $\lambda\ \mathcal{I} \Rightarrow \lambda\ \mathcal{F} \Rightarrow (\lambda\ f : mf(A) \Rightarrow \mathcal{I}(\lambda\ x : A \Rightarrow (\mathcal{F}(x))(f)))$.

Nous prouvons ensuite qu'il s'agit bien d'une monade, *i.e.* qu'elle vérifie les propriétés suivantes :

- $bind(unit(x), \mathcal{F}) = \mathcal{F}(x)$;
- $\forall\ (x : A)\ F$, $bind(unit(x), \mathcal{F}) = \mathcal{F}(x)$;
- $\forall\ \mu, \mathcal{F}_1, \mathcal{F}_2$, $bind(bind(\mu, \mathcal{F}_1), \mathcal{F}_2) = bind(\mu, \lambda x\ .\ bind(\mathcal{F}_1(x), \mathcal{F}_2))$.

5.2 Interprétation du langage *Rml*

Rml est un mini-langage fonctionnel probabiliste défini par Audebaud et Paulin dans ALEA [2]. Nous souhaitons être en mesure d'interpréter le langage *Rml* via le λ-calcul de

Moggi [23]. D'un point de vue théorique, il suffit que la catégorie sur laquelle est définie la monade soit cartésienne close, ce qui est le cas de la catégorie *hSet* (de par l'extensionnalité fonctionnelle disponible dans HoTT). En outre, la structure d'ω-cpo sur le type des intégrales inférieures permet l'interprétation de points fixes et de fonctions récursives. Ainsi pouvons-nous, sans ajouter d'axiomes à HoTT, interpréter un langage probabiliste proche de *Rml*, incluant la conditionnelle, l'abstraction et l'application de fonction et les fonctions récursives ($t \rightsquigarrow T$ signifie que t que le terme s'interprète en T, $[\![x]\!]$ est l'interprétation de la variable x) :

- $v \rightsquigarrow unit(v)$;
- let $x = a$ in $b \rightsquigarrow bind\ [\![a]\!]\ (\lambda\ x.[\![b]\!])$;
- $f(e_1, \ldots, e_n) \rightsquigarrow bind\ [\![e_1]\!]\ (\lambda\ x_1\ \ldots\ bind\ [\![e_n]\!]\ (\lambda\ x_n.[\![f]\!](x_1, \ldots, x_n))$;
- if b then c_1 else $c_2 \rightsquigarrow bind\ [\![b]\!]\ (\lambda\ x : bool\ .$ if x then $[\![c_1]\!]$ else $[\![c_2]\!])$;
- let rec $f(x) = e \rightsquigarrow fix\ (\lambda\ [\![f]\!] \Rightarrow \lambda\ x \Rightarrow [\![p]\!])$.

Cette interprétation permet de définir des programmes probabilistes simples en tant qu'intégrales inférieures. Nous définissons un "Pile ou Face" (`flip`) sur le type *bool*, le tirage aléatoire d'un nombre (`random`) sur le type *nat* ou encore la marche aléatoire (`walk`) comme programme probabiliste récursif. Les types *nat* et *bool* sont des exemples naturels de types qui sont des *hSet*, ce qui se prouve facilement. C'est pourquoi nous commençons par raisonner sur des programmes simples définis comme intégrales sur ces types.

- **Pile ou Face :**

  ```
  Definition flip : IntPos bool := fun f : mf bool ⇒ (1/2) ∗ (f true) + (1/2) ∗ (f false).
  ```

  Pour plus de généralité, nous prouvons que la combinaison convexe de deux intégrales est une intégrale (si $\mathcal{I}$ et $\mathcal{J}$ sont des intégrales inférieures, et $p, q \in \mathbb{Q}_+$ tels que $p+q \leq 1$ alors $p\mathcal{I} + q\mathcal{J}$ est une intégrale inférieure). Le programme `flip` est un cas particulier de combinaison convexe d'intégrales inférieures.
- **Tirage aléatoire d'un nombre :**

  ```
  Definition random (N : nat) : IntPos nat.
  exists (fun f: mf nat ⇒ (1 / N+1) sum_(i=0)^(N) f(i)).
  ```

  Pour prouver qu'il s'agit d'une intégrale, nous rencontrons essentiellement des difficultés liées aux raisonnements calculatoires, notamment la manipulation des réels inférieurs. Il est courant, en mathématiques constructives, de rencontrer des structures de données inhabituelles et difficiles à manipuler. Ici, la difficulté est plus importante que pour `flip` car la sommation et les fractions rendent les preuves fastidieuses.
- **Marche aléatoire :**

  ```
  let rec walk x = if flip() then x else walk (S x).
  ```

Des difficultés inhérentes au constructivisme sont rencontrées en amont de l'utilisation de ces programmes. Prouver que ce sont des intégrales inférieures nécessite la manipulation des nombres réels inférieurs, et donc d'ensembles de nombres rationnels. Il y a, aussi bien dans la bibliothèque standard de Coq que dans HoTTClasses, peu de techniques automatiques pour raisonner sur les nombres rationnels. Plus généralement, HoTTClasses comporte une tactique automatique `ring` pour les anneaux, mais pas de tactique automatique `field` pour les corps. Néanmoins, une fois les briques de base de ces programmes définies, il n'y a pas de difficulté particulière à les utiliser ou à les combiner.

6 Conclusions et perspectives

Nous avons proposé une combinaison de la théorie homotopique et de la topologie synthétique permettant de résoudre des problèmes de théorie de la mesure et de l'intégration constructive. Nous avons formalisé nos résultats dans l'assistant de preuves Coq et avons utilisé notre formalisme pour définir une sémantique axiomatique pour les programmes probabilistes. Cette alternative à des bibliothèques comme ALEA allège certains mécanismes, de par l'existence des types quotients et de l'extensionalité fonctionnelle dans la théorie homotopique des types. Par ailleurs, du point de vue de la théorie des catégories, nous proposons une extension de la construction monadique de Giry à la topologie synthétique. Le code est disponible en ligne [3] et comporte plus de 6 000 lignes de code Coq. La formalisation inclut la vérification des fondements de la topologie synthétique, une construction des réels inférieurs avec un nombre conséquent de résultats et une construction des ouverts d'un type A comme applications de type $A \to \mathbb{S}$. À partir de ces objets, nous avons construit une théorie des valuations et des intégrales inférieures, que nous appliquons à l'interprétation de programmes probabilistes.

Notre construction s'étend aux probabilités sur des types continus. Dans ALEA, le formalisme ne permet de raisonner que sur des probabilités sur des types discrets (nat, $bool$, ...). Cela revient en effet à prendre les valuations de type $(A \to bool) \to \mathbb{R}_l^+$. Les valuations de ce type assignent une mesure à tous les sous-ensembles de A, ce qui dans le cas $A = [0,1]$ contredit la théorie de Lebesgue, puisqu'on considère des objets (fonctions, évènements) non-ouverts, et donc non-mesurables en repassant à la théorie de la mesure. Au contraire, les éléments de $[0,1] \to \mathbb{S}$ correspondent exactement aux ouverts de $[0,1]$. Cependant, la construction d'une valuation sur $[0,1]$ comporte des difficultés. Nous devrons ajouter un axiome de topologie synthétique pour que $[0,1]$ soit compact [20] et couvrir tout ouvert de $[0,1]$ par une base dénombrable d'intervalles rationnels ouverts. Il faudra ensuite en prendre le supremum et montrer qu'il satisfait les propriétés d'une valuation. Par ailleurs, l'interprétation de programmes probabilistes sur $[0,1]$ nécessite de repasser aux intégrales inférieures, ce qui nécessite de prouver la réciproque du théorème 1. Cela revient à construire une intégrale au sens de Lebesgue, et donc à définir des notions intermédiaires comme les fonctions étagées.

La bibliothèque ALEA forme la base du système Certicrypt [6]. Bien que le système Easycrypt lui succédant ne l'utilise plus, l'idée générale reste proche. Sato [27] propose une extension de la sémantique du langage apWhile d'Easycrypt aux types continus. Nous espérons pouvoir faire le lien entre notre travail et une éventuelle formalisation de cette sémantique étendue.

Nous avons repris la sémantique axiomatique de *Rml* décrite dans ALEA [2]. Une autre perspective intéressante serait de gagner en généralité en faisant un plongement profond de *Rml* dans Coq. Cela permettrait de connecter notre sémantique dénotationnelle à une sémantique opérationnelle. Huang et Morrisett [15] utilisent des distributions "calculables" comme sémantique pour leur langage de programmation probabiliste, AugurV2. Nous pensons que leur sémantique peut être connectée à la nôtre grâce à des arguments de réalisabilité [21], et nous voudrions utiliser notre formalisation pour partiellement vérifier leur compilateur.

Remerciements

L'idée générale derrière ce travail a émergé lors d'une discussion entre Bas Spitters et Christine Paulin en 2014. Nous lui sommes reconnaissants pour avoir proposé à Florian Faissole de travailler sur les bases de ce sujet en fin d'année 2015. Merci aux reviewers anonymes pour leurs remarques.

3. https://github.com/FFaissole/Valuations/

Références

[1] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. Partiality, revisited - the partiality monad as a quotient inductive-inductive type. In *Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 534–549, 2017.

[2] Philippe Audebaud and Christine Paulin-Mohring. Proofs of randomized algorithms in Coq. In *MPC*, 2006.

[3] Andrej Bauer. First steps in synthetic computability theory. *Electron. Notes Theor. Comput. Sci.*, 155 :5–31, May 2006.

[4] Andrej Bauer, Jason Gross, Peter Lefanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT library. *arXiv :1610.04591*, 2016.

[5] Andrej Bauer and Davorin Lesnik. Metric spaces in synthetic topology. *Ann. Pure Appl. Logic*, 163 :87–100, 2012.

[6] Santiago Zanella Béguelin. *Formal certification of game-based cryptographic proofs.* PhD thesis, 2010.

[7] Thierry Coquand and Bas Spitters. Integrals and valuations. *Journal of Logic and Analysis*, 1(3) :1–22, 2009.

[8] Martín Escardó. Synthetic topology : of data types and classical spaces. *ENTCS*, 87 :21–156, 2004.

[9] Martín Escardó and Cory Knapp. Partial elements and recursion via dominances in univalent type theory. 2017.

[10] Geätan Gilbert. Formalising real numbers in homotopy type theory. *CPP'17*, 2016.

[11] Michele Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. 1982.

[12] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2) :95–152, 2010.

[13] Osman Hasan and Sofiène Tahar. *Formalization of Continuous Probability Distributions*, pages 3–18. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[14] Johannes Hölzl. Markov processes in isabelle/hol. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 100–111, New York, NY, USA, 2017. ACM.

[15] Daniel Huang and Greg Morrisett. An application of computable distributions to the semantics of probabilistic programming languages. In *European Symposium on Programming Languages and Systems*, pages 337–363. Springer, 2016.

[16] Martin Hyland. First steps in synthetic domain theory. In *Category Theory*, pages 131–156. Springer, 1991.

[17] Claire Jones and Gordon Plotkin. A probabilistic powerdomain of evaluations. In *LICS*, 1989.

[18] Dexter Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3) :328 – 350, 1981.

[19] Robbert Krebbers and Bas Spitters. Type classes for efficient exact real arithmetic in Coq. *LMCS*, 9(1 :1) :1–27, 2013.

[20] Davorin Lešnik. Synthetic topology and constructive metric spaces. *Diss. University of Ljubljana*, 2010.

[21] Peter Lietz. *From constructive mathematics to computable analysis via the realizability interpretation.* PhD thesis, Darmstadt, 2005.

[22] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique.* PhD thesis, Université Paris VI, décembre 2001.

[23] Eugenio Moggi. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Jun 1989.

[24] Russell O'Connor. *Certified Exact Transcendental Real Number Computation in Coq*, pages 246–261. Springer, 2008.

[25] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *ACM SIGPLAN Notices*, volume 37, pages 154–165. ACM, 2002.

[26] Egbert Rijke and Bas Spitters. Sets in homotopy type theory. In *MSCS, special issue : From type theory and homotopy theory to univalent foundations*, 2014.

[27] Tetsuya Sato. Approximate relational hoare logic for continuous random samplings. *ENTCS*, 325 :277–298, 2016.

[28] Michael Shulman. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *arXiv :1509.07584*, 2015.

[29] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *MSCS, special issue on 'Interactive theorem proving and the formalization of mathematics'*, 21 :1–31, 2011.

[30] The Univalent Foundations Program. *Homotopy Type Theory : Univalent Foundations for Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[31] Steven Vickers. A monad of valuation locales. 2011.

Vérification de programmes OCaml fortement impératifs avec Why3

Jean-Christophe Filliâtre[1,2], Mário Pereira[1,2], and Simão Melo de Sousa[3,4]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Saclay – Île-de-France, Orsay, F-91893
[3] Universidade da Beira Interior, Portugal
[4] LISP - Lab. de Informática, Sistemas e Paralelismo, Portugal

Résumé

Cet article présente une méthodologie pour prouver des programmes OCaml fortement impératifs avec l'outil de vérification déductive Why3. Pour un programme OCaml donné, un modèle mémoire spécifique est construit et on vérifie un programme Why3 qui le manipule. Une fois la preuve terminée, on utilise la capacité de Why3 à traduire ses programmes vers le langage OCaml, tout en remplaçant les opérations sur le modèle mémoire par les opérations correspondantes sur des types mutables d'OCaml. Cette méthode est mise à l'épreuve sur plusieurs exemples manipulant des listes chaînées et des graphes mutables.

1 Introduction

Depuis la version 4.03 du compilateur OCaml [1], il est possible de déclarer des types algébriques avec des composantes mutables. Ainsi, il est possible de définir un type de listes simplement chaînées de la façon suivante :

```
type 'a cell =
  | Nil
  | Cons of { mutable content: 'a; mutable next: 'a cell }
```

Un tel type évite l'indirection qui serait causée par le recours à deux types, un type somme d'une part et un type enregistrement d'autre part. D'autres solutions moins élégantes consisteraient à utiliser des valeurs récursives ou, pire encore, la fonction `Obj.magic` [2]. Avec un type comme le type `cell`, on obtient une représentation analogue à celle d'un code C ou Java, la valeur `Nil` jouant le rôle du pointeur `null`. Une différence notable, cependant, est que le système de type d'OCaml assure qu'on ne cherchera pas à accéder au champ `content` ou `next` d'une valeur `Nil`. Cette extension du langage OCaml ouvre des perspectives intéressantes en matière de programmation de structures fortement impératives. Comme il est notoire qu'on se trompe facilement en écrivant de tels programmes, nous proposons dans cet article une approche pour prouver des programmes OCaml avec des structures algébriques mutables.

Notre approche repose sur l'outil de vérification déductive Why3 [2], qui propose à la fois une logique, un langage de programmation et une bibliothèque adaptés à la preuve de programmes. Sa logique est une extension de la logique du premier ordre avec polymorphisme,

Ce travail a été en partie financé par la Fondation des Sciences et de la Technologie du Portugal (bourse FCT-SFRH/BD/99432/2014 et FCT-SFRH/BSAB/135039/2017), par le Laboratoire d'Informatique, Systèmes et Paralellisme (LISP FCT- UID/CEC/4668/2016) et par l'Agence Nationale de la Recherche (projet VOCAL ANR-15-CE25-008).

1. https://caml.inria.fr/pub/docs/manual-ocaml/extn.html#sec257
2. Ces différentes solutions sont discutées dans un article des JFLA 2014 [9] ; elles sont aujourd'hui devenues obsolètes pour la plupart du fait de cette extension du langage.

```
type 'a cell = Nil | Cons of { mutable content: 'a; mutable next: 'a cell; }

let get_next = function Nil -> assert false | Cons { next } -> next
let set_next l1 l2 = match l1 with Nil -> assert false | Cons c -> c.next <- l2

let concat l1 l2 =
  if l1 == Nil then l2 else
  begin
    let n = ref l1 in
    while not (get_next !n == Nil) do n := get_next !n done;
    set_next !n l2;
    l1
  end
```

FIGURE 1 – Concaténation en place de listes chaînées.

types algébriques, définitions récursives et prédicats inductifs [8]. Son langage de programmation, WhyML, est un sous-ensemble du langage OCaml auquel a été ajouté du code fantôme [11] et un contrôle statique des alias [10]. Sa bibliothèque propose à la fois des modules de spécification (entiers mathématiques, ensembles, etc.) et des modules de programmation (tableaux, entiers machine, etc.). L'outil Why3 extrait les conditions de vérification d'un programme WhyML à l'aide d'un calcul de plus faibles préconditions et les transmet à de nombreux démonstrateurs automatiques, dont la majorité des démonstrateurs SMT, et à des assistants de preuve interactifs si besoin. Une fois prouvé, un programme WhyML peut être traduit automatiquement en syntaxe OCaml par un mécanisme d'extraction analogue à celui de Coq. Nous montrerons aussi, par une systématisation de la construction des modèles mémoire, comment nous outillons notre méthodologie au dessus de Why3 pour obtenir automatiquement les modèles sur lesquels repose l'effort de spécification et de preuve.

Nous commençons par présenter notre approche dans la section 2, sur l'exemple simple de la concaténation en place de listes chaînées, puis sa validation expérimentale sur quatre études de cas plus complexes dans la section 4. Nous concluons en évoquant quelques perspectives.

2 Méthodologie

Nous illustrons notre approche avec la preuve d'un programme OCaml qui réalise la concaténation en place de deux listes chaînées. Ce programme est contenu dans la figure 1. Il est volontairement écrit dans un style très impératif, avec une boucle `while` et des références, mais pourrait tout aussi bien être écrit avec une fonction récursive. Les deux fonctions auxiliaires `get_next` et `set_next` sont là pour faciliter la manipulation du champ `next` d'une liste dont on sait qu'elle n'est pas égale à `Nil`. Notre objectif est de prouver la correction de la fonction `concat` avec Why3. En particulier, nous montrerons que les fonctions `get_next` et `set_next` ne sont jamais appelées avec `Nil` ou encore que la correction de la fonction `concat` repose sur l'hypothèse que les deux listes sont totalement disjointes. En effet, si les deux listes passées à `concat` partagent un suffixe, le résultat n'est plus une liste chaînée finie.

Modélisation des listes chaînées en Why3. Why3 n'autorise pas la définition d'un type tel que `cell`. Ceci est dû au fait que Why3 utilise un système de types avec effets pour déterminer statiquement tous les alias entre pointeurs dans le programme [10] et le type `cell` sort du cadre

de cette analyse statique. La solution consiste alors à modéliser le contenu du tas, sous la forme d'un ensemble de types et d'opérations pour allouer, lire et écrire dans la mémoire. Une telle idée est déjà employée dans des outils qui utilisent Why3 comme langage intermédiaire, par exemple Frama-C [7]. Contrairement à ces outils, cependant, nous construisons ici un modèle spécifique au type `cell` et nous continuons d'utiliser les autres types de Why3 lorsque c'est possible. Nous commençons par introduire un type non interprété `loc` pour représenter l'adresse mémoire d'un constructeur `Cons`. Le type `cell` reste un type algébrique.

```
type loc 'a
type cell 'a =
  | Nil
  | Cons (loc 'a)
```

Nous modélisons ensuite le contenu du tas à l'aide du type `mem` suivant :

```
type mem 'a = {
  mutable  content: loc 'a -> option 'a;
  mutable     next: loc 'a -> option (cell 'a);
} invariant { forall l. content l = None <-> next l = None }
```

Les champs `content` et `next` correspondent aux deux arguments du constructeur `Cons`. Ils sont introduits comme des fonctions du type `loc` vers des valeurs optionnelles [3]. Les valeurs pour lesquelles les champs `content` et `next` sont différents de `None` sont les valeurs allouées de la forme `Cons`. L'invariant associé au type `mem` assure que ces deux champs sont toujours `None` simultanément. Le fait de modéliser les champs `content` et `next` indépendamment l'un de l'autre est une façon efficace de traduire qu'une modification de l'un n'a pas d'incidence sur l'autre. Cette idée est due à Burstall et remonte à 1972 [3].

Pour opérer sur ce modèle mémoire, nous introduisons plusieurs fonctions : `alloc_cons` pour allouer un `Cons`, `get_content` et `get_next` pour lire, `set_content` et `set_next` pour écrire. Par exemple, la fonction `set_next` est ainsi déclarée :

```
val set_next (ghost mem: mem 'a) (c1 c2: cell 'a)  : unit
  requires { match c1 with Nil -> false | Cons l -> mem.next l <> None end }
  writes   { mem.next }
  ensures  { match c1 with
             | Nil    -> false
             | Cons l -> mem.next = (old mem.next)[l <- Some c2]
             end }
```

Elle reçoit la mémoire en premier argument. Cet argument est fantôme, car le modèle mémoire n'est pas destiné à apparaître au final dans le code extrait. La précondition exige que `l1` soit une valeur différente de `Nil` et allouée. La postcondition décrit comment la mémoire a été affectée.

Preuve de la fonction `concat`. Nous sommes maintenant en mesure d'écrire et de vérifier une version de la fonction `concat` à l'aide de ce modèle mémoire. Le code WhyML est donné dans la figure 2. Il est identique au code OCaml de la figure 1, à l'exception des éléments de spécification et de preuve (pré- et postcondition, code fantôme, invariants et variant de boucle).

3. Le type `option` est défini dans la bibliothèque de Why3. Il est identique à celui d'OCaml, c'est-à-dire `type option 'a = None | Some 'a`.

```
  predicate is_list_from_to (mem: mem 'a)
    (from : cell 'a) (s: view 'a) (until: cell 'a) =
    let n = length s in
    n = 0 /\ from = until
  \/
    n > 0 /\ from = Cons s[0] /\
    (forall i. 0 <= i < n -> Cons s[i] <> until) /\
    distinct s /\
    (forall i. 0 <= i < n - 1 -> mem.next s[i] = Some (Cons s[i+1])) /\
    mem.next s[n-1] = Some until

predicate disjoint (s1 s2: seq 'a) =
  forall i j. 0 <= i < length s1 -> 0 <= j < length s2 -> s1[i] <> s2[j]

let concat
  (ghost mem: mem 'a) (l1 l2: cell 'a) (ghost s1 s2: view 'a) : cell 'a
  requires { is_list_from_to mem l1 s1 Nil }
  requires { is_list_from_to mem l2 s2 Nil }
  requires { disjoint_seq s1 s2 }
  ensures  { is_list_from_to mem result (s1 ++ s2) Nil }
= if l1 == Nil then
    l2
  else begin
    let n = ref l1 in
    let ghost step = ref 0 in
    let ghost tail1 = ref s1 in
    while not (get_next mem !n == Nil) do
      invariant { 0 <= !step }
      invariant { !n = Cons s1[!step] }
      invariant { is_list_from_to mem !n !tail1 Nil }
      invariant { !step + Seq.length !tail1 = Seq.length s1 }
      variant   { Seq.length !tail1 }
      n := get_next mem !n;
      incr step;
      tail1 := !tail1 [ 1 .. ]
    done;
    set_next mem !n l2;
    l1
  end
end
```

FIGURE 2 – Code Why3 de la fonction `concat`.

Commençons par expliquer la spécification de cette fonction. En précondition, nous exigeons que `l1` et `l2` soient des listes finies (lignes 17–18), c'est-à-dire terminées par `Nil`[4]. Il y a de multiples façons de l'exprimer. Nous choisissons ici de matérialiser tous les éléments de chacune des deux listes dans des arguments fantômes `s1` et `s2` de type `seq (cell 'a)`. Le type `seq` est défini dans la bibliothèque standard de Why3, avec des opérations comme l'accès au i-ième élément (noté s`[`i`]`), la longueur (notée `length`), la concaténation (notée `++`), etc. Le prédicat `is_list_from_to` (lignes 1–10) exprime que la liste partant de la cellule `from` est finie, se termine par la cellule `to`, ne contient pas de répétition donc en particulier de boucle (ligne 8) et est composée exactement des cellules contenues dans la liste `s`. Noter que cette définition inclut le cas d'une liste vide, lorsque `from` est égal à `to` et que `s` est la séquence vide (lignes 3–4). L'intérêt d'avoir matérialisé ainsi `s1` et `s2` est que nous pouvons les réutiliser ensuite dans d'autres aspects du contrat, par exemple pour exiger que les deux listes sont disjointes (ligne 19) et pour exprimer la postcondition (ligne 20).

Pour prouver que la fonction `concat` respecte ce contrat, on introduit un certain nombre d'invariants de boucle. Ces invariants gardent trace du fait que la variable `n` avance dans la liste `l1`. Ici, on a choisi de l'exprimer à l'aide d'une variable fantôme `step` qui représente l'indice de `n` dans la liste `l1` et d'une variable fantôme `tail1` qui représente le suffixe de `s1` restant à parcourir. On prouve également la terminaison de la boucle `while` à l'aide d'un variant, à savoir la longueur de la séquence `tail1`. Une fois passé à l'outil Why3, le code de la figure 2 est prouvé entièrement automatiquement en quelques secondes.

Extraction de code OCaml. Une fois la preuve terminée, la dernière étape consiste à traduire le programme Why3 en un programme OCaml. On utilise pour cela un mécanisme d'extraction automatique fourni par Why3. Il consiste à effacer le code fantôme et les annotations logiques, à traduire les constructions de WhyML vers les constructions d'OCaml et faire correspondre les symboles de la bibliothèque standard de Why3 avec ceux de la bibliothèque d'OCaml. Ce dernier aspect est matérialisé par un fichier texte, appelé *driver*, que l'utilisateur peut compléter pour ses propres besoins. Dans notre cas, il s'agit de traduire vers OCaml le type `cell` et toutes les constantes et opérations de notre modèle.

```
syntax type cell        "%1 SinglyLL.cell"
syntax function Nil     "SinglyLL.Nil"
syntax function Cons    "SinglyLL.Cons %1"
syntax val  alloc_cons  "SinglyLL.Cons { content = %1; next = %2 }"
syntax val  (==)        "%1 == %2"
syntax val  get_next    "SinglyLL.get_next %1"
syntax val  set_next    "SinglyLL.set_next %1 %2"
...
```

Le module OCaml `SinglyLL` contient ici la définition du type `cell` et les définitions des fonctions comme `get_next`, à la manière des premières lignes de la figure 1. Pour chaque symbole Why3, le code OCaml est donné sous la forme d'une chaîne de caractères, où `%`n sera remplacé par l'extraction du n-ième argument de ce symbole. Notons que `set_next` apparaît ici comme n'ayant plus que deux arguments, son premier argument ayant été supprimé de par son statut fantôme.

4. En toute rigueur, il n'est pas nécessaire d'exiger que `l2` soit finie. Cependant, l'exiger nous permettra de donner une spécification simple en terme de concaténation.

3 Automatisation

Nous avons développé un outil qui automatise la construction du modèle mémoire et du *driver* associé. Cet outil prend en entrée la définition d'un ou plusieurs types OCaml, mélangeant enregistrements et types algébriques. Pour chaque type contenant des composantes mutables, il construit un modèle mémoire associé analogue à celui présenté dans la section précédente. Par exemple, pour un type comme

```
type tree = { v: int; mutable sub: tree list }
```

il va construire

- un type abstrait `loc_tree` pour modéliser les valeurs de type `tree` ;
- un type `mem_tree` pour modéliser le contenu de la mémoire, de la forme

```
type mem_tree = {
  mutable   v: loc_tree -> option int;
  mutable sub: loc_tree -> option (list loc_tree);
} invariant { forall l. v l = None <-> sub l = None }
```

- une constante `empty_mem_tree` et une fonction d'allocation `mk_tree` ;
- des fonctions d'accès en lecture et écriture aux différents champs, à savoir ici `get_v`, `get_sub` et `set_sub` ;
- enfin, un *driver* pour l'extraction, qui envoie notamment le type `loc_tree` vers le type OCaml `tree` et les fonctions comme `get_v`, `set_v`, etc., vers des définitions OCaml.

Pour un type algébrique, l'outil va construire d'une part un type algébrique Why3 avec les mêmes constructeurs et d'autre part autant de modèles mémoire qu'il y a de constructeurs avec des composantes mutables. Par exemple, sur un type OCaml comme

```
type alg = Nothing | A of { mutable a: int } | B of { mutable b: bool }
```

l'outil va construire deux modèles mémoire distincts pour les deux enregistrements, avec notamment deux types `loc_A` et `loc_B`, et deux types `mem_A` et `mem_B`, ainsi qu'un type algébrique

```
type alg = Nothing | A loc_A | B loc_B
```

On aurait pu aussi construire un seul modèle mémoire, avec un seul type `loc_alg` et un seul type `mem_alg`. Mais distinguer les deux modèles nous permet d'obtenir gratuitement le fait qu'une modification d'une valeur de la forme `A` n'a pas d'incidence sur les valeurs de la forme `B` qui existent par ailleurs. C'est toujours l'idée de *components-as-arrays* de Burstall [3].

Il reste à la charge de l'utilisateur de réunir différents modèles mémoire dans un seul type Why3, s'il le souhaite, ou encore de définir des prédicats sur la base de cette modélisation pour les besoins de la spécification, comme le prédicat `is_list_from_to` dans la section précédente. Ceci ne saurait être automatisé. Nous en donnons des exemples dans la section suivante.

4 Études de cas

Cette section présente une validation expérimentale de notre approche, sous la forme de plusieurs études de cas. L'ensemble des fichiers Why3 de ces preuves peut être téléchargé à l'adresse http://why3.lri.fr/jfla-2018/.

```
1  type t 'a = {
2    mutable       length   : OneTime.t;
3    mutable       first    : cell 'a;
4    mutable       last     : cell 'a;
5    mutable ghost view     : seq 'a;
6    mutable ghost list     : seq (loc 'a);
7    mutable ghost used_mem : mem 'a
8  } invariant { length.OneTime.valid }
9    invariant { length = 0 -> first = last = Nil }
10   invariant { length > 0 -> first = Cons list[0] /\
11                             last = Cons list[length - 1] /\
12                             used_mem.next list[length - 1] = Some Nil }
13   invariant { forall x: loc 'a. T.mem x list <-> used_mem.next x <> None }
14   invariant { forall i. 0 <= i < length - 1 ->
15                  used_mem.next list[i] = Some (Cons list[i+1]) }
16   invariant { forall i. 0 <= i < length ->
17                  used_mem.contents list[i] = Some view[i] }
18   invariant { length = Seq.length view = Seq.length list }
19   invariant { distinct list }
```

FIGURE 3 – Type des files.

4.1 Module `Queue`

Notre premier exemple est celui du module `Queue` de la bibliothèque standard d'OCaml. Il s'agit d'une structure de file réalisée par une liste simplement chaînée, de la manière suivante :

```
type 'a cell = Nil | Cons of { content: 'a; mutable next: 'a cell }
type 'a t = { mutable length: int; mutable first: 'a cell; mutable last: 'a cell }
```

À chaque instant, on conserve un pointeur vers le premier élément de la liste (`first`) et un autre vers le dernier (`last`). Les éléments sont retirés de la file du côté de `first` et ajoutés du côté de `last`. On conserve par ailleurs le nombre total d'éléments de la file (`length`), pour éviter d'avoir à le recalculer.

Modélisation. Le type `cell` des listes chaînées est modélisé en Why3 exactement comme dans la section 2. La figure 3 contient la définition du type Why3 correspondant au type `Queue.t` d'OCaml. On retrouve les trois champs `length`, `first` et `last` du code OCaml, auxquels sont ajoutés trois champs fantômes contenant respectivement la séquence des pointeurs composants la liste chaînées (`list`), la séquence des valeurs contenues dans ces éléments (`view`) et la portion de la mémoire associée à la file (`used_mem`).

Notons, tout d'abord, que le champ `length` est modélisé par un type `OneTime` (ligne 2) d'entiers machines qui obéissent à des restrictions fortes pour éviter tout débordement arithmétique [5]. Les deux champs `list` et `view` dupliquent par convenance des informations déjà présentes dans `used_mem`. En effet, il suffit d'extraire les informations contenues dans la mémoire à partir du pointeur `first` pour retrouver l'intégralité des listes `view` et `list`. Des invariants sont là pour exprimer la cohérence entre ces trois champs. D'une manière générale, notre type `t` est équipé d'un certain nombre d'invariants. Ces invariants sont supposés vérifiés à l'entrée

de toute fonction manipulant une valeur de type `t` et doivent être établis à la sortie de toute fonction modifiant ou renvoyant une valeur de type `t`. Ils se décomposent ainsi :

- l'invariant (ligne 8) indique la validité de l'entier représentant la longueur (le type `OneTime` représente des entiers qui ne peuvent être utilisés que linéairement) ;
- dans le cas d'une file vide, les champs `first` et `last` sont tous deux `Nil` (ligne 9) ;
- dans le cas d'une file non vide, l'adresse mémoire contenue dans le champ `first` (resp. `last`) est bien la première adresse contenue dans la séquence `list` (resp. la dernière) et le dernier élément de la file (pointé par `last`) termine la séquence (son champ `next` est `nil`) (lignes 10 – 12) ;
- toute adresse répertoriée dans `list` correspond à une adresse vers une cellule mémoire allouée et vice-versa (ligne 13) ;
- le contenu des champs `list` et `view` est cohérent avec la file en mémoire (`used_mem`) et ont même taille (lignes 14 – 18) ;
- enfin, toutes les adresses impliquées dans la liste chaînée sont distinctes deux à deux (ligne 19). En particulier, ceci implique qu'il n'y a pas de cycle dans la liste (ligne 19).

Opérations sur les files. Le type des files étant donné, nous pouvons maintenant définir et prouver les opérations attendues sur les files. Nous illustrons ici la preuve de l'opération `transfer`, qui reçoit deux files q_1 et q_2 en paramètre, concatène tous les éléments de q_1 à la fin de q_2, puis vide q_1. Son code spécifié et annoté est donné figure 4. La précondition (ligne 2) exige que les deux files soient distinctes en mémoire, c'est-à-dire qu'elles ne partagent aucune cellule de type `cell`. En particulier, il n'est pas autorisé d'appeler `transfer` en lui passant deux fois la même file en argument. Le contrat ligne 3 indique que les seules valeurs modifiées par la fonction seront celle de q_1 et q_2. La postcondition (lignes 4 et 5) exprime la modification des contenus des deux files.

La fonction `transfer` n'a strictement rien à faire lorsque q_1 est vide (ligne 6). Sinon, elle distingue le cas où q_2 est vide (lignes 7–13) du cas général (lignes 15–22). Dans ce dernier cas, elle fait pointer le champ `next` du dernier élément de q_2 vers le premier de q_1 et réalise l'union des modèles mémoire de q_1 et q_2. La précondition de disjonction a ici son importance si on veut pouvoir prouver les invariants de type de la file une fois modifiée. Le champ `length` est mis à jour en faisant la somme des deux longueurs (ligne 15). Cela a pour effet d'annuler la validité de ces deux longueurs, mais l'une est remplacée par la somme (qui est valide) et l'autre par zéro (qui est valide également). Ceci est expliqué en détail dans l'article décrivant l'intérêt du module `OneTime` au regard des débordements de capacité arithmétique [5]. La preuve de `transfer` est réalisée entièrement automatiquement par un ensemble de plusieurs démonstrateurs de type SMT. Une fois le code Why3 extrait vers OCaml, en utilisant la même technique que celle présentée dans la section 2, on obtient un code très proche de celui de la bibliothèque standard d'OCaml, avec des performances en tout point identiques.

4.2 Parcours en profondeur

L'exemple suivant est celui d'un programme qui effectue un parcours en profondeur sur un graphe mutable. En OCaml les sommets du graphe sont représentés par le type suivant :

```
type node =
  | Null | Node of { mutable m: bool; mutable left: node; mutable right: node }
```

Nous choisissons ici une structure de graphe où sont associés à chaque sommet deux successeurs et une marque booléenne. Cette dernière est utilisée pour marquer les sommets déjà visités

```
let transfer (q1 q2: t 'a) : unit
  requires { disjoint_mem q1.used_mem q2.used_mem }
  writes   { q1, q2 }
  ensures  { q2.view = (old q2.view) ++ (old q1.view) }
  ensures  { q1.view = empty }
= if not (is_empty q1) then
    if is_empty q2 then begin
      q2.length, q1.length <- q1.length, OneTime.zero ();
      q2.first, q2.last <- q1.first, q1.last;
      q2.list <- q1.list;
      q2.view <- q1.view;
      q2.used_mem, q1.used_mem <- q1.used_mem, empty_memory;
      q1.first, q1.last, q1.list, q1.view <- Nil, Nil, Seq.empty, Seq.empty;
    end else begin
      let len = OneTime.add q2.length q1.length in
      q2.length, q1.length <- len, OneTime.zero ();
      set_next q2.used_mem q2.last q1.first;
      q2.last, q2.list, q2.view <-
        q1.last, q2.list ++ q1.list, q2.view ++ q1.view;
      q2.used_mem, q1.used_mem <-
        mem_union q2.used_mem q1.used_mem, empty_memory;
      q1.first, q1.last, q1.list, q1.view <- Nil, Nil, Seq.empty, Seq.empty
    end
```

FIGURE 4 – Code Why3 de la fonction `transfer`.

pendant le parcours. Notre outil (décrit dans la section 3) introduit deux types `loc` et `node`, de la manière suivante,

```
type loc
type node = | Null | Node loc
```

ainsi qu'un type `mem` regroupant les valeurs des trois champs

```
type mem = {
  mutable     m: loc -> option bool;
  mutable  left: loc -> option node;
  mutable right: loc -> option node;
} invariant { forall l. m l <> None <-> left l <> None <-> right l <> None }
```

et des fonctions `get` et `set` pour chacun des trois champs.

On en vient maintenant à la preuve du parcours en profondeur. Nous introduisons une table globale `busy` pour garder trace de tous les sommets dont l'exploration est en cours :

```
val ghost busy: ref (loc -> bool)
```

Dans la littérature, ces sommets sont désignés comme des sommets *gris*.

La notion de chemin entre deux sommets joue un rôle crucial dans la spécification d'un parcours en profondeur. Nous la définissons de la manière suivante :

```
predicate edge (left right: loc -> option node) (x y: node) =
  match x with
  | Null   -> false
  | Node l -> left l = Some y \/ right l = Some y
  end

inductive path (left right: loc -> option node) (x y: node) =
| path_nil:  forall x l r. path l r x x
| path_cons: forall x y z l r. path l r x y -> edge l r y z -> path l r x z
```

Le prédicat `path left right x y` signifie qu'il y a un chemin entre les sommets `x` et `y`, les valeurs des successeurs gauches et droits étant déterminées par les dictionnaires `left` et `right`. Nous sommes maintenant en mesure d'implémenter et spécifier une fonction `dfs` qui réalise le parcours en profondeur. Le profil de cette fonction est le suivant :

```
let rec dfs (ghost mem: mem) (c: node) (ghost root: node) : unit
  if not (c = Null) && not (get_m mem c) then begin
    let ghost l = match c with Null -> absurd | Node l -> l end in
    ghost set busy l True;
    set_m mem c True;
    dfs mem (get_left mem c) root;
    dfs mem (get_right mem c) root;
    ghost set busy l False;
    ()
  end
```

L'argument `c` représente le sommet courant et l'argument fantôme `root` le sommet racine d'où est parti le parcours en profondeur. La précondition de la fonction `dfs` est divisée en trois parties : (i) tous les sommets dans `busy` sont bien coloriés

```
  requires { well_colored mem.left mem.right mem.m !busy }
```

c'est-à-dire, pour tout arc $x \to y$ avec y différent de `Null`, soit x a un statut `busy`, soit s'il est marqué alors y est aussi marqué ; (ii) seuls des sommets atteignables à partir de la racine sont marqués

```
  requires { only_descendants_are_marked root mem.left mem.right mem.m }
```

(iii) enfin, nous exigeons qu'il existe un chemin entre la racine et le sommet `c`

```
  requires { path mem.left mem.right root c }
```

Par manque de place, nous ne montrons pas ici les définitions des prédicats utilisés dans la spécification de la fonction `dfs`. Le lecteur intéressé peut consulter l'appendice en ligne de cet article. Quant à la postcondition de cette fonction, les propriétés `well_colored` et `only_descendants_are_marked` doivent être préservées après l'exécution du parcours

```
  ensures  { well_colored mem.left mem.right mem.m !busy }
  ensures  { only_descendants_are_marked root mem.left mem.right mem.m }
```

Ce sont donc des propriétés invariantes de l'exécution récursive de la fonction `dfs`. À la fin de l'exécution de `dfs` nous devons aussi montrer que si un sommet était déjà marqué avant l'exécution de `dfs`, alors il reste marqué après le parcours

```
  ensures  { forall l. (old mem).m[l] = Some True -> mem.m[l] = Some True }
```

et que si un sommet a un statut `busy` à la fin, c'est parce que son statut était `busy` avant le parcours :

```
  ensures  { forall l. !busy[l] = True -> (old !busy)[l] = True }
```

La postcondition s'achève en disant que si `c` est différent de `Null`, alors il sera bien marqué

```
  ensures  { match c with Null -> true | Node l -> mem.m[l] = Some True end }
```

Notons que si jamais `root` pointe vers la racine d'un graphe infini, cette fonction ne terminera pas. En Why3, on marque une fonction comme potentiellement divergente en ajoutant à sa spécification la clause `diverges`. Si le graphe était fini, et cette information donnée en précondition de `dfs`, nous pourrions alors prouver la terminaison en ajoutant un variant adéquat. C'est ce qui a été fait, par exemple, dans la preuve de correction d'une implémentation de l'algorithme de Shorr-Waite dont il est question dans la section suivante.

Le code de `dfs` est tout à fait classique : si `c` n'est pas `Null` et n'est pas marqué, le programme le marque et s'appelle récursivement sur les deux successeurs de `c`. Pour bien respecter la postcondition de `dfs`, nous prenons soin d'affecter à `c` un statut `busy` avant les appels récursifs sur les successeurs et de l'effacer ensuite. Nous terminons cet exemple par une fonction `mark` qui encapsule le parcours en profondeur à partir d'une racine donnée :

```
let mark (ghost mem: mem) (root: node) : unit
  requires { forall l. mem.m[l] = Some False /\ !busy[l] = False }
  diverges
  ensures  { only_descendants_are_marked root mem.left mem.right mem.m }
  ensures  { all_descendants_are_marked root mem.left mem.right mem.m }
  ensures  { forall l. !busy[l] = False }
=
  dfs mem root root
```

Cette fonction exige qu'au début de l'exécution aucun sommet ne soit marqué (ni même avec un statut `busy`). Au final, seuls les sommets atteignables à partir de la racine sont marqués (correction) et tous les sommets atteignables sont marqués (complétude). Par ailleurs, il ne subsiste aucun sommet avec un statut `busy`. Toutes les conditions de vérifications générées pour les fonctions `dfs` et `mark`, ainsi que pour quelques lemmes auxiliaires, sont automatiquement prouvées en utilisant une combinaison des démonstrateurs Alt-Ergo, CVC4 et Z3.

4.3 Autres exemples

Nous avons validé notre méthodologie sur plusieurs autres exemples que l'espace ne nous laisse pas ici décrire convenablement en détail. Nous présentons ici brièvement le tri fusion en place de listes simplement chaînées et l'algorithme de Schorr-Waite [18]. Ces développements peuvent être trouvés dans l'archive accompagnant la publication.

Tri fusion. Dans cet exemple, on prouve un tri fusion en place sur des listes simplement chaînées. Nous reprenons donc naturellement le modèle mémoire qui nous est maintenant familier (sections 2 et 4.1) et en particulier le prédicat `is_list_from_to` de la figure 2. Pour *encadrer* les effets des différentes fonctions, nous avons aussi introduit des prédicats tels que le prédicat `frame_mem` suivant qui exprime que deux mémoires coïncident sur un ensemble d'adresses :

```
predicate frame_mem (m1 m2: mem 'a) (s: seq (cell 'a)) =
  forall i. 0 <= i < length s ->
    m1.next s[i] = m2.next s[i] /\ m1.content s[i] = m2.content s[i]
```

On montre alors que certaines propriétés sont préservées lorsque la mémoire n'est pas modifiée aux adresses concernées. Par exemple, la préservation du prédicat `is_list_from_to` est exprimée par le lemme suivant :

```
lemma frame_is_list: forall m1 m2: mem 'a, s: view 'a, l: loc 'a.
  is_list_from_to m1 l s Nil -> frame_mem m1 m2 s ->
  is_list_from_to m2 l s Nil
```

Pendant la preuve d'un programme, un tel lemme est typiquement utilisé avec deux états de la mémoire correspondants à deux moments de l'exécution.

Algorithme de Schorr-Waite. À l'instar de l'algorithme de parcours en profondeur, il s'agit d'un algorithme de marquage des nœuds d'un graphe, mais cette fois-ci sans utiliser de mémoire auxiliaire. Il effectue un parcours en profondeur en utilisant le graphe lui-même comme une pile, le modifiant et le restaurant au fur et à mesure du parcours.

Nous avons prouvé une version de l'algorithme de Schorr-Waite sur un graphe où chaque sommet possède exactement deux pointeurs fils et une marque mutable, comme dans le parcours en profondeur vu plus haut section 4.2, mais aussi une seconde marque booléenne.

```
type node =
  | Null
  | Node of { mutable m: bool; mutable c: bool;
              mutable left: node; mutable right: node }
```

Ce champ `c` est utilisé pendant le parcours pour savoir lequel des deux fils est en train d'être visité. Pour les besoins de la preuve, nous déclarons un dictionnaire global qui associe à chaque nœud le chemin du graphe de la racine jusqu'à ce nœud :

```
val ghost path_from_root: ref (loc -> list loc)
```

Pour raisonner sur les chemins trouvés pendant l'algorithme, nous adaptons le prédicat `path` présenté précédemment en lui ajoutant la liste des sommets intermédiaires :

```
inductive path (left right: loc -> loc) (x y: loc) (p: list loc) = ...
```

La liste `p` garde trace des sommets du chemin entre `x` et `y`, en excluant `y`.

Comme pour le parcours en profondeur, on montre que l'algorithme de Schorr-Waite marque, à partir d'un graphe où toutes les marques sont `false`, exactement tous les nœuds atteignables. On montre également qu'il restaure bien la structure de graphe à l'identique. Enfin, on prouve la terminaison de cet algorithme, sous l'hypothèse que le graphe est fini.

5 Travaux connexes

L'idée d'utiliser un modèle mémoire explicite pour vérifier des programmes impératifs n'est pas nouvelle. Le greffon *Jessie* de l'analyseur Frama-C, par exemple, traduit un programme C et sa spécification écrite en ACSL [1] vers le langage de Why3. À la différence de notre approche, *Jessie* construit un modèle mémoire généraliste, indépendamment du programme à vérifier. Un ancêtre de *Jessie*, l'outil Caduceus [12], avait été utilisé pour faire une preuve

formelle de l'algorithme de Schorr-Waite [13], dont nous nous sommes fortement inspirés pour faire la preuve présentée dans la section 4.3.

Une alternative pour prouver des programmes manipulant le tas consiste à utiliser des outils possédant leur propre modèle mémoire interne. À cet égard, nous pouvons citer Dafny [15], VeriFast [14] ou encore VCC [6]. Dans ces outils, la modélisation et la manipulation de la mémoire restent invisibles pour l'utilisateur, contrairement à notre approche.

La vérification de programmes avec pointeurs est devenue populaire avec l'introduction de la logique de séparation [17]. Cette logique de programme propose un formalisme simple pour raisonner sur des structures de données mutables, en étendant la logique de Hoare classique avec de nouveaux constructeurs logiques pour établir que différentes formules sont vérifiées par des zones disjointes de la mémoire. Ces dernières années, plusieurs outils utilisant la logique de séparation ont été développés et le formalisme a gagné en maturité. Récemment, Charguéraud et Pottier ont démontré comment la logique de séparation peut être utilisée non seulement pour vérifier des propriétés fonctionnelles d'un code, mais aussi des propriétés sur son temps d'exécution [4].

6 Conclusion et perspectives

Nous avons présenté dans ce papier une méthode pour implémenter, spécifier et vérifier des programmes OCaml fortement impératifs à l'aide de l'outil de preuve Why3. Par fortement impératifs, nous entendons des programmes qui effectuent des modification complexes du tas et introduisent par conséquent des alias arbitraires entre pointeurs. Ces programmes étant hors de portée du système de contrôle statique des alias de Why3, notre méthode s'appuie sur la construction d'un modèle mémoire spécifique à chaque exemple que nous voulons prouver. Nous avons développé un prototype qui automatise la construction de ce modèle mémoire.

Nous avons utilisé cette méthode avec succès sur de nombreux exemples non triviaux, dont certains sont présentés dans ce papier. Il reste néanmoins difficile en pratique d'utiliser et de raisonner sur un modèle mémoire explicite en Why3. D'une part, le nombre de conditions de vérification générées explose rapidement, ce qui peut rendre la preuve difficile à gérer. D'autre part, des preuves plus complexes, comme celles du tri fusion en place ou de l'algorithme de Schorr-Waite, exigent toujours un grand effort, soit en terme de temps de travail, soit en terme d'expertise. Un meilleur support de la part de Why3 est clairement souhaitable. Nous envisageons la construction d'une bibliothèque de logique de séparation en Why3 afin de rendre plus maniable la preuve de programmes qui manipulent le tas, à la manière de ce qui est fait dans l'outil KIV [16]. En pratique, nous avons déjà rencontré certains éléments classiques de la logique de séparation, tels que le prédicat `frame_mem` utilisé dans la preuve du tri fusion.

Références

[1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language, version 1.4*, 2009. http://frama-c.cea.fr/acsl.html.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's verify this with Why3. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(6) :709–727, 2015. See also http://toccata.lri.fr/gallery/fm2012comp.en.html.

[3] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7 :23–50, 1972.

[4] Arthur Charguéraud and François Pottier. Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits. *Journal of Automated Reasoning*, September 2017.

[5] Martin Clochard, Jean-Christophe Filliâtre, and Andrei Paskevich. How to avoid proving the absence of integer overflows. In Arie Gurfinkel and Sanjit A. Seshia, editors, *7th Working Conference on Verified Software : Theories, Tools and Experiments (VSTTE)*, volume 9593 of *Lecture Notes in Computer Science*, pages 94–109, San Francisco, California, USA, July 2015. Springer.

[6] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC : A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs)*, volume 5674 of *Lecture Notes in Computer Science*. Springer, 2009.

[7] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, number 7504 in Lecture Notes in Computer Science, pages 233–247. Springer, 2012.

[8] Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013. Springer.

[9] Jean-Christophe Filliâtre. Le petit guide du bouturage, ou comment réaliser des arbres mutables en OCaml. In *Vingt-cinquièmes Journées Francophones des Langages Applicatifs*, Fréjus, France, January 2014. https://www.lri.fr/~filliatr/publis/bouturage.pdf.

[10] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Research report, Université Paris Sud, 2016. https://hal.archives-ouvertes.fr/hal-01256434v3.

[11] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3) :152–174, 2016.

[12] Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods*, pages 15–29. Springer, 2004.

[13] Thierry Hubert and Claude Marché. A case study of C source code verification : the Schorr-Waite algorithm. 2005.

[14] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. VeriFast : A powerful, sound, predictable, fast verifier for C and Java. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[15] K. Rustan M. Leino. Dafny : An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[16] W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In William McCune, editor, *14th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 69–72, Townsville, North Queensland, Australia, july 1997. Springer.

[17] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. In *17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Comp. Soc. Press, 2002.

[18] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10 :501–506, 1967.

Articles courts

Définir le fini : deux formalisations d'espaces de dimension finie

Florian Faissole[1]

Inria, Université Paris-Saclay, F-91120 Palaiseau
LRI, CNRS & Université Paris-Sud, F-91405 Orsay
florian.faissole@inria.fr

Résumé

Les espaces de dimension finie permettent de décrire mathématiquement certaines méthodes numériques comme la méthode des éléments finis. Leur formalisation est nécessaire pour certifier que ces méthodes sont correctes et plus précisément qu'elles sont convergentes. Dans cet article, nous présentons deux méthodes pour formaliser les espaces de dimension finie en Coq, dans un cadre de topologie générale. Les deux approches utilisent des mécanismes différents et ne présentent pas les mêmes avantages et inconvénients. La première repose sur l'extension, à l'aide de structures canoniques, de la hiérarchie algébrique de la bibliothèque Coquelicot. Elle permet aisément de montrer que les espaces $\mathbb{R}^n$ sont de dimension finie et plus généralement que le produit cartésien d'espaces de dimension finie est de dimension finie. La seconde repose sur l'utilisation de sous-espaces en tant que prédicats sur l'espace total. Elle permet d'extraire des propriétés topologiques sur des sous-espaces de dimension finie d'un espace de dimension infinie, comme la fermeture des sous-espaces de dimension finie des espaces de Hilbert. Nous proposons par ailleurs une étude comparative de ces deux approches.

1 Introduction

Il y a un intérêt grandissant pour la formalisation de résultats d'analyse, qui servent de fondements à des applications critiques comme la résolution d'équations différentielles en médecine ou en aéronautique. Il y a plusieurs exemples de formalisations d'espaces de dimension finie dans des assistants de preuves comme HOL-Light [9], Isabelle/HOL [6] et Coq [3, 8] : dans la bibliothèque Mathematical Component, les espaces de dimension finie sont notamment utilisés dans la preuve du théorème de Feit-Thompson, un résultat important en théorie des groupes [8]. Les travaux de Harrison [9] et de Brunel [3] portent plus particulièrement sur les espaces euclidiens, $i.e.$ les espaces de Hilbert de dimension finie. Dans ces développements, les espaces de dimension finie sont des citoyens de première classe et nous ne pouvons que considérer des sous-espaces de dimension finie de ces espaces de dimension finie, dont ils héritent des propriétés topologiques. Nous pouvons néanmoins citer les travaux de Mahmoud, Aravantinos et Tahar [12] en Isabelle/HOL, qui définissent les espaces vectoriels de dimensions finie et infinie au même niveau, mais n'en extraient pas de propriétés topologiques.

Nous proposons deux formalisations d'espaces de dimension finie dans l'assistant de preuves Coq. Ces deux formalisations n'ont pas vocation à se substituer l'une à l'autre, mais sont complémentaires. En effet, alors que la première approche ($\mathcal{A}_1$) consiste à définir le type des espaces de dimension finie, la seconde ($\mathcal{A}_2$) permet de considérer les sous-espaces de dimension finie d'un module quelconque de dimension potentiellement infinie. Ainsi est-il possible de définir un module de dimension finie via l'approche $\mathcal{A}_1$ puis d'y considérer un sous-espace de dimension finie plus petite via l'approche $\mathcal{A}_2$. Conceptuellement, on pourrait imaginer ne pas faire de distinguo et construire ces deux espaces au même niveau, puis montrer que l'un est sous-espace de l'autre. Néanmoins, contrairement à d'autres systèmes comme PVS, il n'y a pas de mécanisme de sous-typage dans Coq. Définir une sous-structure est d'ailleurs connu pour être un problème difficile [10, 1].

Ce travail est basé sur la bibliothèque Coquelicot, une extension conservative de la bibliothèque réelle standard de Coq [2]. Cette bibliothèque permet de raisonner sur des structures algébriques très générales. Elle comporte en effet des structures à lois internes (`AbelianGroup`, `Ring`) ou à opérations externes (`ModuleSpace`) ainsi que des structures munies de propriétés topologiques (`UniformSpace`, `CompleteSpace`) ou des espaces munis d'une norme (`NormedModule`, `CompleteNormedModule`). Ces ensembles généraux sont construits grâce au mécanisme de structures canoniques [11] qui permet d'inférer et de hiérarchiser les structures algébriques, et de caractériser des espaces comme $\mathbb{R}$ en tant qu'instances de ces structures.

Dans Coquelicot, la topologie est définie via la notion de filtres (ensembles de voisinages enrichis de certaines propriétés). Ils permettent notamment de représenter des voisinages, comme ceux de la Figure 1.

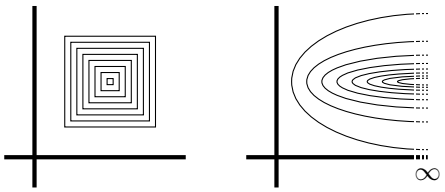


FIGURE 1 – Exemples de filtres, autour d'un point fini (gauche), à l'infini (droite).

Les filtres sont utilisés pour définir l'ensemble des notions topologiques, comme la fermeture, la complétude ou les notions de convergence. Un filtre propre (`ProperFilter`) est un filtre ne contenant pas d'ensembles vides. Un filtre de Cauchy (`cauchy`) est un filtre contenant des boules de rayon arbitrairement petit (généralisation des suites de Cauchy). Un espace complet `T` est muni d'une fonction limite `lim : ((T -> Prop) -> Prop) -> T` prenant un filtre en entrée et retournant un élément de `T` défini comme la limite de ce filtre. Elle vérifie la propriété de complétude (convergence de tout filtre propre de Cauchy) :

```
forall F : (E -> Prop) -> Prop, ProperFilter F -> cauchy F ->
            forall eps : posreal, F (ball (lim F) eps)
```

D'autres notions topologiques propres à nos preuves sont définies dans la Section 3, comme la notion de fermeture et celle de filtre implicite sur un sous-ensemble.

2 Structures canoniques et espaces de dimension finie

Il est possible, en utilisant les structures canoniques, d'étendre la hiérarchie algébrique de Coquelicot. Par exemple, Boldo, Clément, Faissole, Martin et Mayero étendent la hiérarchie algébrique de Coquelicot aux espaces préhilbertiens et hilbertiens [1]. À notre tour, nous proposons de l'enrichir de la structure de $\mathbb{R}$-module de dimension finie [1] ((`sum_n f n`) est la somme des $f(i)$ pour $0 \leq i \leq n$ et `scal` est la multiplication par un scalaire) :

```
Record mixin_of (E : ModuleSpace R_Ring) := Mixin {
          dim : nat ;
          B : nat -> E ;
          B0 : B 0 = zero ;
          col : forall u : E, exists L : nat -> R,
                    u = sum_n (fun n => scal (L n) (B n)) dim }.
```

1. On pourrait généraliser cette construction sur tout $\mathbb{A}$-module sur un anneau $\mathbb{A}$.

Un $\mathbb{R}$-module E de dimension finie est muni d'un entier `dim` et d'une "famille génératrice" `B : nat -> E`. Par ailleurs, `B(0)` est défini comme le vecteur nul afin de caractériser l'espace de dimension 0. La propriété `col` assure que tout vecteur du module de dimension finie est combinaison linéaire des `dim` premiers vecteurs de `B`. Ce choix induit deux astuces facilitant les preuves. Premièrement, la suite `B` peut être définie arbitrairement au-delà du rang `dim`. On peut donc construire `B` comme une suite de vecteurs de E sans avoir à s'assurer que les termes de rang supérieur à `dim` sont nuls. Par ailleurs, `dim` est une surestimation de la dimension du module de dimension finie. En effet, nous nous autorisons à avoir `B(i) = 0` pour `0 ≤ i ≤ dim`. Par exemple, nous montrons que $\mathbb{R}$ est un module de dimension finie de dimension 1 et de famille génératrice $B^{\mathbb{R}}$ telle que $B_0^{\mathbb{R}} = 0$ et $\forall n \in \mathbb{N}^*, B_n^{\mathbb{R}} = 1$.

Afin d'éprouver notre choix de formalisation, nous définissons le produit cartésien de modules de dimension finie. Nous considérons E_1 et E_2 des $\mathbb{R}$-modules de dimensions finies respectives dim_1 et dim_2 et de familles génératrices respectives B^{E_1} et B^{E_2}. Nous définissons une famille génératrice produit cartésien $B^{E_1 \times E_2}$ de la façon suivante :

$$\begin{cases} B_0^{E_1 \times E_2} = (0,0) \\ \forall n \in \mathbb{N}^*, n \leq dim_1 \Rightarrow B_n^{E_1 \times E_2} = \left(B_n^{E_1}, 0\right) \\ \forall n \in \mathbb{N}^*, dim_1 < n \leq dim_1 + dim_2 \Rightarrow B_n^{E_1 \times E_2} = \left(0, B_{n-dim_1}^{E_2}\right) \\ \forall n \in \mathbb{N}^*, n > dim_1 + dim_2 \Rightarrow B_n^{E_1 \times E_2} = (0,0) \end{cases}$$

Nous prouvons ensuite que le produit cartésien de modules de dimension finie est un module de dimension finie. En Coq, bien que la preuve ne soit pas technique, elle peut être fastidieuse et constitue quelques centaines de lignes de preuve.

Lemme 1. *Soit E_1 et E_2 des $\mathbb{R}$-modules de dimensions finies respectives dim_1 et dim_2 et de familles génératrices respectives B^{E_1} et B^{E_2}. Alors $E_1 \times E_2$ est un $\mathbb{R}$-module de dimension finie $dim_1 + dim_2$ et de famille génératrice $B^{E_1 \times E_2}$.*

Démonstration. Par définition d'un module de dimension finie :

$$\forall u \in E_1, \exists \lambda^{E_1} : nat \to \mathbb{R}, u = \sum_{i=0}^{dim_1} \lambda_i^{E_1} B_i^{E_1}$$
$$\forall u \in E_2, \exists \lambda^{E_2} : nat \to \mathbb{R}, u = \sum_{i=0}^{dim_2} \lambda_i^{E_2} B_i^{E_2}$$

Pour (u_1, u_2) donné, nous définissons la suite $\lambda^{E_1 \times E_2}$ de la façon suivante :

$$\begin{cases} \forall n, n \leq dim_1 \Rightarrow \lambda_n^{E_1 \times E_2} = \lambda_n^{E_1} \\ \forall n, n > dim_1 \Rightarrow \lambda_n^{E_1 \times E_2} = \lambda_{n-dim_1}^{E_2} \end{cases}$$

Donc :

$$\begin{aligned} (u_1, u_2) &= \left(\sum_{i=0}^{dim_1} \lambda_i^{E_1} B_i^{E_1}, \sum_{i=0}^{dim_2} \lambda_i^{E_2} B_i^{E_2}\right) = \left(\sum_{i=1}^{dim_1} \lambda_i^{E_1} B_i^{E_1}, \sum_{i=1}^{dim_2} \lambda_i^{E_2} B_i^{E_2}\right) \\ &= \left(\sum_{i=1}^{dim_1} \lambda_i^{E_1} B_i^{E_1}, \sum_{i=1}^{dim_1+dim_2} \lambda_{i-dim_1}^{E_2} B_{i-dim_1}^{E_2}\right) \\ &= \sum_{i=1}^{dim_1} \lambda_i^{E_1 \times E_2} B_i^{E_1 \times E_2} + \sum_{i=dim_1+1}^{dim_1+dim_2} \lambda_i^{E_1 \times E_2} B_i^{E_1 \times E_2} \\ &= \sum_{i=0}^{dim_1+dim_2} \lambda_i^{E_1 \times E_2} B_i^{E_1 \times E_2} \end{aligned}$$

Donc, comme de plus $B_0^{E_1 \times E_2} = (0,0)$, $E_1 \times E_2$ est un $\mathbb{R}$-module de dimension finie $dim_1 + dim_2$ et de famille génératrice $B^{E_1 \times E_2}$. □

Ainsi, nous pouvons en déduire que pour tout $n \in \mathbb{N}$, $\mathbb{R}^n$ est un $\mathbb{R}$-module de dimension finie. Sur papier, un résultat comme celui-ci peut paraître relativement simple, mais sa formalisation induit des écueils. D'une part, il est nécessaire d'inférer la bonne famille génératrice produit B, qui ne peut

pas être caractérisée par une unique expression qui dépend du rang n, mais varie suivant que n soit ou non supérieur à la dimension de E_1. Par ailleurs, il faut trouver les coefficients λ_i de la combinaison linéaire de vecteurs de B, ce qui mène à des raisonnements calculatoires. Souvent ellipsés par les mathématiciens, ces raisonnements sont difficilement automatisables en Coq et doivent être minutieusement conduits par l'utilisateur.

3 Sous-espaces de dimension finie

En analyse fonctionnelle, le théorème de Lax–Milgram établit l'existence d'une unique solution de la formulation faible d'une équation aux dérivées partielles sur tout sous-module fermé d'un espace de Hilbert. Ce théorème peut être appliqué à des sous-modules de dimension finie d'espaces de Hilbert (un espace de Hilbert est un module muni d'un produit scalaire et qui est complet).

La formalisation utilisant les structures canoniques (voir Section 2) n'est pas commode pour définir un espace de dimension finie comme sous-espace d'un espace E de dimension potentiellement infinie. Dans cette section, les sous-espaces de dimension finie sont des prédicats de type $E \to Prop$.

3.1 Définitions

On suppose que $E : Hilbert$. On définit les sous-espaces de dimension finie comme prédicats munis d'une propriété proche de `FDIM`, définie pour les espaces dans la Section 2.

```
Variables (E:Hilbert)(n:nat)(B:nat→E).
Definition FDIM (phi:E→Prop) :=
    match (eq_nat_dec n 0) with
     |left _⇒∀u, phi u↔u=zero (* n=0 *)
     |right _⇒∀u, phi u↔ (* n>0 *)
       ∃L:nat→R, u = sum_n (fun i⇒scal (L i) (B i)) (n-1) end.
```

On aurait pu choisir, sans incidence sur la suite, de définir `FDIM` en postulant l'existence de `n` et `B`.

3.2 Fermeture des sous-espaces de dimension finie des espaces de Hilbert

Le théorème de Lax–Milgram a été prouvé formellement dans l'assistant de preuves Coq [1]. Ce théorème s'applique aux sous-modules fermés des espaces de Hilbert. Afin de vérifier la convergence de la méthode des éléments finis, nous souhaitons appliquer le théorème sur un espace de Hilbert entier (volume irrégulier) et sur un de ses sous-espaces de dimension finie (maillage). L'espace de Hilbert entier est trivialement un sous-module fermé de lui-même et le théorème peut s'y appliquer. Il reste à prouver que tout sous-espace de dimension finie d'un espace de Hilbert est fermé. La preuve formelle de ce résultat est disponible en ligne [2] et est décrite de façon détaillée par Faissole [7] (nous ne donnons ici qu'une intuition des preuves). Sans types dépendants, il n'est pas possible de dire qu'un filtre est un filtre sur un sous-espace φ de E, car il est de type $(E \to Prop) \to Prop$ (donc c'est un filtre sur E). Néanmoins, on peut donner une relation entre un filtre de E et le sous-espace φ :

Définition 1. *Soit $E : Type$. Soit $\mathcal{F} : (E \to Prop) \to Prop$ un filtre sur E.*
Soit $\varphi : E \to Prop$ un sous-espace de E. $\mathcal{F}$ est un filtre implicite sur φ si :

$$\forall \psi : E \to Prop, \mathcal{F}(\psi) \Rightarrow \exists x \in E, \psi(x) \wedge \varphi(x).$$

Il s'agit d'un filtre dont tous les éléments ont une intersection non vide avec φ.

2. https://github.com/FFaissole/FDIM_Topology

Définition 2. *Soit $E : CompleteSpace$, $\varphi : E \to Prop$. Le sous-espace φ est fermé dans E lorsque pour tout filtre $\mathcal{F} : (E \to Prop) \to Prop$, si $\mathcal{F}$ est propre, de Cauchy et implicite sur φ alors $lim(\mathcal{F}) \in \varphi$.*

Notre définition de filtre implicite sur un sous-ensemble est proche de celle de filtre voisinage de Cohen et Rouhling [5], qui, contrairement à notre construction, repose sur `ssreflect`. De même, leur notion de fermeture est très similaire, et nous pensons qu'elles sont équivalentes en logique classique [3]. Le sous-module engendré par $u \in E$ (noté `span u`) est le sous-ensemble des vecteurs de E colinéaires à u. Tout vecteur d'un sous-espace de dimension finie est en fait somme de vecteurs des sous-modules engendrés par les vecteurs de la famille génératrice B.

```
Definition span (u : E) := fun x:E ⇒ (∃ (l : R), x = scal l u).
```

On prouve que le sous-module engendré par un élément d'un espace de Hilbert est fermé :

Lemme 2. *On suppose que $E : Hilbert$, $u \in E$. Alors $span(u)$ est fermé dans E.*

Démonstration. Dans la preuve papier standard, il s'agirait de considérer une suite de Cauchy de la forme $(\lambda_n u)_{n\in\mathbb{N}}$ (dans $span(u)$) et de prouver qu'elle converge dans $span(u)$. On extrairait la suite $(\lambda_n)_{n\in\mathbb{N}}$ (également de Cauchy) et on prouverait qu'elle converge vers une limite ℓ car $\mathbb{R}$ est complet. On en déduirait que $(\lambda_n u)_{n\in\mathbb{N}}$ converge vers ℓu. Comme nous travaillons avec des filtres, on considère $\mathcal{F}$ un filtre propre, de Cauchy et implicite sur $span(u)$. On doit construire un transformeur de filtre pour obtenir le filtre sur $\mathbb{R}$ équivalent à la suite $(\lambda_n)_{n\in\mathbb{N}}$, prouver qu'il est de Cauchy, puis qu'il converge, et enfin en déduire que $\mathcal{F}$ converge. □

Cette preuve est difficile car elle nécessite la traduction de la preuve papier dans le cadre des filtres. Les transformeurs de filtres peuvent notamment être difficiles à définir. Afin de faciliter les preuves, nous considérons ces transformeurs comme des fonctions totales mais les utilisons toujours sur des filtres que nous savons être *implicites* sur l'ensemble adéquat (ce sans quoi cela ne présente pas d'intérêt).

Théorème 1. *Soit $E : Hilbert$ et $\varphi : E \to Prop$. On suppose que φ est de dimension finie n et de famille génératrice orthonormée B. Alors φ est fermé.*

Démonstration. Si $u \in E : Hilbert$, $span(u)$ est un sous-espace fermé de E. On peut en déduire que la somme directe d'un sous-module fermé de E et de $span(u)$ est également fermé (1). Or, on montre qu'un sous-module de dimension finie n est somme directe d'un autre sous-module de dimension finie $n-1$ et de $span(B_n)$ (2). On raisonne par induction en supposant que tout sous-module de dimension $n-1$ est fermé. Par (1) et (2), tout sous-module de dimension n est fermé. □

4 Conclusion et perspectives

Nous proposons deux niveaux de formalisation d'espaces de dimension finie en Coq, tous deux fondés sur la bibliothèque Coquelicot. Le premier d'entre eux place ces espaces au même plan que les autres structures algébriques de Coquelicot. Son avantage est la possibilité d'instancier des espaces usuels comme $\mathbb{R}$ ou $\mathbb{R}^n$ comme cas particuliers de modules de dimension finie. La deuxième approche considère tout espace de dimension finie comme sous-espace d'un espace plus grand, de dimension potentiellement infinie. Nous proposons une preuve formelle de la fermeture des sous-espaces de dimension finie des espaces de Hilbert (utilisant des filtres pour la topologie). Notre développement comporte environ 2500 lignes de Coq, dont 1500 pour la preuve de fermeture. Les preuves papier de ce

3. Cela pose la question plus générale d'interopérabilité entre bibliothèques lorsque des notions équivalentes ou proches sont définies.

résultat [4] excèdent rarement 3 pages. La longueur de notre preuve est due à la construction de transformeurs de filtres (les preuves papier utilisent des transformeurs de suites). La deuxième technique de formalisation est retenue pour servir à l'application du théorème de Lax–Milgram sur les sous-espaces de dimension finie d'espaces de Hilbert. Ces résultats doivent servir à la vérification de la méthode des éléments finis (convergence notamment), mais nous devons formaliser des espaces particuliers comme $L^2(\Omega)$ et $H^1(\Omega)$, dont il faut prouver le caractère hilbertien et extraire un maillage de dimension finie. La vérification de la méthode pourra servir de base à la preuve de programmes l'implémentant, comme la bibliothèque C++ FELiScE [4].

Remerciements

Nous remercions Sylvie Boldo, François Clément, Vincent Martin et Micaela Mayero pour leur aide dans la formalisation des résultats. Merci aux reviewers anonymes pour leurs remarques.

Références

[1] S. Boldo, F. Clément, F. Faissole, V. Martin, and M. Mayero. A Coq Formal Proof of the Lax–Milgram theorem. In *6th ACM SIGPLAN Conference on Certified Programs and Proofs*, pages 79–89, Paris, France, January 2017.

[2] S. Boldo, C. Lelay, and G. Melquiond. Coquelicot : A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1) :41–62, 2015.

[3] A. Brunel. Non-constructive complex analysis in Coq. In *18th International Workshop on Types for Proofs and Programs, TYPES 2011, September 8-11, 2011, Bergen, Norway*, pages 1–15, 2011.

[4] F. Clément and V. Martin. The Lax-Milgram Theorem. A detailed proof to be formalized in Coq. Research Report RR-8934, Inria Paris, July 2016.

[5] Cyril Cohen and Damien Rouhling. A Formal Proof in Coq of LaSalle's Invariance Principle. In *Interactive Theorem Proving*, Brasilia, Brazil, September 2017.

[6] J. Divasón Mallagaray. *Formalisation and execution of Linear Algebra : theorems and algorithms*. PhD thesis, Universidad de La Rioja, 2016.

[7] F. Faissole. Formalization and closedness of finite dimensional subspaces. In *19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, SYNASC 2017, September 2017.

[8] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 163–179, 2013.

[9] J. Harrison. A HOL theory of Euclidean space. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005*, volume 3603 of *Lecture Notes in Computer Science*, Oxford, UK, 2005. Springer-Verlag.

[10] A. Mahboubi. The Rooster and the Butterflies. In Jacques Carette, David Aspinal, Christopher Lange, Petr Sojka, and Wolfgang Windsteiger, editors, *CICM 2013 - Conference on Intelligent Computer Mathematics - 2013*, volume 7961 of *Lecture Notes in Artificial Intelligence*, pages 1–18, Bath, United Kingdom, July 2013. Springer.

[11] A. Mahboubi and E. Tassi. Canonical structures for the working Coq user. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 19–34, 2013.

[12] M. Y. Mahmoud, V. Aravantinos, and S. Tahar. Formalization of infinite dimension linear spaces with application to quantum theory. In *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*, pages 413–427, 2013.

4. Finite Elements for Life Sciences and Engineering : https://gforge.inria.fr/projects/felisce/

Génération aléatoire de programmes guidée par la vivacité

Gergö Barany[1] et Gabriel Scherer[2]

[1] Inria Paris
gergo.barany@inria.fr
[2] Inria Saclay
gabriel.scherer@gmail.com

Résumé

Les programmes générés aléatoirement sont un bon moyen de tester des compilateurs et des outils d'analyse de logiciel. Des centaines de bogues ont été trouvés dans des compilateurs C très utilisés (GCC, Clang) par des tests aléatoires. Pourtant, les générateurs existants peuvent générer beaucoup de code mort (dont les résultats ne sont jamais utilisés). Compiler un tel programme laisse relativement peu de possibilités d'exercer les optimisations complexes du compilateur.

Pour résoudre ce problème, nous proposons la génération aléatoire de programmes guidée par la vivacité. Dans cette approche, le programme aléatoire est construit *bottom-up*, en combinaison avec une analyse de flot de données structurelle pour assurer que le système ne génère jamais de code mort.

L'algorithme est implémenté dans un greffon pour l'outil Frama-C. Nous l'évaluons en comparaison avec Csmith, le générateur aléatoire standard pour le langage C. Les programmes générés par notre outil compilent vers une plus grande quantité de code machine, avec une plus grande variété d'instructions.

Ce papier est une version courte d'un article présenté à LOPSTR 2017.

1 Motivation

Les compilateurs pour les langages de programmation modernes sont compliqués et difficiles à comprendre. Malgré les avances en vérification de compilateurs [Ler09, TMK+16], la plupart des compilateurs réalistes ne sont pas vérifiés formellement. Il faut des tests pour acquérir une confiance en leur correction partielle. Une approche populaire est le test aléatoire, fait avec des fichiers d'entrée produits par un générateur aléatoire de programmes. L'outil le plus connu dans ce domaine est Csmith, un générateur aléatoire de programmes C. Csmith est très puissant : il a trouvé des centaines de bogues dans des compilateurs très utilisés comme GCC et Clang, et même quelques bogues dans des parties non vérifiées du compilateur vérifié CompCert [YCER11].

Le travail présenté ici provient d'un projet de test aléatoire des optimisations effectuées par des compilateurs C. Nous avions besoin d'un générateur aléatoire produisant de grandes fonctions effectuant des calculs compliqués, sans appels de fonctions, pour pouvoir analyser le code machine optimisé généré par des compilateurs différents. Csmith peut être configuré pour cet usage, mais nous avons découvert que le compilateur enlève presque tous les calculs présents dans le code source généré par Csmith. Même des boucles imbriquées entières disparaissent souvent dans le code machine compilé à partir du code source généré par Csmith. Ceci nous empêche de faire des analyses intéressantes sur le code machine final.

Le problème vient du fait que Csmith génère des programmes qui ne contiennent presque que du *code mort* : des calculs dont les résultats ne sont jamais utilisés. Les compilateurs enlèvent tout le code mort, laissant très peu de possibilités d'appliquer nos analyses.

Cet article décrit `ldrgen`, un nouveau générateur de code C guidé par une analyse de vivacité pour éviter de générer du code mort.

2 Analyse de la vivacité

Une variable est *vivante* à une position donnée d'un programme si sa valeur à cette position peut être utilisée plus tard. Sinon, elle est *morte*. De la même façon, une affectation $v = e$ est vivante si elle définit une variable vivante, et morte sinon. Par exemple, dans un morceau de code comme `x = y + z; return y`, la variable `x` est morte juste après son affectation car sa valeur n'est jamais utilisée. L'affectation elle-même est donc morte aussi, et elle peut être enlevée du programme sans en changer le sens.

Notre objectif est de ne générer aucun code mort, et en particulier, de ne générer que des programmes dans lesquels toutes les affectations sont vivantes.

2.1 Analyse itérative

L'analyse de la vivacité est une analyse classique de flot de données [NNH99]. Elle parcourt le graphe de flot de contrôle du programme en arrière et itère jusqu'à trouver un point fixe. Chaque instruction S est liée à deux ensembles de variables vivantes : l'ensemble $S^\bullet$ de variables vivantes juste avant S, et l'ensemble S° de celles vivantes juste après S. Ces ensembles sont liés par des fonctions f_S spécifiques à l'instruction S qui décrivent la propagation de l'information. Pour une affectation $v = e$, à la variable v, de la valeur d'une expression e contenant l'ensemble de variables $FV(e)$, la propagation est faite par l'équation suivante :

$$S^\bullet = f_{v=e} = (S^\circ \setminus \{v\}) \cup FV(e)$$

Pour une instruction de branchement avec une condition c (comme $S =$ `if` (c) $\ldots$), l'analyse unit l'information de tous les successeurs dans le graphe de flot de contrôle et rajoute les variables de la condition :

$$S^\bullet = \bigcup_{S_i \in succ(S)} S_i^\bullet \cup FV(c)$$

Les boucles génèrent des systèmes d'équations récursifs. Ces systèmes sont traditionnellement résolus par itération jusqu'à obtenir un point fixe minimal comme solution.

2.2 Analyse structurelle de programmes sans code mort

L'analyse de flot de données sur le graphe de flot de contrôle est pratique pour identifier le code mort dans un programme donné. Cependant, notre objectif n'est pas d'analyser des programmes donnés mais d'utiliser l'analyse pour guider la génération de code afin d'éviter de générer du code mort. Nous voudrions éviter de construire un graphe de flot de contrôle et de calculer les points fixes pour les boucles par itération.

Nous avons donc décidé d'essayer une approche structurelle basée sur la structure de l'arbre de syntaxe abstraite. Les approches structurelles ne sont appropriées qu'aux programmes structurés (sans `goto` etc.), mais cela suffit pour notre générateur.

L'analyse est présentée dans la figure 1 comme un système de règles d'inférence. Les informations inférées sont des triplets $\langle S^\bullet \rangle \ S \ \langle S^\circ \rangle$ d'une instruction S et deux ensembles de variables vivantes avant et après l'instruction $(S^\bullet, S^\circ)$. Les règles contiennent deux sortes d'hypothèses : L'une sont les équations de propagation de l'information de l'analyse de flot de données. En particulier, la règle Assign pour les affectations de la forme $v = e$ a comme hypothèse l'équation $S^\bullet = (S^\circ \setminus \{v\}) \cup FV(e)$, comme dans le cas de l'analyse de flot de données itérative.

La deuxième sorte d'hypothèses sont les conditions pour assurer que les programmes qui contiennent du code mort ne sont pas acceptés par le système. La condition la plus importante

$$\textsc{Return}\ \frac{}{\langle \{v\} \rangle\ \texttt{return}\ v\ \langle \emptyset \rangle} \qquad \textsc{Skip}\ \frac{}{\langle L \rangle\ \{\}\ \langle L \rangle}$$

$$\textsc{Assign}\ \frac{v \in S^\circ \qquad S^\bullet = (S^\circ \setminus \{v\}) \cup FV(e)}{\langle S^\bullet \rangle\ v = e\ \langle S^\circ \rangle}$$

$$\textsc{Sequence}\ \frac{\langle S_1^\bullet \rangle\ S_1\ \langle S_2^\bullet \rangle \qquad \langle S_2^\bullet \rangle\ S_2\ \langle S_2^\circ \rangle \qquad S_2^\bullet \neq \emptyset}{\langle S_1^\bullet \rangle\ S_1\ ;\ S_2\ \langle S_2^\circ \rangle}$$

$$\textsc{If}\ \frac{\langle S_1^\bullet \rangle\ S_1\ \langle S^\circ \rangle \qquad \langle S_2^\bullet \rangle\ S_2\ \langle S^\circ \rangle \qquad S^\bullet = S_1^\bullet \cup S_2^\bullet \cup FV(c) \qquad S_1 \neq \{\} \vee S_2 \neq \{\}}{\langle S^\bullet \rangle\ \texttt{if}\ (c)\ S_1\ \texttt{else}\ S_2\ \langle S^\circ \rangle}$$

$$\textsc{While}\ \frac{\langle B^\bullet \rangle\ B\ \langle B^\circ \rangle \qquad B^\circ = S^\bullet\ \text{(minimal)} \qquad S^\bullet = S^\circ \cup B^\bullet \cup FV(c) \qquad S^\circ \neq \emptyset}{\langle S^\bullet \rangle\ \texttt{while}\ (c)\ B\ \langle S^\circ \rangle}$$

FIGURE 1 – Système de règles d'inférence pour la reconnaissance de programmes sans code mort

est celle de la règle ASSIGN : Une affectation d'une variable v ne peut être acceptée que si v est vivante après l'affectation. De la même façon, la condition $S_1 \neq \{\} \vee S_2 \neq \{\}$ pour IF assure que les instructions de branchement inutiles comme `if (x) {} else {}` ne sont pas acceptées.

Le traitement des boucles est plus compliqué à cause de la dépendance cyclique entre l'ensemble $B^\bullet$ des variables vivantes au début du corps B de la boucle, et l'ensemble B° des variables vivantes à la fin du corps de la boucle. La condition de minimalité exprime que la solution désirée est un point fixe minimal du système d'équations. Un point fixe unique minimal existe toujours [NNH99].

Un programme S ne contient pas de code mort si le triplet $\langle S^\bullet \rangle\ S\ \langle \emptyset \rangle$ peut être dérivé dans le système de règles d'inférence. Il s'agit ici d'une approximation syntaxique : Les optimisations sémantiques du compilateur peuvent quand-même trouver des simplifications qui peuvent rendre inutile certaines parties du code.

3 Génération de code non mort

Les règles d'inférence peuvent être interprétées comme un générateur exécutable aléatoire (ou exhaustif) de programmes sans code mort. Comme l'analyse traditionnelle de vivacité, la génération se fait en arrière, c'est-à-dire dans la direction opposée au flot de contrôle.

Le générateur commence par générer une variable v et une instruction `return` v. L'ensemble de variables vivantes avant cette instruction est $L = \{v\}$. Le générateur applique des fonctions pour générer de nouvelles instructions aléatoires étant donné un ensemble de variables vivantes. Les nouvelles instructions S sont préfixées au programme, et l'ensemble L est mis à jour selon la fonction correspondante f_S. Cet ensemble guide le générateur : en particulier, pour générer une affectation d'une variable v, L doit contenir v à ce point dans le programme. L'itération se termine après un nombre prédéfini d'instructions, ou si jamais l'ensemble L devient vide. La figure 2 contient le pseudo-code du générateur dans un langage fonctionnel.

Lors de la génération des boucles, nous voudrions éviter la construction d'un graphe de

```
let random_statements L code =
    if L = ∅ then (code, L)
    else
        let (S, L′) = random_statement L in
        random_statements L′ (S :: code)

let random_statement L =
    let statement_generator = random_select [assignment; branch; loop] in
    statement_generator L

let assignment L =
    let v = random_select L in
    let e = random_expression () in
    ("v = e", (L \ {v}) ∪ FV(e))

let branch L =
    let (t, L1) = random_statements L [] in
    let (f, L2) = random_statements L [] in
    let c = random_expression () in
    ("if (c) t else f", L1 ∪ L2 ∪ FV(c))

let loop L =
    (* Générer un point fixe pour la boucle, puis la boucle elle−même. *)
    let c = random_expression () in
    let B′ = random_variable_set () in
    let (code, L') = random_statements (L ∪ B′ ∪ FV(c)) [] in
    let V = {b ∈ B′ | b ∉ L′ ∨ b n'a pas d'occurrences dans code} in
    if V = ∅ then
        ("while (c) code", L′ ∪ L)
    else
        let e = random_expression_on_variables V in
        let v = random_select L′ in
        let code' = "v = e" :: code in
        ("while (c) code′", (L′ \ {v}) ∪ V ∪ L)

(* Commencer la génération à la fin du programme. *)
let v = random_variable ()
let (code, L) = random_statements {v} ["return v"]
```

FIGURE 2 – Pseudo-code du générateur aléatoire guidé par la vivacité.

flot de contrôle et l'itération jusqu'à un point fixe de la boucle. En effet, nous ne pouvons pas facilement faire une analyse itérative d'un corps de boucle qui n'existe pas encore.

Pour éviter ce problème cyclique, notre générateur choisit d'abord un ensemble $B^\bullet$ de variables qui doivent devenir vivantes au début du corps de la boucle, puis génère le corps lui-même. Finalement, il peut générer des instructions supplémentaires pour assurer la validité du choix de variables.

Plus précisément, nous cherchons à générer une liste B d'instructions et un ensemble $B^\bullet$ tels que $B^\bullet = f_B(B^\circ)$ et $B^\circ = S^\circ \cup B^\bullet \cup FV(c)$, étant donnés S° et une expression c pour la condition de la boucle. Nous générons un ensemble B' de nouvelles variables pour représenter les variables intéressantes pour le cas des boucles : les variables utilisées dans la boucle avec une valeur qui peut venir d'une itération précédente de la même boucle. Dit autrement, ces variables peuvent être affectées par le corps de la boucle, mais elles doivent aussi être utilisées dans le corps de la boucle.

Après avoir choisi l'ensemble B', nous pouvons générer une liste d'instructions sous l'hypothèse que les variables dans l'ensemble $S^\circ \cup B' \cup FV(c)$ sont vivantes à sa fin. Le générateur rend cette liste d'instructions (appelée *code* en figure 2) et l'ensemble L' de variables vivantes à son début. Pour que *code* puisse servir comme corps de boucle, il suffit d'assurer que notre choix de B' satisfait la condition ci-dessus : chaque variable $b \in B'$ doit être utilisée, et elle doit être vivante au début de la boucle. Il suffit de préfixer *code* avec une instruction qui utilise toutes les variables b ne satisfaisant pas encore cette condition pour obtenir une liste finale B et un ensemble $B^\bullet$ qui satisfont $B^\bullet = f_B(B^\circ)$, $B^\circ = S^\circ \cup B' \cup FV(c)$, et $B' \subseteq B^\bullet$.

Par exemple, imaginons vouloir générer une boucle `while` étant donné un ensemble $S^\circ = \{\mathtt{x}\}$ de variables vivantes après la boucle. Nous générons une condition aléatoire $c = \mathtt{y} < 10$ et un ensemble aléatoire $B' = \{\mathtt{z}\}$ de nouvelles variables à utiliser dans la boucle. Cela nous donne l'instance suivante de la règle While :

$$\frac{\langle B^\bullet \rangle \ B \ \langle B^\circ \rangle \qquad B^\circ = S^\bullet \qquad S^\bullet = \{\mathtt{x}\} \cup B^\bullet \cup \{\mathtt{y}\} \qquad B' = \{\mathtt{z}\} \subseteq B^\bullet}{\langle S^\bullet \rangle \ \mathtt{while}\ (\mathtt{y} < 10)\ B \ \langle \{\mathtt{x}\} \rangle}$$

Il reste à générer le code B pour le corps de la boucle sous l'hypothèse $B^\circ = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$. Si le générateur renvoie le code `x = z + x; y = y + 1;`, la variable `z` est vivante à son début, et toutes les conditions sont satisfaites. Nous avons donc construit une dérivation du triplet

$$\langle \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \rangle \ \mathtt{while}\ (\mathtt{y} < 10)\ \{\mathtt{x} = \mathtt{z} + \mathtt{x}; \mathtt{y} = \mathtt{y} + 1;\} \ \langle \{\mathtt{x}\} \rangle \, .$$

En revanche, si `z` n'est pas vivante au début du code généré pour le corps de la boucle, il suffit de préfixer le code avec une affectation utilisant `z` en lecture.

4 Implémentation

Notre générateur `ldrgen` est implémenté dans un greffon pour Frama-C, qui est un outil extensible pour l'analyse et transformation de logiciels écrits en C [KKP+15]. `ldrgen` est un logiciel libre, disponible à https://github.com/gergo-/ldrgen. L'implémentation du générateur comprend environ 600 lignes d'OCaml, suivant la structure du pseudo-code en figure 2. Pour l'instant, l'outil génère un sous-ensemble du langage C avec des opérations arithmétiques et bit à bit sur tous les types de base, ainsi que des branchements avec `if`, des boucles `while`, et des boucles `for` d'une forme restreinte pour calculer une opération de réduction sur des tableaux de taille fixe. Il ne génère pas encore de `struct` ni d'arithmétique de pointeurs.

Une description plus complète se trouve dans la version originale de cet article [Bar17].

Table 1 – Comparaison du code généré par Csmith et `ldrgen` en 1000 appels de chacun.

	générateur	min	médiane	max	total
lignes de code	Csmith	25	368.5	2953	459021
	`ldrgen`	12	411.5	1003	389939
instructions	Csmith	1	15.0	1006	45606
	`ldrgen`	1	952.5	4420	1063503
opcodes uniques	Csmith	1	8	74	146
	`ldrgen`	1	95	124	204

5 Évaluation

Nous évaluons le code généré par `ldrgen` en comparaison avec Csmith. Pour ces expériences, nous avons configuré Csmith pour générer une seule fonction sans appels à d'autres fonctions. (Le mode standard de Csmith est de générer une application complète et de cacher la plupart des opérations arithmétiques dans des fonctions auxiliaires qui protègent des débordements.)

Nos résultats pour le code généré par 1000 appels de chaque outil sont résumés dans la table 1. La première partie de la table montre que nous avons choisi des options de configuration pour les deux outils pour générer des quantités comparables de code source. Ceci nous permet de faire une comparaison juste des programmes générés.

La deuxième partie de la table montre le nombre d'instructions machine émises par GCC (`-O3`, sur x86-64) pour le code source généré par chaque outil. La médiane de 15 instructions machine pour Csmith nous démontre qu'au moins la moitié des fonctions générés par Csmith ont une taille triviale. Ces fonctions très petites sont exactement ce que nous voulions éviter car elles ne sont pas intéressantes pour notre analyse de code machine. En moyenne, le code généré par Csmith compile vers environ une seule instruction machine par 10 lignes de code source. En revanche, avec `ldrgen` nous obtenons environ 2,5 instructions machine par ligne de code.

La dernière partie de la table concerne la variété d'instructions générées. Nous regardons les nombres d'instructions machine différentes par fonction générée et au total. Même la fonction la plus diverse générée par Csmith contient moins de variété que la fonction médiane générée par `ldrgen`. Au total, le code généré par `ldrgen` permet une couverture du jeu d'instructions 40 % plus élevée par rapport au code généré par Csmith. Par inspection des ensembles d'instructions uniques, nous avons trouvé que presque toute la différence vient des instructions vectorielles (SIMD) qui sont émises pour les boucles `for` sur des tableaux générées par `ldrgen`. Csmith sait générer des boucles similaires, mais leurs résultats ne sont presque jamais utilisés.

6 Conclusions

Nous avons présenté `ldrgen`, un générateur de code source C aléatoire. Le but de ce générateur est de ne jamais générer du code mort, afin d'obtenir des grandes quantités d'instructions machine variées générés par des compilateurs optimisants. `ldrgen` est guidé par une analyse structurelle de vivacité lors de la génération de code. Une nouvelle approche structurelle de l'analyse de flot de données, interprétée comme système d'inférence, nous permet de faire l'analyse sans devoir implémenter l'itération jusqu'à un point fixe. Par rapport au générateur de code Csmith, `ldrgen` génère une plus grande quantité de code machine plus varié.

Références

[Bar17] Gergö Barany. Liveness-driven random program generation. In *27th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2017)*, 2017.

[KKP+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : A software analysis perspective. *Formal Aspects of Comp.*, 27(3) :573–609, 2015.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7) :107–115, July 2009.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[TMK+16] Yong Kiam Tan, Magnus O. Myreen, Ramana Kumar, Anthony Fox, Scott Owens, and Michael Norrish. A new verified compiler backend for CakeML. In *ICFP 2016*, 2016.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI '11*, pages 283–294. ACM, 2011.

OCaml étendu avec du filtrage par comotifs

Paul Laforgue[1] and Yann Régis-Gianas[2]

[1] Univ Paris Diderot, Paris, France
paul.laforgue123@gmail.com
[2] Univ Paris Diderot, Sorbonne Paris Cité, IRIF/PPS, UMR 8243 CNRS, PiR2, INRIA Paris-Rocquencourt, Paris, France
yrg@irif.fr

Résumé

Cet article décrit une extension du langage de programmation OCaml avec de nouvelles constructions pour définir et manipuler des structures de données infinies: les types de données coalgébriques et le filtrage par comotifs.

1 Introduction

Les types de données algébriques et le filtrage par motifs sont des mécanismes d'OCaml particulièrement utiles pour spécifier et définir des programmes travaillant sur des objets finis et inductifs, comme les listes et les arbres par exemple. Le programmeur OCaml apprécie la saveur mathématique et la simplicité calculatoire de ces mécanismes car ce sont les ingrédients d'une programmation sûre et efficace.

Concernant le traitement des objets infinis, le tableau est moins reluisant. Prenons le cas des listes infinies et essayons de définir la liste des entiers naturels:

```
let rec from : int → int list = fun n → n :: from (succ n)
let naturals = from 0
let rec nth : int → 'a list → 'a = fun n s → match n with
  | 0 → List.hd s
  | m → nth (pred m) (List.tl s)
```

Comme OCaml est un langage strict, l'évaluation de naturals provoque une divergence car la construction des entiers naturels s'opère d'un coup. Au contraire, on aimerait que la construction des objets infinis ne se fasse qu'à la demande et partiellement, lorsque l'on doit en *observer* un fragment dont dépend un calcul.

Fort de cette idée, nous pouvons utiliser le mot-clé **lazy** pour retarder systématiquement la construction des listes infinies, et invoquer Lazy.force pour calculer une à une les têtes de ces listes au moment où le calcul le nécessite:

```
type 'a stream = ('a cell) Lazy.t and 'a cell = Cell of 'a * 'a stream
let rec from : int → int stream = fun n → lazy (Cell (n, from (succ n)))
let naturals = from 0
let rec nth : int → 'a stream → 'a = fun n s →
  let Cell (hd, tl) = Lazy.force s in
  match n with
  | 0  → hd
  | m → nth (pred m) tl
```

Si les types et les opérations du module Lazy rendent possible le calcul sur des structures infinies, elles obscurcissent aussi la définition de naturals en introduisant des considérations de bas-niveau sur l'ordre des calculs, absentes des définitions mathématiques habituelles.

Le filtrage par comotifs [1] est une généralisation du filtrage par motifs. En OCaml avec comotifs, un objet infini est défini sans précaution particulière vis-à-vis de la divergence:

```
type 'a stream = {Head : 'a; Tail : 'a stream}
let corec from : int → int stream with
  | (.. n)#Head → n
  | (.. n)#Tail → from (succ n)
let naturals = from 0
let rec nth : int → 'a stream → 'a = fun n s → match n with
  | 0 → s#Head
  | m → nth (pred m) s#Tail
```

Contrairement aux types algébriques qui sont définis par des constructeurs, les types coalgébriques sont définis par des destructeurs, aussi appelés observations. Défini ici en ligne 1, le type coalgébrique 'a stream a deux destructeurs: Head produisant une valeur de type 'a et Tail produisant une valeur de type 'a stream.

En ligne 2, from est défini par filtrage de comotifs. Pour cela, le programmeur raisonne par cas sur les différentes observations, Head et Tail, qui peuvent s'appliquer à from n. Dans un comotif, la notation .. réfère à l'objet infini observé et #O à l'observation O que l'on est en train de définir dans la branche courante. Ainsi, la ligne 3 se lit: "Si on observe la tête de from n alors renvoyer n." Tandis que la ligne 4 se lit: "Si on observe la suite de from n alors calculer from (succ n)". Remarquez que contrairement à un filtrage par motifs classique, les deux branches n'ont pas le même type: la première a le type 'a et la seconde le type 'a stream.

Ce sont donc les applications d'observations qui déclenchent les calculs des structures infinies. De telles applications se retrouvent dans la définition de nth: s#Head déclenche le calcul de la tête de s, s#Tail celui de la suite de s.

Plan La section 2 décrit informellement notre transformation à travers un exemple. Le filtrage par comotifs est un mécanisme de haut niveau qui supporte des constructions avancées allant au-delà de notre exemple introductif (imbrication, inclusion des motifs, prise en compte d'égalités de type). Elles sont présentées en section 3. Une particularité de notre codage est d'introduire les observations comme des objets de première classe et de premier ordre. Cette caractéristique offre un nouveau champ d'action au programmeur et nous l'illustrons en section 4. Finalement, nous concluons et parlons de travaux futurs en section 5.

Prototype Notre prototype est accessible en utilisant le gestionnaire de paquet `opam` et la commande: `opam switch 4.04.0+copatterns`.

2 Transformation

Dès qu'un langage de programmation fonctionnelle est muni de types algébriques généralisés (GADTs) et de polymorphisme de second-ordre, on peut étendre son analyse par motifs en analyse par comotifs à l'aide d'une transformation syntaxique purement locale [2]. En un mot, cette transformation traduit toute valeur coalgébrique définie par une analyse par comotifs en une fonction définie par cas sur la forme des observations de ce type coalgébrique. Dans cette section, nous présentons ce résultat informellement en décrivant les images par notre transformation de la définition du type stream et de l'analyse par comotifs définissant from.

La traduction de la déclaration du type stream commence par introduire un nouveau type stream_obs correspondant aux observations du type stream:

```
type ('a, 'o) stream_obs =
  | Head : ('a, 'a) stream_obs
  | Tail :  ('a, 'a stream) stream_obs
```

Ce type possède deux constructeurs: Head et Tail. Le premier paramètre de type 'a correspond tout simplement à celui de stream, il s'agit du type des éléments de la liste infinie. Le second paramètre 'o est plus intéressant: il correspond au type de retour des observations. Notons que les deux constructeurs instancient différemment 'o : pour Head, 'o vaut 'a alors que pour Tail, 'o vaut 'a stream. Cette spécialisation des schémas de type de chaque constructeur est caractéristique d'un GADT.

Le type coalgébrique stream est traduit par un type algébrique à un seul constructeur nommé Stream par convention. Celui-ci a pour argument une fonction qui attend une observation en entrée et qui est polymorphe vis-à-vis du type de retour de cette observation. Comme il est d'usage en OCaml, ce polymorphisme de second-ordre est introduit grâce à un enregistrement dont l'unique champ dispatch a un type polymorphe:

```
and'a stream = Stream of {dispatch : 'o.('a, 'o) stream_obs → 'o}
```

Ici la fonction dispatch a donc pour type $\forall\sigma.(\alpha,\sigma)$ stream_obs $\to \sigma$. Son type, qui n'est pas sans rappeler celui d'un terme en forme CPS, est le témoin d'une inversion de contrôle entre l'environnement d'évaluation et la valeur coalgébrique.

Naturellement, la transformation du filtrage par comotifs est dirigée par celle des types coalgébriques. Lors de la transformation d'un filtrage par comotifs, une fonction auxiliaire dispatch est systématiquement créée. Dans le cas très simple de notre exemple, cette fonction est une analyse par motifs qui a exactement la même structure que l'analyse par comotifs qu'elle représente. Pour les analyses par comotifs plus complexes de la section 3, l'idée fondamentale reste la même: en effet, les analyses complexes peuvent s'exprimer par imbrication d'analyses par comotifs dites *simples* [2].

```
let rec from : int → int stream = fun n →
  let dispatch : type o.(int, o) stream_obs → o = function
    | Head → n
    | Tail → from (succ n)
  in Stream {dispatch}
```

Notez que cette fonction est bien typée car (i) dans la première branche o et int sont équivalents donc n de type int est bien du type attendu o ; (ii) dans la seconde branche o et int stream sont équivalents, donc from (succ n) de type int stream est bien du type attendu o.

Dernier ingrédient de notre traduction: le cas de l'application des observations. Il s'agit tout simplement d'appliquer la fonction dispatch sur l'observation considérée. On se dote de:

```
let head {dispatch} = dispatch Head
let tail {dispatch} = dispatch Tail
```

ce qui permet de traduire l'observation s#Head en l'application head s.

3 Constructions avancées du filtrage par comotifs

Observations imbriquées On peut chaîner les observations au sein d'une même analyse [1]:

```
let corec fib : int stream with
  | ..#Head → 0
  | ..#Tail : int stream with
  | ..#Tail#Head → 1
  | ..#Tail#Tail → map2 (+) fib fib#Tail
```

En ligne 3, la syntaxe ..#Tail : int stream **with** introduit une sous-analyse par comotifs du stream produit par l'observation Tail de fib. Notez l'annotation de type qui doit obligatoirement préciser le type de ..#Tail: c'est une limite de notre approche dont nous reparlons en section 5.

Mémoïsation dans le filtrage Dans l'exemple précédent, les deux occurrences de fib dans l'appel à map2 produisent deux flux distincts en mémoire ne partageant aucun de leurs calculs: par conséquent, la complexité du calcul d'un élément de rang n de cette suite est exponentielle en n. En préfixant du mot-clé **lazy** la définition de fib, notre transformation introduit une mémoïsation qui garantit que les observations de fib (et de ses sous-flux) sont partagées, ce qui restaure la complexité linéaire de cet algorithme:

```
let corec lazy fib : int stream with
  | ..#Head → 0
  | ..#Tail : int stream with
  | ...#Head → 1
  | ...#Tail → map2 (+) fib fib#Tail
```

Des motifs dans les comotifs Un comotif peut utiliser un motif pour filtrer sur certains sous-ensembles des arguments d'un objet infini. Par exemple, la liste infinie cycle n suivante est de la forme $[\mathsf{n}, \mathsf{n}-1, \ldots, 1, 0, 3, 2, 1, 0, 3, \ldots]$:

```
let corec cycle : nat → nat stream with
  | (. . n)#Head → n
  | (. . Zero)#Tail → cycle (Succ (Succ (Succ Zero)))
  | (. . (Succ n))#Tail → cycle n
```

cycle est un flux paramétré par un entier n. En ligne 2, l'argument n est la réponse à l'observation de la tête de cycle n. L'observation de la suite dépend de la forme de n: si n est Zero, la séquence reprend à Succ (Succ (Succ Zero)), sinon le cycle se poursuit au prédecesseur de n.

Types coalgébriques indexés Comme les GADTs, les types coalgébriques peuvent être indexés de façon à capturer des invariants statiques permettant de justifier l'absence de certains cas absurdes dans les analyses par comotifs. Par exemple, un objet coalgébrique de type ('a, 'b) bounded_iterator encapsule une collection d'élements de type 'a et restreint statiquement la quantité d'éléments de cette collection auquelle le client a accès. On se donne les types suivants:

```
type 'a fuel = Dry : zero fuel | More : 'a fuel → ('a succ) fuel
type ('a, 'b) bounded_iterator = {
  GetVal : 'a;
  Next : ('a, 'b) bounded_iterator ← ('a, 'b succ) bounded_iterator;
}
```

[1] La définition de map2 est laissée en exercice au lecteur.

Le GADT ‘a fuel représente la notion de crédits donnés au client de l’itérateur [2]. Le premier paramètre du type bounded_iterator représente le type des élements de la collection et le second paramètre le nombre de crédits encore disponibles pour le client de l’itérateur. La structure héberge une valeur de type ‘a observable par GetVal. L’observation Next renvoie un nouvel objet dont le nombre de crédits a été décrémenté. Le type de cette observation est indexé par ‘b succ. On contraint ainsi l’observation Next à être uniquement appliquée aux itérateurs qui disposent d’un nombre strictement positif de crédits. Tout module qui offre un type de collection ‘a t muni de deux opérations head et tail peut désormais être étendu par une notion d’itérateur borné à l’aide du foncteur MkIterator suivant:

```
module type Seq = sig
  type 'a t
  val head : 'a t → 'a
  val tail : 'a t → 'a t
end

module MkIterator (S : Seq) = struct
    let corec wrap : type a b.a S.t → b fuel → (a, b) bounded_iterator with
        | (.. l n)#GetVal → S.head l
        | (.. l (More n))#Next → wrap (S.tail l) n
end
```

L’analyse par comotifs de la fonction wrap n’a pas de code pour le cas où le fuel vaut Dry et l’observation est Next. C’est logique! Un tel cas serait mal typé puisqu’il imposerait au paramètre b d’être à la fois de la forme zero et de la forme ‘b succ, deux types incompatibles.

4 Destructeurs de premier ordre et première classe

En section 2, nous avons montré que pour chaque type coalgébrique notre transformation introduit un GADT pour représenter ses observations. La définition de ce GADT suit une convention de nommage documentée ce qui permet au programmeur de faire explicitement référence aux constructeurs de données et de type de ce GADT. De cette façon, les destructeurs peuvent être utilisés dans le calcul comme n’importe quelle autre valeur. Comme ils sont représentés par de simples constructeurs, ils peuvent aussi être comparés ou sérialisés.

Pour illustrer ce mécanisme, considérons le scénario suivant. Le programmeur définit un type enregistrement loc avec trois champs: name de type string, une abscisse x et une ordonnée y de type int et souhaite offrir des fonctions de sélection et de mise-à-jour de ces champs. Pour cela, il est confronté à la pénible tâche de devoir répéter du code:

```
type loc = {name : string; x : int; y : int}
let select_name lc = lc.name      and update_name s lc = {lc with name = s}
let select_x    lc = lc.x         and update_x    b lc = {lc with x = b}
let select_y    lc = lc.y         and update_y    n lc = {lc with y = n}
```

Ce problème est dû au fait que les étiquettes ne sont pas des objets de première classe en OCaml. Nous aimerions disposer de combinateurs génériques tels que select ou update auxquels nous n’aurions qu’à passer l’étiquette concernée en argument. Reprenons donc le même scénario mais cette fois-ci, au lieu d’un type enregistrement, nous utilisons un type coalgébrique.

```
type loc = {Name : string; X : int; Y : int}
```

Le combinateur select peut alors s’exprimer très naturellement. Il suffit d’extraire la fonction dispatch d’une valeur de type loc, puis de l’appliquer à un destructeur d passé en argument.

```
let select (d : 'a loc_obs) (Loc {dispatch} : loc) : 'a = dispatch d
```

[2] On suppose que les constructeurs de types zero et succ sont distincts.

Définir update requiert davantage de travail. En première approximation, on pourrait écrire:

```
let update (type a) (d1 : a loc_obs) (x : a) (Loc {dispatch}) =
  let dispatch : type o.o loc_obs → o = fun d2 →
    if d1 = d2 then x else dispatch d2
  in Loc {dispatch}
```

mais ce programme est mal typé! D'une part, d1 et d2 n'ont pas nécessairement le même type et d'autre part, rien n'informe ici le typeur que x a le type o. Fort heureusement, on peut facilement écrire une fonction de comparaison eq_loc[3] dont le type de retour est plus riche qu'un simple booléen: dans le cas positif, cette fonction retourne une preuve d'égalité entre les indices des types des deux constructeurs:

```
type (_, _) eq = Eq : ('a,' a) eq
val eq_loc : type a b.a loc_obs * b loc_obs → ((a, b) eq) option
```

Dès lors, la fonction update peut être corrigée en utilisant eq_loc:

```
let update (type a) (d1 : a loc_obs) (x : a) (Loc {dispatch}) =
  let dispatch : type o.o loc_obs → o = fun d2 → match eq_loc (d1, d2) with
    | Some Eq → x
    | _ → dispatch d2
  in Loc {dispatch}
```

Cette fonction est bien typée[4] car eq_loc accepte des arguments dont les indices diffèrent et dans le cas Some Eq, l'expression x est typée dans un contexte de typage enrichi par l'égalité a = o qui permet de lui affecter le type o.

5 Conclusion et perspectives

Pour résumer, OCaml peut être étendu par des comotifs sans trop d'effort et la syntaxe des comotifs permet de définir des objets infinis de façon concise, richement typée, et efficace.

Notre approche purement syntaxique impose néanmoins au programmeur d'annoter par leur type toutes les analyses par comotifs. Ceci nous permet de construire une annotation de type pour la fonction dispatch introduite par la transformation et ainsi de nous assurer que le raffinement de type de l'analyse par motifs de dispatch a bien lieu. En déplaçant notre transformation à l'intérieur du moteur de typage d'OCaml, une partie de ces annotations pourraient être automatiquement inférées.

References

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In *ACM SIGPLAN Notices*, volume 48, pages 27–38. ACM, 2013.

[2] Paul Laforgue and Yann Régis-Gianas. Copattern matching and first-class observations in OCaml, with a macro. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*, PPDP '17, pages 97–108, New York, NY, USA, 2017. ACM.

[3] Laissé en exercice au lecteur.

[4] Mais notons qu'elle est très inefficace! Une implémentation plus réaliste pourrait peut-être s'appuyer sur les dictionnaires à clés hétérogènes fournis par le module hmap de Daniel C. Bünzli.

Programmer des chatbots en OCaml avec Watson Conversation Service

Guillaume Baudart, Louis Mandel et Jérôme Siméon

IBM Research AI

Abstract

Les chatbots sont des applications avec une interface conversationnelle. Ils sont composés d'une partie application et d'un moteur de conversation. Dans cet article nous introduisons la bibliothèque `wcs-ocaml` qui permet de programmer les deux parties d'un chatbot en OCaml en se reposant sur le moteur de conversation *Watson Conversation Service* (WCS).

1 Watson Conversation Service

De plus en plus d'entreprises délèguent leur services clients et une partie de leur support technique à des agents conversationnels ou chatbots. Ces chatbots communiquent avec l'utilisateur en langage naturel. Grâce à des avancées techniques récentes, les chatbots sont de plus en plus largement adoptés [14]. Ils peuvent être utilisés dans une page web, dans un système de messagerie, voire au téléphone et sont programmés pour répondre à des questions courantes, aider à la navigation d'un site web, ou pour guider le remplissage de formulaires en ligne.

La Figure 1 présente l'architecture générale d'un chatbot programmé avec le service de conversation d'IBM Watson^{TM}: *Watson Conversation Service* (WCS) [8]. La plupart des services récents de développement de chatbots utilisent une architecture similaire [20]. Elle est composée de deux parties : une application (interface utilisateur, logique de contrôle, orchestration) et un moteur de conversation.

Le moteur de conversation implémenté par WCS reçoit des entrées de l'utilisateur sous forme textuelle. Il analyse ces entrées avec un interprète de langage naturel (*Natural Language Processing*) qui extrait les intentions (ce que l'utilisateur souhaite accomplir) et les entités (les sujets dont l'utilisateur parle). Le résultat de cette analyse est envoyé à l'interprète de dialogue (*Dialog Interpreter*) qui pilote la conversation et décide de la réponse à donner à l'utilisateur.

L'application contrôle les interactions avec l'utilisateur (par exemple, lire les entrées et renvoyer les réponses). Elle est également en charge de maintenir l'état du chatbot. Le service de conversation a une interface fonctionnelle et ne conserve pas d'état entre deux tours de conversation. Enfin, l'application peut appeler des services externes (*e.g.*, service météo) pour

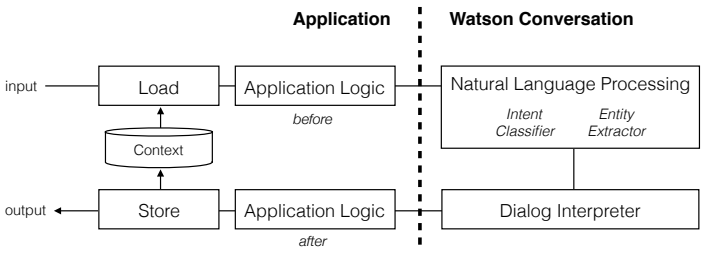


Figure 1: Architecture d'un chatbot programmé avec WCS.

compléter une réponse, réaliser une action (*e.g.*, mise à jour d'une base de données) ou faire des calculs arbitraires.

WCS offre un langage graphique dédié à la spécification des conversations. La partie application appelle le moteur de conversation à l'aide d'une API REST [9] et peut donc être écrite dans n'importe quel langage. Cette approche donne une grande liberté aux programmeurs, mais peut conduire à des difficultés de coordination entre les deux parties d'un chatbot.

Dans cet article, nous présentons `wcs-ocaml` qui embarque WCS dans OCaml et permet de programmer l'intégralité d'un chatbot dans le même environnement. La bibliothèque `wcs-ocaml` est *open source* (https://ibm.github.io/wcs-ocaml) et peut être installée à l'aide du gestionnaire de paquets opam.

Le reste de l'article est organisé de la manière suivante. La section 2 décrit l'intégration en OCaml du langage dédié à la programmation de WCS. La section 3 montre comment appeler le service WCS pour programmer une application de chatbot complète. Les travaux connexes sont discutés dans la section 4 avant de conclure.

2 Spécifier le moteur de conversation WCS

En WCS, les programmes sont appelés *workspaces*. Un workspace est un objet JSON qui inclut les définitions des *intentions*, des *entités*, et du *dialogue*. Pour illustrer ces différentes notions, considérons KnockKnockBot, un chatbot qui raconte des blagues.

> *Bot: Knock Knock*
> *User: Who's there?*
> *Bot: Broken Pencil*
> *User: Broken Pencil who?*
> *Bot: Never mind it's pointless...*

Intentions Les intentions correspondent à ce que l'utilisateur cherche à accomplir. Par exemple, on peut chercher à savoir qui frappe à la porte. Une intention est définie par un ensemble d'exemples qui servent à entraîner l'interprète de langage naturel[1].

```
let who_intent =
  Wcs.intent "Who" ~examples: [ "Who's there?"; "Who is there?"; "Who are you?"; ] ()
```

```
val who_intent : Wcs_t.intent_def
```

Ce type de définition donne une certaine souplesse à l'interprète. Ainsi la phrase *Who's that stumbling around in the dark?* sera associée avec succès à l'intention `who_intent` bien que ne figurant pas dans la liste d'exemples.

Entités Les entités sont les sujets évoqués par l'utilisateur. Dans notre exemple, on peut définir une entité pour les noms des personnages évoqués dans la blague. Une entité peut prendre plusieurs valeurs et chaque valeur possible est associée à une liste de synonymes.

```
let char_entity =
  Wcs.entity "Characters" ~values: [ "Broken Pencil", ["Damaged Pen"; "Fractured Pencil"] ] ()
```

```
val char_entity : Wcs_t.entity_def
```

Pour être détecté par l'interprète, un des synonymes de l'une des valeurs doit apparaître dans le texte soumis par l'utilisateur.

[1] Le code présenté utilise la version 2017-05-26.03 de `wcs-ocaml` ™

Dialogue Le dialogue permet de spécifier le comportement du moteur de conversation par un automate qui réagit aux intentions et entités détectées par l'interprète de langage naturel, et aux informations directement envoyées par l'application. Par exemple la structure de notre blague peut être retranscrite par un automate à trois états : 1) `knock` amorce la blague, 2) `whoisthere` répond à la question avec le nom du personnage, 3) `answer` conclut avec la chute.

```
let knockknock who_intent char_entity answer =
  let knock =
    Wcs.dialog_node ("Knock")
      ~conditions_spel: (Spel.bool true)
      ~text: "Knock knock" ()
  in
  let whoisthere =
    Wcs.dialog_node ("WhoIsThere")
      ~conditions_spel: (Spel.intent who_intent)
      ~text: (entity_value char_entity)
      ~parent: knock ()
  in
  let answer =
    Wcs.dialog_node ("Answer")
      ~conditions_spel: (Spel.entity char_entity ())
      ~text: answer
      ~parent: whoisthere
      ~context: (`Assoc ["return", `Bool true]) ()
  in
  [ knock; whoisthere; answer ]
```

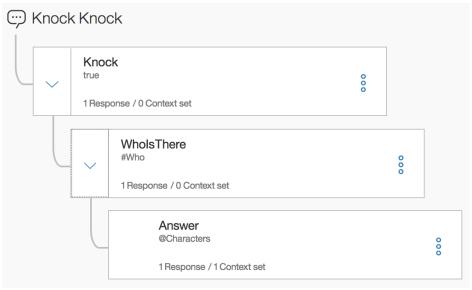


Figure 2: Interface graphique.

```
val knockknock :
Wcs_t.intent_def -> Wcs_t.entity_def -> string -> Wcs_lib.Wcs_t.dialog_node list
```

Les états de l'automate, appelés *nœuds de dialogue* sont construits à l'aide de la fonction `Wcs.dialog_node`. Ce constructeur prend en argument le nom du nœud et un ensemble d'options : la condition d'entrée exprimée dans le langage dédié SpEL [21] (`~condition_spel`), le texte de la réponse du chatbot (`~text`), le père du nœud dans l'automate (`~parent`). Un nœud peut également définir des champs dans l'objet de contexte (`~context`). Cet objet représente l'état du système, il permet de passer des informations entre deux tours de conversation. Ici, le nœud `answer` définit dans le contexte la variable booléenne `return` qui permet à l'application de décider si la conversation est terminée.

La bibliothèque `wcs-ocaml` inclut un module pour construire les expressions SpEL soit comme un arbre de syntaxe, soit directement dans la syntaxe SpEL comprise par WCS [10]. Ainsi, la condition du nœud `answer` peut également être écrite : `~condition:"@Characters"` dénotant une condition qui est vraie lorsque l'entité '`Characters`' est reconnue. L'avantage de la première méthode est de produire une erreur à la compilation si l'entité `char_entity` n'existe pas.

Workspace Nous avons enfin tous les éléments pour définir un workspace complet qui peut être exécuté par Watson Conversation : une liste d'intentions, une liste d'entités et un ensemble de nœuds de dialogue.

```
let ws_knockknock =
  Wcs.workspace "Knock Knock"
    ~intents: [ who_intent ] ~entities: [ char_entity ]
    ~dialog_nodes: (knockknock who_intent char_entity "Never mind it's pointless") ()

val ws_knockknock : Wcs_t.workspace
```

À partir de cette définition de workspace avec `wcs-ocaml`, il est possible de générer l'objet JSON correspondant et de le charger dans WCS en utilisant l'API ou l'interface graphique. Le workspace peut ensuite être utilisé par une application ou simplement visualisé avec l'éditeur de WCS (figure 2).

Dans `wcs-ocaml`, les workspaces sont des structures de données fortement typées qui sont générées automatiquement avec atdgen [12] en suivant la description des objets JSON manipulés par WCS [9]. La bibliothèque `wcs-ocaml` permet ainsi de manipuler programmatiquement les workspaces.

Le module `Wcs` de construction de workspaces utilise abondamment les labels, ce qui offre une syntaxe plus proche de JSON. Les labels correspondent aux champs des objets JSON dans le workspace et peuvent être donnés dans un ordre arbitraire. Le système de type offre des garanties statiques sur la validité des noms des champs et permet de distinguer les champs nécessaires et optionnels.

De plus, ce module utilise la liaison de nom d'OCaml pour faire la liaison de noms d'objets dans le workspace. Par exemple le lien entre un nœud de dialogue et son prédécesseur (qui est soit le nom du nœud père soit celui du fils précédent) est une simple chaîne de caractères dans le JSON du workspace. En `wcs-ocaml`, le constructeur pour un nœud de dialogue prend en argument un autre un nœud de dialogue correspondant à son prédécesseur, ce qui permet de garantir au moment de la compilation que tous les liens pointent vers des nœuds existants.

3 Programmer une application de chatbot

Le workspace n'est qu'une partie d'un chatbot. Comme illustré sur la figure 1, WCS a aussi besoin d'une partie application. La bibliothèque `wcs-ocaml` permet également d'invoquer un workspace déployé sur WCS avec l'appel REST `message` [9].

Invoquer des workspaces Cette fonction `message` prend en argument le texte de l'utilisateur et le *contexte*, un objet JSON qui contient les informations nécessaires à WCS comme le nom du nœud de dialogue courant. Le contexte peut également contenir les données qui permettent de communiquer entre l'application et le workspace. Par exemple, le nœud `answer` (section 2) définit une variable de contexte `return` pour signaler à l'application que le dialogue est terminé.

À partir de la fonction `message`, on peut définir une fonction `interpret` qui va interpréter un tour de conversation pour un workspace `id` et extraire la valeur du champ `return` du contexte. Pour utiliser WCS, la fonction a besoin des certificats d'authentification du service. Ils sont donnés avec l'argument `creds`.

```
let rec interpret creds id req =
  let resp = Wcs_call.message creds id req in
  let ctx, return = Json.take resp.msg_rsp_context "return" in
  { resp with msg_rsp_context = ctx }, return
```

Le contexte est un objet JSON arbitraire représenté avec la bibliothèque Yojson. Il n'y a donc que peu d'information de typage pour les variables utilisées pour communiquer entre l'application et le workspace. Mais en utilisant `wcs-ocaml` pour construire à la fois le workspace et l'application, il devient possible de partager des structures de données. Dans l'exemple, on aurait pu définir un type `return = { return: bool; }` et utiliser des fonctions de sérialisation en JSON dans le nœud `answer` et de désérialisation dans la fonction `interpret` pour échanger les données de façon plus sure.

Une version étendue de cette fonction `interpret` est fournie par le module `Wcs_bot` de `wcs-ocaml`. En plus d'extraire la valeur du champ `return`, elle permet d'exécuter des sous-dialogues. Les sous-dialogues sont une des briques de base pour la programmation de chatbots par composition d'autres chatbots. Cela correspond à de l'appel de fonction. Enfin, la fonction `Wcs_bot.interpret` interprète la variable de contexte `skip_user_input` qui active plusieurs transitions de l'automate de dialogue sans demander de nouvelles entrées à l'utilisateur. La signature complète de cette fonction est la suivante :

```
val interpret :
  ?before:(message_request -> message_request) ->
  ?after:(message_response -> message_response) ->
  credential -> string -> message_request -> string * message_response * json option
```

En plus de la réponse et de la valeur de `return` extraite du contexte, cette fonction renvoie l'identifiant du workspace à exécuter lors du prochain tour de conversation (l'identifiant peut changer quand on appelle un sous-dialogue). Les arguments optionnels `~before` et `~after` permettent d'exécuter des fonctions arbitraires avant et après l'appel à `message` comme illustré sur la figure 1.

Programmer l'application La fonction `Wcs_bot.interpret` n'exécute qu'un seul tour de conversation. Mais nous pouvons maintenant définir simplement un chatbot qui traduit la blague à la volée pour un utilisateur francophone :

```
let exec creds ws_knockknock =
  let rec loop ws_id ctx =
    let txt = input_line stdin in
    let req = Wcs.message_request ~text: txt ~context: ctx () in
    let ws_id, rsp, return =
      Wcs_bot.interpret ~before: englify ~after: frenchify creds ws_id req
    in
    let () = List.iter print_endline rsp.msg_rsp_output.out_text in
    begin match return with
    | Some v -> v
    | None -> loop ws_id rsp.msg_rsp_context
    end
  in
  loop ws_knockknock `Null
```

Les fonctions `englify` et `frenchify` permettent de passer de l'anglais au français en utilisant par exemple l'API Watson Language Translator [11], ou l'API Google Translate [7]. Notre blague devient alors :

Bot: Toc Toc
User: Qui est là ?
Bot: Crayon Brisé
User: Crayon Brisé qui ?
Bot: Peu importe c'est inutile...

Déployer un chatbot La bibliothèque `wcs-ocaml` fournit aussi les fonctions pour déployer les workspaces sur le service WCS. Il est ainsi possible d'intégrer le déploiement des workspaces dans l'application. Cela évite des problèmes d'incompatibilité de versions entre les deux parties du système. Faciliter le déploiement de l'application permet des mises à jour fréquentes. Les programmeurs peuvent ainsi prendre rapidement en compte les retours d'expériences des utilisateurs pour améliorer le chatbot [22].

4 État de l'art

Les systèmes de dialogue permettent le développement d'interfaces interactives basées sur le langage naturel [1]. La popularités des assistants personnels intelligents (tels que Siri ou Alexa) contribue à l'intérêt croissant dans ce domaine. McTear [17] et Jurafsky et Martin [13, Chapitre 29] donnent un aperçu des différentes technologies disponibles. L'utilisation d'automates finis est un modèle relativement expressif mais assez bas-niveau [8, 22]. Les *frames* sont une alternative spécialisée pour remplir des formulaires [2, 15]. Les systèmes plus récents [20] combinent un ou plusieurs services (par exemple pour obtenir un interprète de langage naturel plus performant) avec une bibliothèque intégrée dans un langage de programmation [18, 5].

La notion de workspace dans Watson Conversation System fournit un modèle attrayant, car plus déclaratif et offrant une interface de visualisation. L'un des inconvenient de cette approche est la nécessité de développer les chatbots en deux parties : le workspace et l'application correspondante. L'utilisation de `wcs-ocaml` fournit une solution complète et répond à ce problème. Plusieurs autres SDK pour WCS sont disponibles [8] (entre autres pour Python, Node et Java). `wcs-ocaml` est le premier SDK pour WCS qui intègre la construction du workspace et de l'application et qui fournit un model pour l'appel de sous-dialogue. Il existe également d'autres services de chatbot associés à leur propre SDK [6, 19].

5 Conclusion

Nous avons présenté `wcs-ocaml`, une bibliothèque *open source* qui permet de programmer des chabots en OCaml avec *Watson Conversation Service*. La bibliothèque `wcs-ocaml` a été utilisée pour développer ChessBot (un chatbot pour ChessEye[2]) et RuleBot (un chatbot pour l'édition de *règles métiers* [4]). Elle a également été utilisée conjointement avec la bibliothèque d'analyse statique *T.J. Watson Libraries for Analysis* (WALA)[3] pour le déboggage de workspaces WCS.

Pour le développement de ChessBot et RuleBot, nous avons dû utiliser des formes de compositions plus avancées que les sous-dialogues: pipeline, parallélisme et préemption. Ces composition s'expriment naturellement dans un langage réactif de haut niveau tel que ReactiveML [16]. La bibliothèque `wcs-ocaml` a permis une intégration rapide en ReactiveML, ce langage étant lui aussi fondé sur OCaml.

RuleBot repose également sur un mécanisme d'échange de données constant (des fragments de l'arbre de syntaxe du langage de règles) entre l'application et le service WCS. L'utilisation de `wcs-ocaml` pour programmer l'application et le workspace nous a permis d'assurer que la construction des ces fragments d'arbre de syntaxe (qui sérialise un type OCaml en structure JSON) et l'interprétation des réponses du moteur de conversation (qui reconstruit un type OCaml à partir de la structure en JSON) sont cohérents.

Remerciements Nous tenons à remercier Avraham Shinnar pour son travail sur le SDK typescript qui a été une source d'inspiration pour `wcs-ocaml`. Nous remercions également Timothy Bourke pour le package LaTeXchecklistings [3] qui nous a permis de vérifier le code présenté dans cet article.

[2] https://github.com/chesseye/chesseye
[3] http://wala.sourceforge.net/wiki/index.php

References

[1] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch. Natural language interfaces to databases – an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[2] D. G. Bobrow, R. M. Kaplan, M. Kay, D. A. Norman, H. Thompson, and T. Winograd. GUS, a frame-driven dialog system. *Artificial Intelligence*, 8(2):155–173, 1977.

[3] T. Bourke and M. Pouzet. *The checklistings package*, 2015.

[4] M. J. Boyer and H. Mili. Agile business rule development. In *Agile Business Rule Development*, pages 49–71. Springer, 2011.

[5] Facebook. Messenger platform, 2017. https://developers.facebook.com/docs/messenger-platform/ (Retrieved October 2017).

[6] Facebook. Messenger platform, 2017. https://developers.facebook.com/docs/messenger-platform/ (Retrieved December 2017).

[7] Google. Google translate service, 2017. https://cloud.google.com/translate/ (Retrieved October 2017).

[8] IBM. *Overview of the IBM Watson Conversation service*, 2017. https://www.ibm.com/watson/developercloud/doc/conversation/index.html.

[9] IBM. Watson conversation service api, 2017. https://www.ibm.com/watson/developercloud/conversation/api/v1 (Retrieved October 2017).

[10] IBM. Watson conversation service documentation, 2017. https://console.bluemix.net/docs/services/conversation/expression-language.html (Retrieved October 2017).

[11] IBM. Watson language translator service, 2017. https://www.ibm.com/watson/services/language-translator/ (Retrieved October 2017).

[12] M. Jambon. *atdgen documentation*, 2017. https://mjambon.github.io/atdgen-doc (Retrieved October 2017).

[13] D. Jurafsky and J. H. Martin. *Speech and Language Processing*. Prentice Hall, second edition, 2009.

[14] R. Kaplan. Beyond the GUI: It's time for a conversational user interface. *Wired*, 2013.

[15] B. Lucas. VoiceXML for web-based distributed conversational applications. *Communications of the ACM (CACM)*, 43(9):53–57, 2000.

[16] L. Mandel, C. Pasteur, and M. Pouzet. ReactiveML, ten years later. In *Proceedings of 17th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming*, 2015.

[17] M. F. McTear. Spoken dialogue technology: Enabling the conversational interface. *ACM Computing Surveys (CSUR)*, 34(1):90–169, 2002.

[18] Microsoft. Bot framework documentation, 2017. https://docs.microsoft.com/en-us/bot-framework/ (Retrieved October 2017).

[19] Microsoft. Bot framework documentation, 2017. https://docs.microsoft.com/en-us/bot-framework/ (Retrieved December 2017).

[20] A. Patil, K. Marimuthu, R. Niranchana, et al. Comparative study of cloud platforms to develop a chatbot. *International Journal of Engineering & Technology*, 6(3):57–61, 2017.

[21] Spring. Spring expression language (SpEL), 2017. https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html#expressions (Retrieved October 2017).

[22] J. D. Williams, N. B. Niraula, P. Dasigi, A. Lakshmiratan, C. G. J. Suarez, M. Reddy, and G. Zweig. Rapidly scaling dialog systems with interactive learning. In *International Workshop on Spoken Dialog Systems*, 2015.

Simplification efficace pour la théorie des tableaux

Benjamin Farinier[1], Robin David[2], and Sébastien Bardin[1]

[1] CEA LIST, Université Paris-Saclay, France
prenom.nom@cea.fr
[2] Quarkslab, Paris, France
rdavid@quarkslab.com

Résumé

La théorie des tableaux tient une place essentielle en vérification de logiciels puisqu'elle permet de modéliser la mémoire ou certaines structures de données. Les techniques de simplification dites de *read-over-write* usuellement mises en place remontent la chaîne des écritures faites dans un tableau pour substituer aux lectures la valeur écrite à leur indice. Cependant, lorsque le parcours de cette chaîne est non borné, cette phase de simplification est quadratique et ne passe pas à l'échelle. Inversement, de nombreux cas de *read-over-write* sont manqués lorsque ces simplifications sont bornées. Par ailleurs, dans le cas particulier où toutes les écritures et lectures ont lieu à des indices constants, une mappe peut être utilisée pour simplifier efficacement toutes les occurrences de *read-over-write*. Nous proposons dans ce papier une nouvelle approche basée sur une structure de données en liste de mappes qui, tout en passant à l'échelle, permet de retrouver les avantages d'une représentation en mappe dans le cas où toutes les écritures et lectures sont à indices constants, et d'étendre en partie ces avantages aux cas des écritures ou lectures symboliques.

1 Introduction

Les procédures de décision automatique pour le problème de la satisfiabilité modulo théorie sont au cœur de nombreuses techniques récentes de vérification formelle [4, 1, 3]. Notamment, la théorie des tableaux [10] tient une place essentielle en vérification de logiciels puisqu'elle permet de modéliser la mémoire ou certaines structures de données. Intuitivement, étant donné un ensemble $\mathcal{I}$ d'indices et un ensemble $\mathcal{E}$ d'éléments, cette théorie décrit l'ensemble $\mathsf{Array}\,\mathcal{I}\,\mathcal{E}$ des tableaux associant à chaque indice $i \in \mathcal{I}$ un élément $e \in \mathcal{E}$ [1]. Ces tableaux sont définis via l'axiomatisation des opérations de lecture (select) et d'écriture (store) énoncée en Figure 1.

$$\begin{array}{ll} \mathsf{select} : \mathsf{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E} & \forall a\, i\, e.\, \mathsf{select}\,(\mathsf{store}\ a\ i\ e)\, i = e \\ \mathsf{store} : \mathsf{Array}\,\mathcal{I}\,\mathcal{E} \to \mathcal{I} \to \mathcal{E} \to \mathsf{Array}\,\mathcal{I}\,\mathcal{E} & \forall a\, i\, j\, e.\, (i \neq j) \Rightarrow \mathsf{select}\,(\mathsf{store}\ a\ i\ e)\, j = \mathsf{select}\ a\ j \end{array}$$

FIGURE 1 – Théorie des tableaux

Malgré sa simplicité, le problème de la satisfiabilité pour la théorie des tableaux est NP-complet [2]. En effet il implique de pouvoir décider des égalités entre indices lus et écrits lors des lectures après écritures dans un tableaux (*read-over-write* ou ROW), décisions pouvant amener à des distinctions de cas en cascade. Les techniques usuelles de résolution consistent à éliminer un maximum de lectures lors d'un prétraitement par simplification des ROW [8] en utilisant les axiomes de la Figure 1 comme des règles de réécriture, puis éventuellement à introduire

1. Les tableaux logiques sont des tables d'association infinies définies de manière implicite comme une succession d'écritures. La notion de taille se retrouve par combinaison de théories (ex. : arithmétique des indices).
2. Réduction du problème de l'équivalence de programmes avec tableaux [7].

paresseusement les axiomes de ROW restants [2]. Ce n'est cependant pas satisfaisant lorsqu'on considère de longues chaînes d'écritures, comme on peut le rencontrer dans les approches à base de dépliage telles que l'exécution symbolique [3] ou la vérification de modèles bornée [4] [3]. La théorie des tableaux peut alors devenir un goulot d'étranglement pour la résolution des formules. Notamment, la phase de simplification des ROW est souvent très limitée car remonter pour toutes les lectures la chaîne des écritures correspondantes peut être quadratique en le nombre d'opérations de tableau et ne passe pas à l'échelle. Inversement, de nombreux cas de simplification des ROW sont manqués lorsque ce parcours est borné. C'est typiquement le cas quand toutes les écritures et lectures ont lieu à des indices constants, là où une implémentation efficace de mappe [4] aurait pu être utilisée pour simplifier toutes les occurrences de ROW.

Contributions et travaux futurs. Nous présentons nos travaux en cours sur une nouvelle approche de simplification des ROW par *liste de mappes* et *normalisation des bases* (LMNB). Cette nouvelle approche permet, tout en passant à l'échelle, de retrouver les avantages d'une représentation en mappe dans le cas où toutes les écritures et lectures sont à indices constants, et d'étendre en partie ces avantages aux cas des écritures ou des lectures symboliques. Nos premiers résultats expérimentaux montrent que LMNB passe à l'échelle sur des formules de très grande taille et comportant de nombreux tableaux, par exemple issues d'exécution symbolique [6]. Le gain de temps apporté par LMNB à la résolution par solveur SMT est significatif, passant de plusieurs heures à quelques secondes sur des cas extrêmes. Nos travaux futurs comprennent une évaluation expérimentale plus poussée ainsi qu'une intégration profonde de LMNB dans un solveur SMT.

```
(declare-fun ebp () (BitVec 32))
(declare-fun esp () (BitVec 32))
(declare-fun memory0 () (Array _ _))

(define-fun memory1 () (Array _ _)
  (store memory1 (esp - 16) 1415)

(assert (esp = (ebp - 64)))
(assert ((select memory1 9265)
       = (select memory1 (ebp + 48))))
```

```
(declare-fun ebp () (BitVec 32))
(declare-fun esp () (BitVec 32))
(declare-fun memory0 () (Array _ _))

(define-fun memory1 () (Array _ _)
  (store memory1 (ebp + 48) 1415)

(assert (esp = (ebp - 64)))
(assert ((select memory0 9265) = 1415))
```

FIGURE 2 – Une formule SMT de la théorie des tableaux (gauche) et sa simplification (droite).

2 État de l'art sur la simplification des ROW

La capacité et l'efficacité à simplifier les lectures et écritures sont intrinsèquement liées à la représentation qui est choisie pour modéliser un tableau et les différentes versions résultant des écritures successives qui y sont faites. Nous présentons dans cette section deux représentations usuelles, celle en liste et celle en mappe, avec leurs avantages et inconvénients.

Représentation en liste. La représentation la plus simple modélise les différentes versions d'un tableau par une liste chaînée des écritures faites dans le tableau depuis sa déclaration. Un tableau fraîchement déclaré est représenté par une liste vide, tandis que le tableau obtenu par

3. En vérification déductive, les invariants fournis par l'utilisateur permettent de pallier à cela.
4. http://www.culture.fr/franceterme/result?francetermeSearchTerme=mappe

FIGURE 3 – Représentation des tableaux en liste d'écritures.

l'écriture d'un élément e à un indice i est représenté par un nœud contenant le couple (i, e) et pointant vers la liste représentant le tableau écrit. La figure 3 illustre cette modélisation. Cette approche est la plus générale possible puisqu'elle utilise directement la chaîne d'écritures comme représentation. Elle est pour cette raison celle adoptée par la plupart des solveurs.

Pour simplifier une lecture à un indice j, on essaie de décider si $i = j$ est valide pour le couple (i, e) contenu dans le nœud de tête de la liste représentant le tableau lu. Si l'on y parvient, alors on est dans une situation de ROW et on peut remplacer la lecture par la valeur e. Si l'on n'y parvient pas, on essaie de décider si $i \neq j$ est valide, et si tel est le cas on réitère sur le nœud suivant dans la liste chaînée, sinon on s'arrête.

Le problème inhérent à cette représentation est l'augmentation du coût des simplifications au fur et à mesure que le nombre d'écritures augmente. Comme le montre la Figure 6, Section 4, ce coût devient rédhibitoire lorsqu'on traite des formules de taille importante. En effet, on peut être amené pour chaque lecture à remonter entièrement la liste des écritures, soit une complexité quadratique dans le pire cas. Cette augmentation est d'autant plus dommageable que ce pire cas intervient dans les situations où la simplification est maximale, à savoir quand la validité de l'inégalité entre indices peut être à chaque fois prouvée, et la lecture remplacée par un accès dans le tableau initial. Un palliatif de cette augmentation du coût est de borner la remontée de la liste à un nombre de nœuds fixé, limitant de fait le temps de simplification, mais au prix d'une moindre simplification (Figure 6, Section 4).

Représentation en mappe. Dans le cas particulier où tous les indices de lecture et d'écriture sont comparables, par exemple dans le cas où tous les indices sont constants, une mappe persistante permettant des accès logarithmiques en lecture et écriture peut être utilisée pour simplifier toutes les occurrences de ROW. Par comparable, on entend qu'une opération de comparaison $\prec$ est définie et décidable pour toute paire d'indices présents dans la formule. Si une telle hypothèse peut être parfois vérifiée, elle ne l'est pas forcément, comme par exemple quand un des indices est un symbole non interprété. Cette représentation est utilisée en exécution symbolique [3] lors de la construction des formules logiques dans le cas d'une politique de concrétisation forte [9, 5] afin de limiter l'introduction de tableaux.

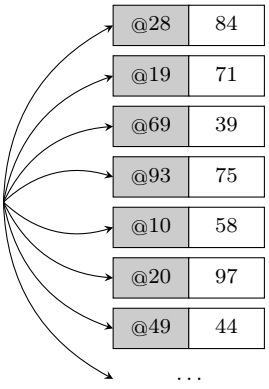


FIGURE 4 – Représentation des tableaux en mappe d'écritures.

Ici, un tableau fraîchement déclaré est représenté par une mappe vide dont les sortes des indices et éléments concordent avec celles du tableau, et le tableau obtenu par l'écriture d'un élément e à un indice i est représenté par la mappe du tableau écrit à laquelle on a simplement ajouté l'élément e à l'indice i, comme illustré par la figure 4. La simplification d'une lecture à un indice j revient alors à sa substitution par l'élément que la mappe associe à j. Dans le cas où aucun élément n'est trouvé, la lecture a lieu sur le tableau initial. On peut alors soit substituer le tableau lu par le tableau initial, soit remplacer la lecture par un symbole nouvellement introduit. On veillera à utiliser le même symbole pour des lectures ayant lieu à un même indice, chose possible à assurer puisque tous les indices sont supposés comparables.

3 Simplification efficace des ROW

Nous présentons maintenant notre technique LMNB pour la simplification des ROW. LMNB combine deux ingrédients essentiels : une représentation par liste de mappes permettant le passage à l'échelle, et une comparaison efficace des indices par normalisation des bases.

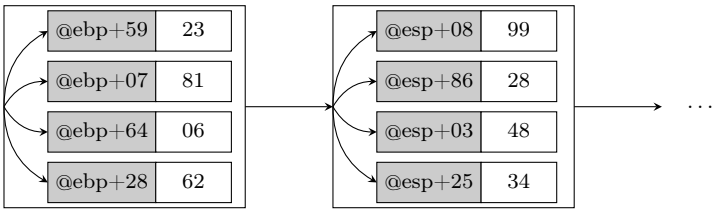


FIGURE 5 – Représentation des tableaux en liste de mappes d'écritures.

Représentation en liste de mappes. La représentation en liste de mappes que nous proposons a pour but premier de combiner les avantages de la représentation en mappe dans les cas où tous les indices sont comparables, tout en gardant la généralité de la représentation en liste dans les autres cas. Dans cette modélisation, les nœuds de la liste sont des mappes associant à des indices tous comparables deux à deux les éléments qui y ont été écrits. La figure 5 donne une illustration de cette modélisation. Ainsi, un tableau fraîchement déclaré est représenté par une liste de mappes vide, tandis que pour représenter le tableau obtenu par l'écriture d'un élément e à un indice i, on distingue deux cas. Soit i est comparable avec tous les indices des éléments déjà insérés dans la mappe de tête de la liste chaînée représentant le tableau écrit. Auquel cas on ajoute à cette mappe l'élément e à l'indice i. Si ce n'est pas le cas, alors on ajoute à la liste un nouveau nœud contenant une mappe singleton associant seulement e à i.

La simplification des ROW se fait comme suit. Pour une lecture à un indice j, soit les indices de la mappe contenue dans le nœud de tête de la liste chaînée représentant le tableau lu sont comparables avec j. Alors si j fait partie des indices présents dans la mappe, on substitue la lecture par l'élément qui lui est associé. Si j n'est pas présent dans la mappe, on réitère avec le nœud suivant. Soit j n'est pas comparable avec les indices présents dans la mappe. Alors pour tous les indices i présents dans la mappe, on essaie de décider si $i = j$ ou si $i \neq j$ est valide. S'il existe un indice i tel que $i = j$, alors on substitue la lecture par l'élément associé à i. Si pour tous les indices i présents dans la mappe $i \neq j$, alors on réitère avec le nœud suivant. Enfin s'il existe un indice i tel que ni $i = j$ ni $i \neq j$ ne peut être prouvé valide, alors on s'arrête.

Intuitivement, l'intérêt de cette représentation est que son comportement varie entre celui de la représentation en liste et celui de la représentation en mappe en fonction de la proportion d'indices comparables deux à deux. En effet, quand tous les indices sont comparables, la liste ne contient qu'une seule mappe de tous les indices, soit l'équivalent de la représentation par mappe, tandis que lorsqu'aucune paire d'indices n'est comparable, la liste ne contient que des mappes singletons, soit l'équivalent de la représentation par liste.

Normalisation des bases et partage de sous-termes. La décision des égalités et inégalités entre indices de lecture et d'écriture est dans notre approche basée sur un système de règles de réécritures. Nous détaillons ici deux étapes originales qui contribuent fortement à cette décision.

La première est la *normalisation des bases*. Un motif fréquent dans des formules issues de l'analyse de programme faisant intervenir des tableaux est la lecture ou l'écriture à un indice

TABLE 1 – Nombre de TIMEOUT par solveur, temps de résolution en seconde par solveur et nombre de select, cumulés pour l'ensemble des 6.590 formules en fonction du prétraitement. Le TIMEOUT (1.000 secondes) n'est pas inclus dans la somme des temps de résolution.

	#TIMEOUT et temps de résolution								#select	
	Boolector		CVC4		Yices		Z3		tous tableaux	non initiaux
sans simplification	0	120,4	2	1.489	2	2,729	0	835,5	1.463.016	1.462.716
liste-16	0	117,1	2	1.413	2	2,703	0	799,2	1.089.065	1.066.493
liste-256	0	114,6	2	1.383	2	2,722	0	591,6	819.326	774.221
map	0	113,0	2	1.332	2	2,584	0	586,0	819.326	774.221
LMNB	0	72,4	2	1.177	2	2,608	0	419,3	75.772	15.028

défini comme la somme d'une *base* (n'importe quel terme contenant au moins une variable symbolique) et d'une constante. L'avantage de traiter des termes sous cette forme est qu'il suffit de décider de l'égalité de leur base pour pouvoir les comparer par la constante ajoutée. Il est donc particulièrement intéressant de normaliser un maximum d'indices non seulement en jouant sur l'associativité et la commutativité de certaines opérations, mais aussi en substituant les variables liées à des termes eux-même sous cette forme par leur contenu.

La seconde est le *partage de sous-termes*. Cette amélioration n'est pas nouvelle en soit, mais a un intérêt original dans ce contexte. En plus d'améliorer la normalisation des bases en donnant un même nom aux sous-termes égaux, elle permet d'éliminer un inconvénient de la simplification des ROW. En effet, si l'on imagine les tableaux comme des ensembles de variables indexées, la simplification des ROW peut alors être vue comme une phase d'*inlining* spécifique qui peut dans certains cas amener à une explosion de la taille des termes. Cette situation survient lorsqu'après une écriture d'un élément e à un indice i, plusieurs lectures à l'indice i sont simplifiées. Il peut en résulter un grand nombre de copies du terme e, terme pouvant lui même faire intervenir des lectures à simplifier. En nommant et partageant les termes lus et écrits dans les tableaux cette phase de partage prévient cet écueil.

4 Implémentation et évaluation expérimentale

Afin d'évaluer l'efficacité de notre approche, nous avons implémenté les simplifications de ROW en tant que prétraitement pour des formules SMT de la logique QF_ABV (*quantifier-free formulas over the theory of bitvectors and arrays*), et cela pour les différentes représentations précédemment décrites. L'avantage de l'approche en prétraitement est qu'elle permet d'être indépendante du solveur utilisé pour résoudre la formule simplifiée, et donc d'évaluer notre approche sur plusieurs d'entre eux. Un inconvénient de cette approche est qu'elle ne profite pas des différents composants du solveur. À terme, une intégration plus profonde serait souhaitable.

On évalue les performances des simplifications pour les différentes représentations et améliorations présentées selon quatre critères : 1. le temps de simplification, 2. le temps de résolution après simplification, 3. le nombre total de lectures après simplification, 4. le nombre total de lectures après simplification ne portant pas sur un tableau initial, les lectures sur ces derniers n'ayant pas à être simplifiées. Les formules utilisées pour évaluer les performances sont générées par le moteur d'exécution symbolique BINSEC [6] à partir de codes exécutables réels. Ce type de formule est caractérisé par un très grand nombre d'indices de lecture et d'écriture sous forme de *base + constante*. La présence de différentes bases empêche l'utilisation de la structure en mappe simple et justifie l'utilisation de notre représentation en liste de mappes. Les performances ont été évaluées sur un processeur Intel(R) Xeon(R) CPU E5-2660 v3 @ 2,60GHz.

La table 1 donne pour les solveurs Boolector, CVC4, Yices et Z3 [5] le temps de résolution et montre l'effet de la simplification sur un ensemble de 6.590 formules obtenues par exécution symbolique statique de différents binaires. Ces formules sont relativement petites, et de fait le temps de prétraitement est ici négligeable, de l'ordre du centième de seconde, et ce quelle que soit la représentation. Il n'est donc pas évalué. La durée de TIMEOUT est fixée à 1.000 secondes (16m40s). La représentation en liste est évaluée avec deux bornes, une à 16 afin de montrer l'impact d'une borne trop petite sur les simplifications, et l'autre à 256 pour montrer qu'une borne suffisamment grande permet à la représentation en liste simple de se comporter comme la représentation en liste de mappes. La représentation en liste de mappes est évaluée seule, puis accompagnée de la normalisation des bases et du partage des sous-termes.

La figure 6, en plus de donner un exemple où la simplification des ROW apporte un gain significatif à la résolution, montre les effets du choix de la représentation sur le temps de prétraitement pour une formule de très grande taille (45Mo, 151.000 déclarations, 84.000 assertions, 58.000 écritures, *293.000 lectures*) obtenue par exécution symbolique dynamique. Seul Boolector est utilisé pour cette évaluation car il est le seul à tenir la charge sans simplification. Le temps de résolution est alors d'environ *24 heures*. La représentation en liste de mappes réduit en *61 secondes* (ligne hachée horizontale) le nombre de lectures à *2.467*, toutes sur des tableaux initiaux. Le temps de résolution est alors de *14 secondes* (ligne pleine horizontale). La représentation en liste est évaluée avec une borne de remontée allant de 4.096 à 393.216 par incrément de 4.096. Elle parvient au même degré de simplification au-delà d'une borne de 385.024, mais en *53 minutes* environ. Le temps de résolution est alors le même.

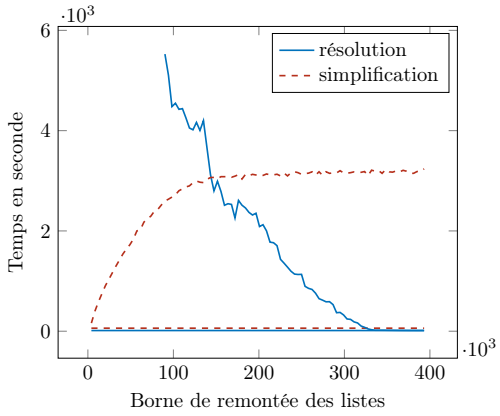


FIGURE 6 – Temps de simplification des ROW et de résolution par Boolector.

5 Conclusion et perspectives

Nous avons présenté LMNB, une technique originale de simplification des *read-over-write* pour la théorie des tableaux. Notre approche comble deux lacunes essentielles des méthodes existantes de simplification : le passage à l'échelle sur des formules de très grande taille pouvant apparaitre en vérification de programme, et la qualité de la simplification. Ces avantages ont été démontrés expérimentalement sur des formules réalistes, où notre prétraitement permet un gain clair de performance sur la phase de résolution ultérieure. La principale limitation actuelle provient de notre système de décision des égalités et inégalités entre indices. En effet, s'il fonctionne plutôt bien pour décider l'égalité, il n'est dans la plupart des cas pas capable de décider l'inégalité entre deux indices. Nous envisageons de prendre en compte des informations de type intervalle afin de pouvoir décider certaines inégalités portant sur des bases différentes.

5. http://smtcomp.sourceforge.net/2017/results-QF_ABV.shtml

Références

[1] A. R. Bradley and Z. Manna. *The Calculus of Computation : Decision Procedures with Applications to Verification (Chapters 5, 6 and 12)*. Springer, 2007.

[2] R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3) :165–201, 2009.

[3] C. Cadar and K. Sen. Symbolic execution for software testing : three decades later. *Commun. ACM*, 56(2) :82–90, 2013.

[4] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS, Barcelona, Spain, March 29 - April 2, 2004*, pages 168–176, 2004.

[5] R. David, S. Bardin, J. Feist, L. Mounier, M. Potet, T. D. Ta, and J. Marion. Specification of concretization and symbolization policies in symbolic execution. In *ISSTA, Saarbrücken, Germany, July 18-20, 2016*, pages 36–46, 2016.

[6] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis. In *SANER, Suita, Osaka, Japan, March 14-18, 2016*, pages 653–656, 2016.

[7] P. J. Downey and R. Sethi. Assignment commands with array references. *J.ACM*, 25(4) :652–666, 1978.

[8] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *CAV, Berlin, Germany, July 3-7, 2007*, pages 519–531, 2007.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART : directed automated random testing. In *PLDI, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

[10] D. Kroening and O. Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

Un mécanisme d'extraction vers C pour Why3

Raphaël Rieu-Helft[1,2]

[1] TrustInSoft
[2] Inria, Université Paris-Saclay, F-91120 Palaiseau

Résumé

Nous présentons un mécanisme d'extraction d'un sous-ensemble des programmes écrits avec l'outil de vérification Why3 vers le langage C. Un modèle mémoire partiel mais simple permet aux utilisateurs d'écrire des programmes Why3 dans un style impératif très proche du C, avec une gestion manuelle et vérifiée formellement de la mémoire. De tels programmes peuvent ensuite être extraits de manière directe vers du code C idiomatique, ce qui évite d'introduire des pertes de performances à l'extraction.

1 Introduction

L'objectif de ce travail est d'obtenir des programmes C à la fois efficaces et vérifiés formellement. L'idée principale est d'écrire le code dans un langage de haut niveau, à savoir le langage de programmation WhyML [6] fourni par la plateforme de vérification Why3, et de l'extraire vers C. WhyML est un dialecte de ML visant à faciliter la preuve automatique, notamment en faisant en sorte que tous les alias soient connus statiquement par typage [4]. WhyML est dédié à la vérification statique de comportements fonctionnels décrits à l'aide d'annotations fournies par l'utilisateur (assertions intermédiaires [8], code *ghost* [5]...) dans un langage de spécification expressif [2, 3]. Il s'agit ensuite de convertir un tel programme de haut niveau en code exécutable et performant. Le mécanisme d'*extraction* de Why3 traduit le code WhyML vers un langage de programmation classique en oubliant les spécifications et annotations destinées à la vérification. Why3 supporte nativement l'extraction vers OCaml, et nous y avons ajouté une extraction vers C. Pour obtenir du code C qui inclut des accès mémoire bas niveau à l'aide de pointeurs, il est nécessaire de mettre au point un modèle des pointeurs et du tas C en Why3, dans lequel les alias potentiels entre pointeurs sont contrôlés (Section 2). Nous avons conçu ce modèle mémoire pour permettre une compilation directe et transparente d'un sous-ensemble de WhyML vers C (Section 3). Nous présentons un exemple en Section 4.

Ce travail a déjà été publié dans le contexte du développement d'une bibliothèque d'arithmétique de grands entiers efficace et vérifiée, extraite de Why3 vers C. [9].

2 Modèle mémoire

Nous avons ajouté à la bibliothèque standard de Why3 un modèle mémoire du C dans lequel les opérations de base sur les pointeurs sont axiomatisées (figure 1, expliquée plus tard dans cette section). Au moment de l'extraction, ces opérations peuvent être simplement remplacées par leurs équivalents C (en commentaires sur la figure). Notre modèle est plus restrictif que le standard C, afin de simplifier le modèle et les preuves. Par exemple, l'arithmétique de pointeur est réduite à l'addition d'un pointeur et d'un entier, tandis que les soustractions et inégalités entre pointeurs ne sont pas permises. Le modèle ne prévoit pas non plus de conversion d'entier vers pointeur, et l'opérateur d'adresse `&` n'est pas présent.

La raison principale pour ces restrictions est le problème de l'aliasing entre pointeurs. Le tas mémoire C est vu comme un ensemble de blocs mémoire appelés *objets* dans le standard C99 [7].

```
1  type ptr 'a = { mutable data : array 'a ; offset : int }
2
3  function plength (p:ptr 'a) : int = p.data.length
4
5  function pelts (p:ptr 'a) : (int → 'a) = p.data.elts
6
7  val malloc (sz:uint32) : ptr 'a              (*  malloc(sz * sizeof('a))   *)
8    requires { sz > 0 }
9    ensures  { plength result = sz ∨ plength result = 0 }
10   ensures  { result.offset = 0 }
11
12 val free (p:ptr 'a) : unit                              (*         free(p)    *)
13   requires { p.offset = 0 }
14   writes   { p.data }
15   ensures  { plength p = 0 }
16
17 predicate valid (p:ptr 'a) (sz:int) =
18   0 ≤ sz ∧ 0 ≤ p.offset ∧ p.offset + sz ≤ plength p
19
20 val get (p:ptr 'a) : 'a                                 (*              *p    *)
21   requires { 0 ≤ p.offset < plength p }
22   ensures  { result = p.data[p.offset] }
23
24 val set (p:ptr 'a) (v:'a) : unit                         (*          *p = v    *)
25   requires { 0 ≤ p.offset < plength p }
26   writes   { p.data.elts }
27   ensures  { pelts p = Map.set (pelts (old p)) p.offset v }
28
29 val incr (p:ptr 'a) (ofs:int32) : ptr 'a                 (*           p+ofs    *)
30   requires { p.offset + ofs ≤ plength p }
31   alias    { p.data ~ result.data }
32   ensures  { result.offset = p.offset + ofs }
33   ensures  { result.data = p.data }
34
35 val get_ofs (p:ptr 'a) (ofs:int32) : 'a                  (*         *(p+ofs)   *)
36   requires { 0 ≤ p.offset + ofs < plength p }
37   ensures  { result = p.data[p.offset + ofs] }
38
39 val set_ofs (p:ptr 'a) (ofs:int32) (v:'a) : unit         (*     *(p+ofs) = v   *)
40   requires { 0 ≤ p.offset + Int32.to_int ofs < plength p }
41   ensures { pelts p = Map.set (pelts (old p)) (p.offset + Int32.to_int ofs) v }
42   writes { p.data.elts }
```

FIGURE 1 – Un modèle des pointeurs et du tas mémoire du C.

Deux pointeurs sont dits *alias* l'un de l'autre lorsqu'ils pointent vers le même objet mémoire (y compris avec des décalages différents), c'est-à-dire qu'une écriture via l'un affecte les valeurs lues via l'autre. La connaissance des relations d'aliasing entre pointeurs est cruciale pour la preuve. Notre modèle restreint permet de profiter du système de types de Why3 [4] pour connaître statiquement les alias présents entre pointeurs. Un modèle plus général, utilisant par exemple un tas explicite, serait envisageable mais ne permettrait pas de s'appuyer sur le système de types. Il faudrait alors exprimer de nombreuses hypothèses d'aliasing ou de non-aliasing entre pointeurs dans les préconditions de fonctions, ce qui compliquerait énormément le travail de preuve même pour des programmes simples.

Le type polymorphe WhyML `ptr 'a` (figure 1 ligne 1) représente les pointeurs vers des blocs contenant des données de type `'a`. Le champ `data` d'un pointeur est un tableau contenant

les données du bloc, et le champ `offset` indique vers quelle case du tableau il pointe. Cette construction supporte les alias entre pointeurs : différents pointeurs peuvent référencer le même tableau (donc pointer vers le même bloc en mémoire). Grâce au système de types de Why3, les alias sont connus statiquement et une assignation à travers un pointeur est propagée à ses alias.

Les pointeurs sont alloués avec la fonction `malloc`. En cas d'échec, celle-ci renvoie un pointeur *invalide*, représenté par un bloc de longueur 0. La fonction `free` invalide son argument en remplaçant la longueur du bloc pointé par 0. Un pointeur est dit *valide pour une taille* t (ligne 17) lorsque son offset plus t n'excède pas la longueur du bloc pointé. La fonction `get` (ligne 20) représente le déréferencement de pointeur en lecture. La fonction `set` représente l'assignation en mémoire ; la clause `writes` spécifie l'effet d'écriture sur le bloc.

La fonction `incr` (ligne 29) renvoie la somme d'un pointeur et d'un entier. Conformément au standard C [7, Section 6.5.6, "Additive Operators"], on peut seulement calculer un pointeur vers l'intérieur d'un bloc valide ou vers l'élément juste après un bloc valide, ce qui est reflété dans la précondition.

Par défaut, dans une signature de fonction WhyML, toutes les régions des arguments et de la valeur de retour sont supposées non aliasées deux à deux. Ce n'est pas le cas pour l'incrémentation de pointeur en C (le résultat est un alias du pointeur passé en argument). Dans le cadre de ce travail, nous avons ajouté à Why3 le mot-clé `alias`, qui sert à déclarer un alias entre une région d'un argument et une région présente dans la valeur de retour d'une fonction en unifiant les types qui les représentent.

Ainsi, la déclaration `alias` dans la signature de `incr` déclare que le pointeur renvoyé par `incr` est un alias du pointeur passé en argument en unifiant les types représentant les régions `p.data` et `result.data`. Les régions sont ainsi unifiées à la fois du point de vue du contenu du bloc pointé et du point de vue du comportement de `free` (l'appeler sur un pointeur invalide aussi tous ses alias).

Ceci permet d'écrire une spécification particulièrement courte pour `free` : son effet `writes` sur `p.data` y induit un effet dit de *reset* [4], ce qui signifie que la région précédemment pointée n'est plus accessible par aucun de ses alias, qui sont donc invalidés.

3 Extraction vers du C idiomatique

L'objectif de la procédure d'extraction est de générer du code C efficace. Certaines fonctionnalités du langage WhyML, comme les types algébriques ou les fonctions d'ordre supérieur, sont difficiles à traduire en C sans introduire de constructions complexes (clôtures, allocation de mémoire automatique, ramasse-miettes...) qui nuiraient aux performances du code extrait. Nous avons donc décidé de ne supporter qu'un sous-ensemble de WhyML. Il ne s'agit donc pas d'extraire des programmes WhyML arbitraires vers C, mais d'extraire des programmes WhyML écrits dans un style proche du C. Nous permettons donc à notre procédure d'extraction d'échouer, ce qui arrive quand le programme à extraire contient des fonctionnalités WhyML non supportées. Les fonctionnalités de WhyML supportées sont celles qui peuvent être traduites vers C de manière transparente, comme les boucles et les références. En abandonnant un certain nombre de fonctionnalités du langage, on gagne une grande simplicité dans la procédure d'extraction, et le code extrait ressemble au code WhyML d'origine. Le programmeur WhyML peut donc prédire à quoi ressemblera le programme C extrait, et peut donc obtenir plus facilement du code C efficace. Cette transparence permet aussi de faire davantage confiance au code extrait et à la procédure d'extraction, qui n'est pas vérifiée formellement.

Compilation d'exceptions en instructions `break` ou `return`. Si l'extraction n'a pas vocation à supporter toutes les fonctionnalités de WhyML, il est en revanche souhaitable qu'un maximum de fonctionnalités du C soient exprimables dans le sous-ensemble de WhyML traité. En particulier, on veut pouvoir obtenir des programmes C comportant des instructions `break` ou `return`. Dans WhyML, ces structures de contrôle sont exprimées à l'aide d'exceptions. On cherche donc à détecter les exceptions WhyML qui peuvent être extraites directement vers `break` ou `return` en C. Les autres exceptions sont rejetées.

Dans le cas de `break`, on détecte le motif suivant et on extrait toutes les instances de `raise B` dans le corps de la boucle (mais pas dans d'éventuelles boucles imbriquées) par `break`.

```
try while ... do ... raise B ... done with B → () end
```

Similairement, pour `return`, on détecte le motif suivant dans les définitions de fonctions et on extrait toutes les instances de `raise (R e)` par `return e`. À noter que la construction `try with` doit être en position terminale dans le corps de la fonction.

```
let f (args) = ... ; try ... raise (R e) ... with R v → v end
```

Notre procédure d'extraction reconnaît ces motifs indépendamment des noms des exceptions utilisées. Toutes les instances de `try with` ou `raise` qui ne correspondent pas à l'un des motifs sont rejetées par l'extraction.

Valeurs de retour multiples. De nombreuses fonctions WhyML renvoient plusieurs valeurs sous forme d'un n-uplet. Ceci n'a pas d'équivalent natif en C. Nous avons choisi d'extraire chaque fonction renvoyant un n-uplet en une fonction C renvoyant `void` et prenant comme arguments supplémentaires un pointeur par élément du n-uplet. On peut ensuite détecter l'appel

```
let (x1, x2, ...) = f(args) in ...
```

et l'extraire comme

```
f(&x1, &x2, ..., args); ...
```

La fonction `add_with_carry` (figure 3) est un exemple d'appel d'une telle fonction.

4 Un exemple : addition de grands entiers

Le type Why3 `uint64` (défini en figure 2) est un type abstrait, équipé d'une projection `to_int` qui associe à un entier machine sa valeur mathématique. À l'extraction, ils sont traduits directement vers le type C `uint64_t`. Les préconditions des opérations arithmétiques élémentaires sur ces entiers (comme `add` sur la figure) maintiennent l'invariant que cette valeur est comprise entre 0 et $2^{64} - 1$. Un entier de taille arbitraire est représenté par un tableau d'`uint64`, et sa valeur mathématique est donnée par la fonction logique `value`.

La figure 3 contient une fonction WhyML tirée de notre bibliothèque de calcul en précision arbitraire [9]. Elle effectue l'addition de deux entiers de taille arbitraire. Par souci de concision, les annotations de preuve ont été omises.

L'algorithme est celui qui est enseigné à l'école : on ajoute les "chiffres" (ici en base 2^{64}) les moins significatifs des deux opérandes, on maintient une retenue, on passe à l'étape suivante.

Il y a trois boucles pour prendre en compte le cas où les deux opérandes n'ont pas la même taille. Lorsque tous les chiffres du nombre le plus court ont été pris en compte (première boucle `while`), on termine de propager la retenue (deuxième boucle), puis on recopie les chiffres restants du nombre le plus long (troisième boucle).

```
type uint64
val function to_int (n:uint64) : int
meta coercion function to_int
predicate in_bounds (n:int) = 0 ≤ n ≤ 0xffff_ffff_ffff_ffff
axiom to_int_in_bounds: forall n:uint64. in_bounds (to_int n)

val add (x y:uint64) : uint64
  requires { in_bounds (to_int x + to_int y)  }
  ensures  { to_int result = to_int x + to_int y }
```

FIGURE 2 – Extrait de la spécification Why3 des entiers machine.

```
let wmpn_add (r x y:ptr uint64) (sx sy:int32) : uint64
    requires { 0 ≤ sy ≤ sx }
    requires { valid x sx }
    requires { valid y sy }
    requires { valid r sx }
    ensures { value r sx + (power radix sx) * result =
              value x sx + value y sy }
    ensures { 0 ≤ result ≤ 1 }
    writes { r.data.elts }
 =
    let zero = UInt64.of_int 0 in
    let lx = ref zero in
    let ly = ref zero in
    let c = ref zero in
    let i = ref (Int32.of_int 0) in
    while Int32.(<) !i sy do
      lx := get_ofs x !i;
      ly := get_ofs y !i;
      let res, carry = add_with_carry !lx !ly !c in
      set_ofs r !i res;
      c := carry;
      i := Int32.(+) !i (Int32.of_int 1);
    done;
    try
    while Int32.(<) !i sx do
      if (UInt64.(=) !c zero) then raise Break;
      lx := get_ofs x !i;
      let res, carry = add_with_carry !lx zero !c in
      set_ofs r !i res;
      c := carry;
      i := Int32.(+) !i (Int32.of_int 1);
    done
    with Break → assert { !c = 0 }
    end;
    while Int32.(<) !i sx do
      lx := get_ofs x !i;
      set_ofs r !i !lx;
      i := Int32.(+) !i (Int32.of_int 1);
    done;
    !c
```

```
uint64_t wmpn_add(uint64_t * r4,
    uint64_t * x5, uint64_t * y3,
    int32_t sx, int32_t sy)
{
  uint64_t lx4, ly2, c2;
  uint64_t res2, carry2, res3, carry3;
  int32_t i6, o9, o10, o11;
  lx4 = (0ULL); ly2 = (0ULL);
  c2 = (0ULL); i6 = (0);
  while (i6 < sy)
    {
      lx4 = *(x5+(i6));
      ly2 = *(y3+(i6));
      (add64_with_carry)(&res2, &carry2,
          lx4, ly2, c2);
      *(r4+(i6)) = res2;
      c2 = carry2;
      o9 = (i6 + 1);
      i6 = o9;
    }
  while (i6 < sx)
    {
      if(c2 == 0ULL)
        break;
      lx4 = *(x5+(i6));
      (add64_with_carry)(&res3, &carry3,
          lx4, 0ULL, c2);
      *(r4+(i6)) = res3;
      c2 = carry3;
      o10 = (i6 + 1);
      i6 = o10;
    }
  while (i6 < sx)
    {
      lx4 = *(x5+(i6));
      *(r4+(i6)) = lx4;
      o11 = (i6 + 1);
      i6 = o11;
    }
  return (c2);
}
```

FIGURE 3 – Fonction d'addition de grands entiers et code extrait correspondant.

L'implémentation est écrite dans un style impératif proche du C, et utilise des exceptions pour simuler `break`. Une fonction renvoyant deux valeurs, la primitive `add_with_carry`, est utilisée. Elle additionne deux entiers machine et une retenue entrante et renvoie la somme sous forme de deux entiers machine : la somme modulo 2^{64} et la retenue sortante.

Le code extrait (figure 3, à droite) ressemble fortement au code WhyML. Les types natifs du C (entiers machine, pointeurs) sont utilisés directement. Les variables introduites dans le corps de la fonction WhyML par des constructions `let ... in` sont toutes déclarées au début de la fonction C extraite. L'absence de conflits de noms est garantie par la syntaxe interne de Why3.

Le fait que le programme ait été écrit en WhyML puis extrait vers C n'ajoute pas de complexité significative par rapport à un programme écrit directement en C. Si l'on écrivait ce programme directement en C, la différence principale serait l'élimination des variables intermédiaires `o9`, `o10`, `o11`, `res2`, `res3`, `carry2`, `carry3` (qui sont des artefacts de la syntaxe interne de Why3) ainsi que `i6`, `lx4` et `ly2` (qui sont présentes dans le code WhyML pour faciliter la preuve). Cependant, cette optimisation est largement à la portée du compilateur C, il n'y a donc pas de perte de performance. Nous prenons le parti de ne pas trop optimiser le programme à l'extraction, afin de garder le mécanisme (non formellement vérifié) aussi simple et transparent que possible. L'enjeu principal est d'éviter les constructions et structures de contrôle complexes, qui pourraient être mal optimisées même par un bon compilateur.

5 Conclusion

L'objectif de cette procédure d'extraction est de développer des programmes C efficaces et vérifiés formellement. Nous avons déjà pu nous en servir pour obtenir une bibliothèque d'arithmétique en précision arbitraire [1] offrant les mêmes primitives et des performances similaires à celles de GMP [2] pour les entiers de taille inférieure à 1000 bits [9].

Notre modèle mémoire et la notion de pointeurs qu'il fournit nous ont permis d'écrire les fonctions WhyML de notre bibliothèque en imitant autant que possible l'implémentation de GMP, et de développer une procédure d'extraction simple de Why3 vers C. De plus, comme le modèle mémoire s'appuie sur le mécanisme de contrôle d'alias de Why3, les spécifications et les preuves de nos fonctions ne concernent que leurs propriétés arithmétiques, sans s'embarrasser de conditions sur les alias.

Une prochaine étape de ce travail pourrait consister à extraire non seulement le code des fonctions vers C, mais aussi leurs spécifications (par exemple vers le langage de spécifications ACSL[1]), afin de ne plus devoir inclure l'extraction dans la base de confiance. Le code C extrait pourrait ensuite être vérifié à l'aide d'outils existants de vérification de code C en tirant parti du fait que les algorithmes ont déjà été prouvés à l'aide de Why3.

Remerciements Je remercie Pascal Cuoq et Guillaume Melquiond pour leurs remarques et suggestions.

1. Preuve et instructions d'installation disponibles à l'adresse http://toccata.lri.fr/gallery/multiprecision.en.html
2. http://gmplib.org/

Références

[1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL : ANSI/ISO C specification language, 2008. http://frama-c.com/acsl.html.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and 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.

[3] Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013. Springer.

[4] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Research report, Université Paris Sud, 2016. https://hal.archives-ouvertes.fr/hal-01256434v3.

[5] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3) :152–174, 2016.

[6] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[7] International Organization for Standardization. *ISO/IEC 9899 :1999 : Programming Languages – C*, 2000.

[8] K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop*, Redmond, WA, USA, November 2010.

[9] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to Get an Efficient yet Verified Arbitrary-Precision Integer Library. In *9th Working Conference on Verified Software : Theories, Tools, and Experiments*, Heidelberg, Germany, July 2017.

Vérification automatique de chaînes de fonctions de sécurité dans des réseaux programmables SDN avec SYNAPTIC

Nicolas Schnepf, Rémi Badonnel, Abdelkader Lahmadi, Stephan Merz

Université de Lorraine, CNRS, Inria, Loria, F-54000 Nancy

Résumé

La protection des équipements intelligents tels que smartphones devient un point sensible du fait de la popularité de ces plateformes et des contraintes qu'elles induisent. Pour assurer cette protection, nous proposons des méthodes de vérification automatique de chaînes de sécurité déployées en environnements cloud. En particulier, nous définissons des traductions du langage PYRETIC dédié à la description de chaînes de sécurité vers les langages SMTlib et nuXmv afin d'utiliser les outils associés pour vérifier des propriétés capturant le comportement attendu d'une chaîne en termes de sécurité. Nous décrivons également un ensemble de résultats préliminaires obtenus dans ce cadre, en particulier une évaluation du temps de réponse et de l'espace mémoire requis par différentes méthodes formelles pour notre procédure de vérification.

1 Contexte

La sécurité des réseaux modernes repose sur des fonctions de sécurité tels que les pare-feux, les systèmes de détection d'intrusion ou les systèmes de prévention de fuite de données ; ces fonctions de sécurité peuvent être virtualisées et exposées dans le réseau de sorte à automatiser et externaliser leur déploiement dans des infrastructures cloud. Ces fonctions peuvent ainsi être composées dynamiquement de sorte à former des chaînes SFC (*Security Function Chaining*) tels que présentés dans [8]. L'idée de ces chaînages est qu'ainsi le trafic réseau sera analysé par chacune des fonctions qui le composent, on peut par exemple imaginer un chaînage composé d'un pare-feu en entrée suivi de deux systèmes de détection d'intrusion branchés en parallèle suivis enfin d'un système de prévention de fuite de données.

La programmabilité offertes par les réseaux SDN (*Software-Defined Networks*) facilite la construction de ces chaînes de sécurité et permet d'automatiser leur déploiement, leur activation et leur configuration en fonction du contexte [6]. En effet, l'architecture SDN repose sur le découplage du plan des données et du plan de contrôle : le plan des données est le plus souvent implanté au moyen de switchs programmables mais de nombreux travaux visent à y intégrer des fonctions de sécurité plus avancées telles que celles que nous avons présentées au paragraphe précédent ; le plan de contrôle est quant à lui représenté par un composant central nommé le contrôleur dont le rôle est de gérer la configuration du plan des données en fonction d'évènements réseau tels que l'ouverture d'une connexion. La communication entre les deux plans se fait au moyen d'un protocole dédié, typiquement le protocole OpenFlow [11] qui permet au contrôleur de propager les règles de configuration des switchs programmables mais également de collecter des informations statistiques sur leur exécution tel que par exemple la fréquence d'activation d'une règle donnée.

Néanmoins, la multiplication et la complexité des chaînes avec leurs fonctions de sécurité pour protéger un ou plusieurs équipements risquent d'introduire des erreurs de configuration. L'un des principaux défis consiste donc à vérifier automatiquement ces chaînes par application de méthodes formelles de fonctions de sécurité de sorte à s'assurer qu'elles garantissent bien les propriétés de sécurité que l'on attend d'elles.

Il existe déjà un certain nombre d'approches, notamment [1, 2, 9], traitant de la vérification de chaînes de services réseaux. Cependant, ces travaux se concentrent sur la vérification soit du plan de contrôle, soit du plan de données du paradigme SDN et ne couvrent donc pas l'intégralité des configurations réseau, par exemple dans le cas où l'un des équipements du plan des données émet une alerte entraînant la mise à jour du plan de contrôle. Ce manque pourrait être comblé par la conception et l'implantation d'une procédure de vérification couvrant ces deux plans à la fois. L'optimisation des performances d'une telle procédure demande notamment une attention toute particulière pour répondre dynamiquement aux comportements malveillants au sein du réseau : en effet, la complexité des techniques de vérification automatique en terme de temps de réponse et de consommation d'espace mémoire peut être prohibitive si nous considérons des scénarios où les chaînes seraient générées et vérifiées dynamiquement pendant l'exécution du système.

2 Vérification formelle de chaînes de fonctions de sécurité

Dans ce contexte, nous proposons une solution pour la vérification automatique de chaînes de fonctions de sécurité que nous avons nommée SYNAPTIC.

2.1 Le checker synaptic

Notre solution se base sur un langage de programmation de réseaux SDN permettant la spécification des chaînes à un haut niveau d'abstraction qui peut ensuite être traduit vers OpenFlow. Ce langage PYRETIC [7] permet de spécifier des politiques SDN grâce à la grammaire suivante.

$\langle policy \rangle ::= \langle unit \rangle$
$\quad | \quad \langle modify \rangle$
$\quad | \quad \langle sequence \rangle$
$\quad | \quad \langle parallel \rangle$

$\langle unit \rangle ::= \mathsf{identity}$
$\quad | \quad \mathsf{drop}$
$\quad | \quad \mathsf{match}(x_1 = y_1 \ldots x_n = y_n)$
$\quad | \quad \sim \langle unit \rangle$

$\langle modify \rangle ::= \mathsf{modify}(x_1 = y_1 \ldots x_n = y_n)$

$\langle sequence \rangle ::= \langle policy \rangle \gg \langle policy \rangle$

$\langle parallel \rangle ::= \langle policy \rangle + \langle policy \rangle$

La règle unitaire identity transfère tout le trafic ; drop supprime tous les paquets ; match transfère seulement les paquets dont les champs vérifient certaines conditions. L'opérateur $\sim$ exprime la négation de ces conditions. Le constructeur modify transfère tout le trafic en y apposant une modification. Des chaînes plus élaborées peuvent être construites au moyen des opérateurs de composition séquentielle $\gg$ et parallèle $+$.

À titre d'exemple, la politique donnée ci-dessous permettra le transfert de tout le trafic destiné aux ports 4000 et 5000 :

match(dstport=4000) + match(dstport=5000)

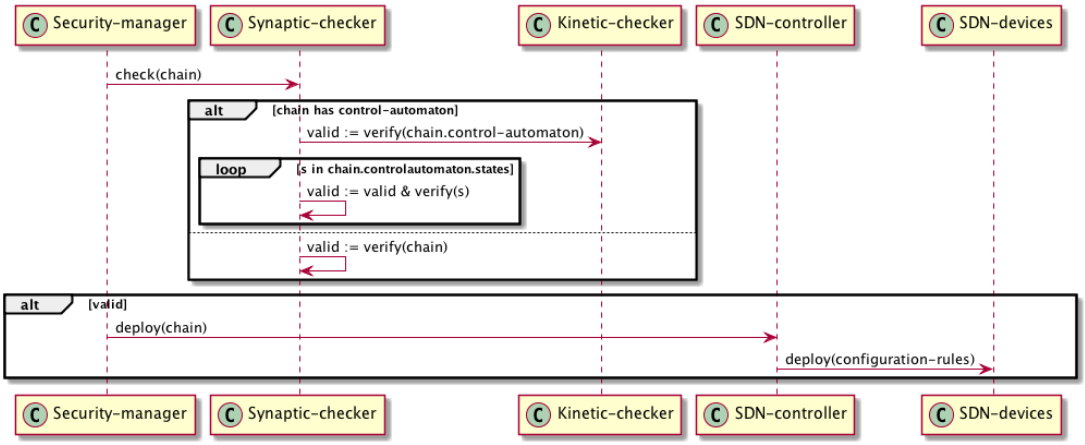


FIGURE 1 – Diagramme d'interactions entre les composants de notre architecture.

A contrario, la politique suivante supprimera tout le trafic sur le port 5000 :

$$\mathsf{match}(\text{dstport}{=}5000) \gg \mathsf{drop}$$

En utilisant l'opérateur de négation $\sim$, cette dernière politique peut être simplifiée de la manière suivante :

$$\sim \mathsf{match}(\text{dstport}{=}5000)$$

PYRETIC est également étendu par un langage nommé KINETIC [10] permettant de spécifier des politiques du plan de contrôle sous la forme d'automates à états finis. Ces automates peuvent ensuite être vérifiés par application de méthodes de model checking. C'est en complément à cette approche que nous proposons notre checker SYNAPTIC [12] qui utilise le formalisme introduit par PYRETIC et KINETIC pour spécifier des politiques SDN mais qui couvre tant le plan de données que le plan de contrôle. Pour étendre la vérification du plan de contrôle apportée par KINETIC, nous définissons des algorithmes de traduction permettant de générer des modèles formels des chaînes de règles PYRETIC pouvant être vérifiées à l'aide de SMT solving ou de model checking. Pour compléter ces modèles nous avons introduit dans SYNAPTIC un langage permettant de spécifier les propriétés devant être validées par les chaînes ainsi que les règles de traduction à y appliquer. Le comportement de notre checker est présenté sur la figure 1.

2.2 Vérification des chaînes par SMT solving

La génération d'un modèle SMT-LIB des politiques SDN spécifiées avec PYRETIC se fait en les exprimant sous la forme d'expressions logiques. Cette conversion est obtenue de manière intuitive en appliquant la fonction de réécriture f : PYRETIC $\rightarrow$ SMT-LIB définie dans la figure 2 dans lesquelles p, p_1 et p_2 désignent des variables contenant des politiques PYRETIC.

Le modèle SMT-LIB ainsi construit doit encore être complété par des conditions contraignant l'ensemble des flux acceptés par une politique réseau : en ce sens nous avons développé un langage étendant PYRETIC avec la possibilité d'exprimer des propriétés du plan de données telles que $dstport = 5000$ spécifiant que le trafic doit être autorisé sur le port 5000 ou bien $dstip \neq 192.16.0.0/16$ qui spécifie que le trafic doit être bloqué pour le préfixe réseau 192.16.0.0/16. Ces propriétés sont compilées en SMT-LIB avec la politique PYRETIC qui est

$$
\begin{array}{rcl}
f\ \mathsf{identity} & = & \mathsf{true} \\
f\ \mathsf{drop} & = & \mathsf{false} \\
f\ (\mathsf{match}(x_1 = y_1 \ldots x_n = y_n)) & = & (\mathsf{and}\ (=\ x_1\ y_1)\ \ldots\ (=\ x_n\ y_n)) \\
f\ (\sim p) & = & (\mathsf{not}\ (f\ p_1)) \\
f\ (\mathsf{modify}(x_1 = y_1 \ldots x_n = y_n)) & = & (\mathsf{and}\ (=\ x'_1\ y_1)\ \ldots\ (=\ x'_n y_n)) \\
f\ (p_1 \gg p_2) & = & (\mathsf{and}\ (f\ p_1)\ (f\ p_2)) \\
f\ (p_1 + p_2) & = & (\mathsf{or}\ (f\ p_1)\ (f\ p_2))
\end{array}
$$

FIGURE 2 – Traduction de règles PYRETIC en SMT-LIB.

censée les garantir, après quoi on peut faire appel à un SMT solver. Dans notre checker SYNAPTIC, nous avons intégré cvc4 [3] et veriT [4]. Le solveur cherche à vérifier si la chaîne valide les propriétés, c'est-à-dire que la politique PYRETIC accepte effectivement le trafic autorisé par un opérateur et rejette celui qui aura été interdit.

2.3 Vérification des chaînes par model checking

L'application des techniques de model checking symbolique disponibles dans nuXmv [5] repose sur la génération d'un automate représentant le comportement d'une chaîne décrite en PYRETIC. Notre fonction de traduction g : PYRETIC $\rightarrow$ NUXMV calcule ainsi la liste des transitions décrivant un automate, chaque transition étant un quadruplet composé d'un état source, d'une garde, d'une action sur les variables et d'un état cible. La définition de cette fonction est donnée dans la figure 3 où h calcule l'ensemble des transitions correspondant à une politique PYRETIC, un état source et un état destination et la fonction *state* génère un nouvel état.

$$
\begin{array}{rcl}
g\ p_1 & = & h\ p_1\ (state())\ (state()) \\
h\ \mathsf{identity}\ s\ d & = & [(s, \mathsf{true}, \mathsf{true}, d)] \\
h\ \mathsf{drop}\ s\ d & = & [(s, \mathsf{false}, \mathsf{true}, d)] \\
h\ (\mathsf{match}(x_1 = y_1 \ldots x_n = y_n))\ s\ d & = & [(s, x_1 = y_1 \wedge \ldots \wedge x_n = y_n, \mathsf{true}, d)] \\
h\ (\sim p)\ s\ d & = & \mathsf{let}\ [(\_, c, \_, \_)] = h\ p\ s\ d\ \mathsf{in}\ [(s, \neg c, \mathsf{true}, d] \\
h\ (\mathsf{modify}(x_1 = y_1 \ldots x_n = y_n))\ s\ d & = & [s, \mathsf{true}, x'_1 = y_1 \wedge \ldots \wedge x'_n = y_n, d] \\
h\ (p_1 \gg p_2)\ s\ d & = & \mathsf{let}\ s' = state()\ \mathsf{in}\ (h\ p_1\ s\ s')@(h\ p_2\ s'\ d) \\
h\ (p_1 + p_2)\ s\ d & = & (h\ p_1\ s\ d)@(h\ p_2\ s\ d)
\end{array}
$$

FIGURE 3 – Génération d'automates à partir de règles PYRETIC.

Pour illustrer cette procédure, considérons l'exemple de la chaîne de pare-feux spécifiée dans le listing 1. L'automate correspondant à cette chaîne, obtenu en appliquant notre procédure de traduction, est présenté dans la figure 4.

```
F1 = match(srcip=IP("198.122.37.15"))
F2 = match(srcport=1000) + match(srcport=2000) + match(srcport=3000)
F3 = match(srcport=4000) + match(srcport=5000) + match(srcport=6000)
F4 = match(dstport=7000) + match(dstport=8000) + match(dstport=9000)

chain = ((F1 >> F2)+(~F1 >> F3)) >> F4
```

Listing 1 – Illustration d'une spécification en PYRETIC d'une chaîne de sécurité.

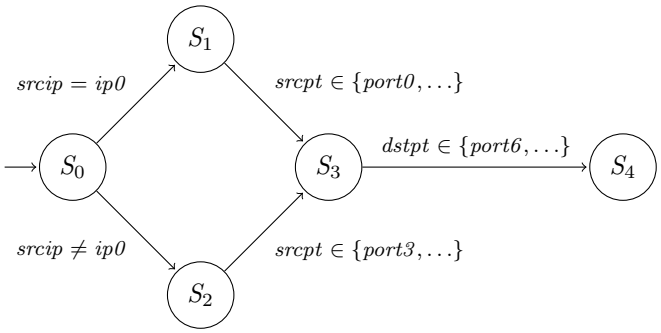


FIGURE 4 – Automate correspondant à la chaîne du listing 1.

Les propriétés contraignant la politique sont traduites en logique CTL : la procédure de vérification déclare succès si tous les flux autorisés sont acceptés par l'automate et si tous les flux interdits sont rejetés. Dans notre checker SYNAPTIC, nous avons intégré le model checker nuXmv [5] pour la validation du plan de données. En effet le checker NuSMV était déjà utilisé par KINETIC, et nous avons choisi son extension nuXmv en raison de la possibilité de représenter les champs d'entête par des variables avec un domaine potentiellement infini et de prendre en compte les contraintes d'égalité.

3 Evaluation de performances

Pour évaluer les performances de ces différentes méthodes de vérification en terme de temps de réponse et de consommation mémoire, nous avons défini plusieurs critères caractérisant la taille de la chaîne reçue en entrée. En particulier, nous avons évalué l'impact du nombre de configurations possibles pour l'automate KINETIC d'une même chaîne, la taille d'une chaîne en termes de longueur et de largeur, c'est-à-dire respectivement le nombre de règles composées en séquence et en parallèle, et enfin le nombre de propriétés à vérifier avec la chaîne. Pour chacun de ces paramètres, nous avons implanté un module python permettant la génération synthétique des différentes chaînes de sécurité. Les résultats sont reportés dans les tableaux 1 et 2 où la seconde colonne indique la valeur maximale de chaque paramètre et où les colonnes suivantes donnent le temps de réponse (en secondes), respectivement la consommation mémoire (en MO) des différents algorithmes de vérification. [1]

Critère	Max	CVC4	veriT	nuXmv
Nombre de configurations	100	1.175	0.769	3.475
Largeur de la chaîne	500	1.721	3.987	0.250
Longueur de la chaîne	100	—	0.26	0.13

TABLE 1 – Temps de réponse en secondes observés avec différents outils de vérification.

Ces résultats indiquent clairement que nuXmv fournit de meilleurs résultats que veriT et CVC4 pour des chaînes importantes en longueur et largeur, même si les solveurs SMT ont un léger avantage en ne considérant que le nombre de configurations. Les résultats obtenus lorsque

1. L'entrée "—" indique un dépassement du temps maximal de 90 secondes pour cette expérience.

Critère	Max	CVC4	veriT	nuXmv
Nombre de configurations	100	28.027	28.027	28.027
Largeur de la chaîne	500	202.678	62816.808	29.894
Longueur de la chaîne	100	—	126.815	25.952

TABLE 2 – Consommations mémoire observées avec différents outils de vérification.

nous avons évalué l'influence du nombre de propriétés, non présentés ici, confirment que nuXmv apporte de meilleurs résultats que CVC4 et veriT en terme de temps de réponse. De manière générale, nuXmv semble être le meilleur candidat pour la vérification de chaînes de fonctions de sécurité à la volée, au cours de ces expérimentations. Néanmoins, notre stratégie requiert encore d'être optimisée dans le cas où il faudrait vérifier des chaînes caractérisées par un nombre élevé de configurations. Observons également que les résultats présentés dans cette section ont été obtenus à partir de politiques générées de manière synthétique et qu'ils demandent à être confirmés à l'aide d'exemples de politiques générées à partir de traces réseau caractérisant le comportement des utilisateurs.

4 Conclusions

Nous avons présenté les grandes lignes de notre travail concernant la vérification de chaînes de fonctions de sécurité pour la protection des environnements intelligents. En particulier nous avons présenté les algorithmes de traduction d'une politique de sécurité PYRETIC dans deux formalismes différents, vérifiables par application de SMT solving et de model checking. Nous avons implanté ces algorithmes comme une extension au langage PYRETIC que nous avons nommée SYNAPTIC et au sein de laquelle nous avons intégré les SMT solvers cvc4 et veriT ainsi que le model checker nuXmv. Enfin nous avons procédé à l'évaluation des performances de ces différentes méthodes en termes de temps de réponse et de consommation mémoire de sorte à identifier la plus adaptée pour assurer la vérification en temps réel.

Au sortir de cette phase d'évaluation c'est le model checker nuXmv qui apparaît comme le meilleur candidat pour l'implantation d'un module de vérification temps réel ; néanmoins cette approche demande encore à être améliorée au vu des performances qu'elle apporte dans le cas de chaînes de fonctions de sécurité comportant de nombreux états de configuration. Parmi nos travaux futurs, nous pouvons notamment mentionner l'inférence automatique des modèles du trafic réseau des applications utilisateurs en vue de déterminer les chaînes de sécurité les plus adaptées, ainsi que l'affinement de notre modèle des chaînes pour améliorer les performances de vérification.

Références

[1] Ehab Al-Shaer and Saeed Al-Haj. FlowChecker, Configuration Analysis and Verification of Federated OpenFlow Infrastructures. In *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration (CCS'10)*, 2010.

[2] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. Vericon : Towards Verifying Controller Programs in Software-Defined Networks. In *Proc. 35th ACM SIGPLAN Intl. Conf. Programming Language Design (PLDI'14)*, pages 282–293, Edinburgh, UK, 2014.

[3] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proc. 23rd Intl. Conf. Computer Aided Verification (CAV 2011)*, pages 171–177, Snowbird, UT, USA, 2011.

[4] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT : An Open, Trustable and Efficient SMT-Solver. In *Proc. 22nd International Conference on Automated Deduction (CADE-22)*, pages 151–156, Montreal, Canada, 2009.

[5] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *Proc. 26th Intl. Conf. Computer Aided Verification (CAV 2014)*, pages 334–342, Vienna, Austria, 2014.

[6] Nick Feamster, Jennifer Rexford, and Ellen Zegura. The Road to SDN, an Intellectual History of Programmable Networks. *SIGCOMM Computer Communication Review*, 44(2) :87–98, 2014.

[7] Nate Foster, Michael J. Freedman, Arjun Guha, Rob Harrison, Naga Praveen Kata, Christopher Monsanto, Joshua Reich, Mark Reitblatt, Rexford Jennifer, Cole Schlesinger, Alec Story, and David Walker. Languages for Software-Defined Networks. In *Software Technology Group*, 2016.

[8] Gaëtan Hurel, Rémi Badonnel, Abdelkader Lahmadi, and Olivier Festor. Behavioral and Dynamic Security Functions Chaining for Android Devices. In *Proceedings of the 11th IFIP/IEEE/ACM SIGCOMM International Conference on Network and Service Management (CNSM'15)*, 2015.

[9] Ahmed Khurshid, Xuan Zou, Winxuan Zhou, Matthew Caesar, and P. Brighten. VeriFlow : Verifying Network-wide Invariants in Real Time. In *Proceedings of the first Workshop on Hot Topics in Software-Defined Networks (HotSDN'12)*, 2012.

[10] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhamad Shahbaz, Nick Feamster, and Russ Clark. Kinetic : Verifiable Dynamic Network Control. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation (NSDI'15)*, 2015.

[11] Nick McKeown and Guru Parulkar. openflow, enabling innovation in campus networks. In *ACM SIGCOMM Computer Communication Review*, 2008.

[12] Nicolas Schnepf, Stephan Merz, Rémi Badonnel, and Abdelkader Lahmadi. Automated verification of security chains in software-defined networks with Synaptic. In *Proceedings of the 3rd IEEE Conference on Network Softwarization (NetSoft'17)*, 2017.

Why3 a dit : gardez le contrôle en toute situation

Jean-Christophe Léchenet[12], Nikolai Kosmatov[1], and Pascale Le Gall[2]

[1] CEA, LIST, Laboratoire de Sûreté des Logiciels
PC 174, 91191 Gif-sur-Yvette Cedex
prenom.nom@cea.fr
[2] Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes
CentraleSupélec, Université Paris-Saclay, 91190 Gif-sur-Yvette France
prenom.nom@centralesupelec.fr

Résumé

Le slicing est une technique permettant d'extraire, à partir d'un programme donné, un programme plus petit, appelé tranche ou slice, tel que le programme et sa slice aient un comportement identique vis-à-vis d'un critère donné (appelé critère de slicing). Le calcul de la slice est classiquement basé sur des dépendances : dépendances de contrôle et dépendances de donnée. Notre objectif actuel est de construire un outil de slicing générique, qui abstraie les spécificités des langages et qui soit réutilisable, en évitant de reconstruire un outil de slicing pour chaque nouveau langage. Une étape majeure dans cette direction est de traiter les flots de contrôle non structurés.

Une méthode de calcul des dépendances de contrôle pour tout type de flot de contrôle a été proposée et prouvée (sur papier) en 2011 par Danicic et al pour un graphe orienté (fini) quelconque. Dans ce travail, nous avons réalisé une formalisation de cet algorithme dans Coq. Nous proposons également un nouvel algorithme pour le calcul des dépendances de contrôle qui optimise la technique de Danicic. L'optimisation consiste à enregistrer des informations intermédiaires sur les chemins du graphe et s'appuyer sur ces informations pour accélérer les itérations suivantes. Cela rend l'algorithme et la preuve plus complexes, et nécessite des invariants plus subtils. Nous formalisons et prouvons le nouvel algorithme dans l'outil Why3. Afin de comparer notre algorithme à celui de Danicic, nous faisons une évaluation expérimentale des deux algorithmes sur des milliers de graphes générés aléatoirement et allant jusqu'à plusieurs milliers de nœuds. Les résultats montrent que la nouvelle technique est nettement plus efficace et peut s'appliquer à des graphes (et, donc, des CFGs de programmes) réels.

1 Introduction

Le *slicing* est une technique proposée par Mark Weiser en 1979 [14, 15] pour simplifier un programme vis-à-vis d'un point d'intérêt appelé critère de slicing. L'objectif du slicing est de créer un programme plus petit que le programme initial mais ayant le même comportement que ce dernier vis-à-vis du critère de slicing. Le calcul de la slice se décompose généralement en deux phases : d'abord, le calcul de l'ensemble d'instructions à préserver, le *slice set*, basé sur les dépendances de contrôle et de données ; puis la construction de la slice elle-même à partir du programme initial et du slice set. Depuis 1979, plusieurs variantes de slicing ont été proposées. Celle de Weiser est désormais nommée *slicing statique arrière.* Cette variante est considérée aussi dans cet article, dans sa version intra-procédurale.

Selon Weiser, la slice est obtenue à partir du programme initial en supprimant une ou plusieurs instructions qui n'ont pas d'impact sur le critère de slicing. Il convient de préciser qu'une instruction est toujours supprimée avec toutes les instructions qu'elle contient le cas échéant (pour des boucles ou des branches de conditions). On ne peut, par exemple, supprimer la condition d'une boucle sans supprimer son corps. Lorsqu'une slice peut ainsi être construite par simples suppressions à partir du programme initial, on dit qu'elle est un *quotient* de ce dernier,

```
1   if (x >= 0) {
2     while (x > 0) {
3       x--;
4     }
5     x++;
6   } else {
7     while (x < 0) {
8       x++;
9     }
10    x--;
11  }
```

(a) Programme original

```
1   if (x >= 0) {
2
3
4
5
6   } else {
7     while (x < 0) {
8       x++;
9     }
10
11  }
```

(b) Slice par rapport à la ligne 8

FIGURE 1 – Programme jouet et sa slice

selon la dénomination utilisée dans [3]. Une limitation de cette définition, déjà identifiée par Weiser [14] et reprise par d'autres travaux [13, 8, 4], vient du fait que, pour certains langages, la suppression d'une instruction peut conduire à un programme mal formé. Une autre limitation apparaît lorsqu'on souhaite optimiser le slicing grâce à une analyse statique préalable. Par exemple, si l'analyse de valeurs garantit que la condition d'une conditionnelle est toujours fausse, on pourrait vouloir garder le contenu de la branche `else` et supprimer la branche `then` et la conditionnelle elle-même dans la slice, qui ne serait alors pas un quotient. D'autres limitations en présence de sauts (`goto`) sont illustrées ci-dessous.

Dans un travail précédent [9, 11, 2], nous avons formalisé le slicing et prouvé sa correction en Coq [12, 5] pour un langage structuré simple WHILE. La figure 1 montre l'exemple d'un programme structuré et de sa slice. Le programme de la figure 1(a) est un programme jouet écrit dans un langage semblable au C. Il incrémente ou décrémente la variable `x` suivant qu'elle est positive ou strictement négative au début de l'exécution du programme, et refait un pas en arrière, de telle sorte que la variable `x` est mise à `1` si `x >= 0`, `-1` sinon. La figure 1(b) contient la slice de ce programme par rapport à l'instruction de la ligne 8. On peut remarquer que c'est bien un quotient du programme initial puisqu'elle peut être obtenue à partir de ce dernier en supprimant les instructions 2–5. Si une analyse statique préliminaire montre que la condition de la ligne 1 est toujours fausse dans cet exemple, on pourrait vouloir garder dans la slice uniquement les lignes 7–9, mais cette slice ne serait pas un quotient.

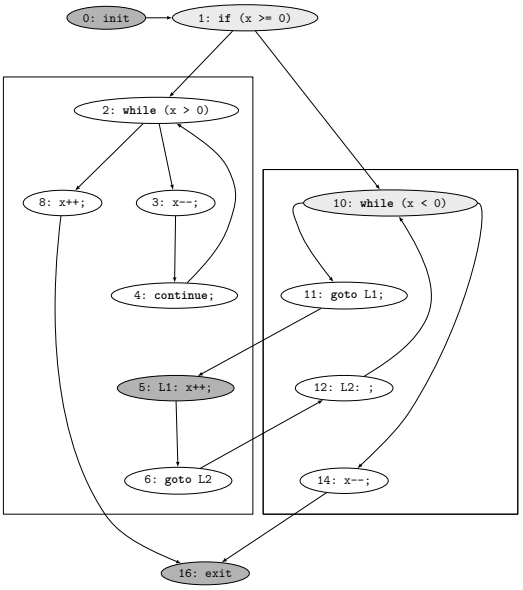
Dans un langage plus riche, avec des opérateurs moins contraints comme `goto`, on peut aussi vouloir s'affranchir de cette contrainte du quotient. Le programme 1(a) a été transformé pour donner un programme au même comportement présenté à la figure 2(a). Il est identique au premier programme si ce n'est que le corps de la deuxième boucle est inséré dans le corps de la première grâce à des `goto`. Plus précisément, les `goto` aux lignes 11 et 6 permettent l'exécution de la ligne 5 lorsque le corps de la deuxième boucle est exécuté, tandis que le `continue` de la ligne 4 fait en sorte que les lignes 5–6 soient contournées lorsque le corps de la première boucle est exécuté.

Intuitivement, on souhaiterait que la slice par rapport à la ligne 5 soit similaire à celle de la figure 1(b). Il faudrait notamment supprimer la boucle ligne 2. Mais évidemment la ligne 5, faisant partie du critère de slicing, doit être préservée (cf. fig. 2(b)). Cependant, la règle stricte imposant que la slice soit un quotient interdit de supprimer un `while` (ici, ligne 2) sans supprimer aussi le corps de la boucle.

Pour s'affranchir de cette règle, on souhaite construire un *slicer* abstrayant les spécificités

```
1   if (x >= 0) {
2     while (x > 0) {
3       x--;
4       continue;
5       L1: x++;
6       goto L2;
7     }
8     x++;
9   } else {
10    while (x < 0) {
11      goto L1;
12      L2: ;
13    }
14    x--;
15  }
```

(a) Programme réécrit avec des `goto`



(b) CFG, et le slice-set désiré $W = \{0, 1, 5, 10, 16\}$

FIGURE 2 – Programme jouet modifié et son CFG. Le critère de slicing est $V_0' = \{0, 5, 16\}$.

du langage d'entrée. On souhaite de plus qu'il soit indépendant du langage, pour pouvoir le réutiliser quel que soit le langage considéré. Une étape majeure dans l'écriture d'un tel slicer est un algorithme générique calculant les dépendances de contrôle y compris en présence de flots de contrôle non structurés.

En 2011, Danicic et al [6] ont proposé une théorie des dépendances de contrôle qui cherche à capturer l'essence de ce type de dépendance. Leur formalisation représente les programmes manipulés par le slicing sous la forme de graphes de flot de contrôle (CFG) et les critères de slicing sous la forme d'ensembles de sommets. La figure 2(b) représente le programme de la figure 2(a) sous la forme d'un CFG. Le critère de slicing $V_0' = \{0, 5, 16\}$ (en gris foncé) correspond à celui utilisé pour la figure 1. Il contient en plus les nœuds d'entrée et de sortie. La slice est alors calculée sous forme d'un CFG (et n'a pas forcément de représentation canonique par un programme). De plus, dans cette approche, si la condition de la ligne 1 est prouvée toujours fausse, on peut simplement retirer l'arête (1,2) pour ne plus devoir garder cette condition dans le slice set.

Contributions. [1] Nous avons implémenté en Coq la théorie des dépendances de contrôle proposée par Danicic et al qui généralise les autres relations de ce type (cf. [6]). Leur formalisation contient notamment une notion de dépendance de contrôle applicable à des graphes orientés arbitraires, un algorithme calculant des ensembles stables par rapport à celle-ci, ainsi que la preuve de correction (sur papier) de cet algorithme. Ces trois concepts font partie de notre formalisation en Coq qui permet d'extraire une implémentation certifiée. Pour pallier une faible efficacité de cet algorithme (dont l'amélioration a été supposée possible par ses auteurs [6]), nous avons également proposé un nouvel algorithme, bénéficiant d'un stockage de résultats de calcul intermédiaires et leur réutilisation dans les itérations suivantes. Nous avons prouvé

1. La formalisation de l'algorithme de Danicic en Coq, la formalisation du nouvel algorithme dans Why3 et nos implémentations des deux algorithmes en OCaml sont disponibles sur http://perso.ecp.fr/~lechenetjc/control/.

cet algorithme dans l'outil Why3 [1, 7] et démontré par des expérimentations ses meilleures performances par rapport à l'algorithme initial de Danicic.

Cet article court est basé sur les résultats d'un papier soumis [10]. La section 2 présente la notion de dépendance de contrôle et le principe de l'algorithme introduits par Danicic et al. Puis, la section 3 présente notre algorithme optimisé, discute de sa vérification et des expérimentations. Enfin, la section 4 conclut l'article.

2 Stabilité par dépendances de contrôle faibles

Cette section présente la théorie des dépendances de contrôle faibles développée dans [6]. Dans cette section, G est un graphe orienté fini et V' un sous-ensemble des sommets de G. On illustrera les définitions sur le CFG de la figure 2(b). Pour les exemples, on utilisera le critère de slicing $V'_0 = \{0, 5, 16\}$.

2.1 Définitions

Le concept d'ensemble stable par dépendances de contrôle faibles (SDCF, *weakly control-closed* dans [6]) généralise la notion de dépendance de contrôle à des graphes orientés finis arbitraires. En ne considérant que les dépendances de contrôle, l'ensemble de sommets à préserver dans la slice (le slice set) est le plus petit ensemble SDCF contenant l'ensemble de sommets formant le critère de slicing.

Définition 2.1 (Image d'un sommet, *first observable element*)**.** *Soit u et v deux sommets de G et $v \in V'$. On dit que v est une* image *de u dans V' s'il existe un chemin de u à v dont l'intersection avec V' est égale à v. En d'autres termes, il existe un chemin depuis u vers v sur lequel le premier élément de V' atteint est v. En particulier, les sommets dans V' ont pour seule image eux-mêmes.*

Par exemple, l'ensemble des images du sommet 16 dans V'_0 est $\{16\}$, celui de 11 est $\{5\}$, celui de 10 est $\{5, 16\}$, celui de 1 est également $\{5, 16\}$.

Définition 2.2 (Attracteur faible, *weakly committing vertex*)**.** *Un sommet u est un* attracteur faible *vers V' dans G s'il possède au plus une image dans V'.*

D'après ci-dessus, les sommets 11 et 16 sont des attracteurs faibles vers V'_0, 1 et 10 n'en sont pas.

Définition 2.3 (Ensemble SDCF, *weakly control-closed set*)**.** *V' est* stable par dépendances de contrôle faibles (SDCF) *dans G si tous les sommets atteignables depuis V' sont des attracteurs faibles vers V' dans G.*

Intuitivement, si V' est SDCF, il n'existe aucun point de choix en dehors de V' qui soit atteignable depuis V' et qui puisse mener vers deux images différentes dans V'. Ce concept d'ensemble SDCF est effectivement adapté pour le slicing, car de tels points de choix hors de V' pourraient permettre au programme initial de diverger vis-à-vis de sa slice. Dans l'exemple considéré, $\emptyset$ et $\{16\}$ sont SDCF parce qu'aucun sommet de V'_0 n'est atteignable en dehors de l'ensemble considéré. $\{0\}$ et $\{0, 16\}$ sont SDCF parce que 0 n'est atteignable que de 0. V'_0 n'est pas SDCF, mais si on l'étend en $V'_0 \cup \{1, 10\}$, on obtient un ensemble SDCF.

2.2 Reformulation

On désirerait un algorithme pour calculer le plus petit SDCF contenant un ensemble donné (i.e. sa clôture). Les ensembles SDCF possèdent une autre caractérisation qui justifie un tel algorithme.

Définition 2.4 (Décideur faible, *weakly deciding vertex*)**.** *Un sommet u est un* décideur faible *pour V' dans G s'il existe deux chemins non triviaux terminant dans V', qui partent de u et ne possèdent aucun autre sommet commun.*

Dans notre exemple, 10 est un décideur faible pour V'_0, les deux chemins étant $10 \to 11 \to 5$ et $10 \to 14 \to 16$. De même, 1 est un décideur faible pour V'_0 ($1 \to 2 \to 8 \to 16$ et $1 \to 10 \to 11 \to 5$). 2 n'est pas décideur faible pour V'_0 puisqu'il ne possède que 16 comme image dans V'_0.

Propriété 2.1 (Caractérisation des SDCF)**.** *V' est SDCF dans G si et seulement si tous les décideurs faibles pour V', qui sont atteignables depuis V', se trouvent dans V'.*

2.3 Algorithme de Danicic pour le calcul de la clôture

Les décideurs faibles pour V' atteignables depuis V' sont les sommets à ajouter à V' pour obtenir un ensemble SDCF dans G. Intuitivement, ils représentent en effet les points de choix les plus proches de V' pouvant mener à deux images différentes dans V'. L'algorithme pour calculer la clôture proposé par Danicic et al [6] ne calcule pas ces éléments directement, mais propose d'ajouter itérativement des sommets dont la caractérisation est donnée ci-dessous.

Définition 2.5 (Arête critique)**.** *On dit qu'une arête de G est une* arête critique *pour V' si son début a au moins deux images dans V' et que sa fin a exactement une image dans V'.*

On montre que les débuts d'arêtes critiques pour V' sont des décideurs faibles pour V', et réciproquement que tant que V' n'est pas SDCF, il existe une arête critique pour V' dont le début est atteignable depuis V'. En revanche, à une itération donnée, tout décideur faible pour V' n'est pas nécessairement le début d'une arête critique pour V'. Cela justifie l'algorithme itératif suivant.

Algorithme 2.1 (Construction du plus petit ensemble SDCF)**.** *On commence par poser $W = V'$. Trouver une arête critique pour W dont le début est atteignable depuis W, et ajouter son début à W. Répéter cette étape autant que possible. Lorsqu'il n'existe plus de telles arêtes, retourner W.*

Appliquons l'algorithme sur $W_0 = V'_0$. Ici tous les sommets sont atteignables depuis 0, qui est dans W_0, donc on peut ignorer la condition d'atteignabilité. D'après ci-dessus, (10,11) est une arête critique pour W_0, donc on pose $W_1 = W_0 \cup \{10\}$. (1,10) est une arête critique de W_1, puisque 10 n'a que 10 comme image, tandis que 1 a 10 et 16. Donc $W_2 = W_1 \cup \{1\}$. Il n'existe pas d'arête critique pour W_2, donc W_2 est le plus petit ensemble SDCF contenant V'_0. On peut noter que les éléments de $W_2 = \{0, 1, 5, 10, 16\}$ (coloriés dans la figure 2(b)) correspondent bien aux instructions préservées dans la slice structurée de la figure 1(b) (les dépendances de données n'ajoutent ici aucun sommet à la slice).

Nous avons fait la preuve de cet algorithme et la formalisation des notions sous-jacentes en Coq[1].

3 Algorithme optimisé

3.1 Description informelle

Une raison possible du manque d'efficacité de l'algorithme de Danicic est l'absence de partage d'information entre les différentes itérations de l'algorithme. Les images de chaque sommet sont recalculées à chaque itération puisque l'ensemble vis-à-vis duquel elles sont calculées change. Nous proposons une optimisation pour remédier en partie à ce problème : au lieu d'ajouter les nœuds un par un, il est possible à chaque itération de détecter toutes les arêtes critiques et d'ajouter leurs sources, ce qui permet de recalculer moins souvent les images des nœuds du graphe.

Mais nous proposons également d'aller plus loin et de réutiliser l'information obtenue sur les chemins du graphe dans les itérations ultérieures. L'idée principale de notre algorithme optimisé consiste à étiqueter chaque nœud u par une image de u dans l'ensemble résultat en construction W. Ces étiquettes survivent aux itérations et peuvent donc être réutilisées.

Contrairement à l'algorithme de Danicic, notre algorithme ne calcule pas directement le plus petit SDCF, en ce qu'il ne vérifie pas tout de suite l'atteignabilité des sommets qu'il accumule. Il est cependant facile d'éliminer les nœuds non-atteignables dans un second temps pour obtenir le SDCF.

Notre approche nécessite de manier les étiquettes avec précaution, de manière à ce qu'elles restent à jour et restent des images des nœuds qui les portent. En pratique, certains nœuds ne sont parfois pas à jour, mais cela n'empêche pas l'algorithme de fonctionner correctement.

Notre algorithme prend en entrée un graphe G et un sous-ensemble de ses sommets V'. Il manipule trois objets : un ensemble W égal à V' initialement, qui grossit pendant l'algorithme et qui à la fin contient le résultat ; une table d'association *obs* qui associe à un sommet u au plus un sommet dans W atteignable depuis u et qui est une image de u dans W à la fin ; une file L de nœuds en attente de traitement et qui contient tous les nœuds de V' initialement. Chaque itération se déroule comme suit. Un sommet u est prélevé de L si L est non vide et tous les nœuds qui sont directement ou indirectement des prédécesseurs de u non masqués par W sont étiquetés par u. Après cette phase de propagation, de nouveaux décideurs faibles pour V' sont détectés. Chacun de ces nœuds voit son étiquette mise à jour à lui-même et est ajouté dans W et L. Si L n'est pas vide, une nouvelle itération commence. Sinon, l'algorithme se termine et, après un filtrage des nœuds non atteignables depuis V', renvoie dans W le plus petit ensemble SDCF contenant V' (i.e. la clôture de V'), ainsi qu'un étiquetage de chaque nœud par son image dans W, si elle existe.

3.2 Preuve formelle et évaluation expérimentale

Nous avons réalisé la preuve de l'algorithme optimisé dans Why3[1], afin de profiter de prouveurs automatiques. Cette preuve a nécessité d'identifier des invariants très subtils afin d'exprimer correctement les propriétés de l'étiquetage. Tous les lemmes nécessaires sauf un ont été formalisés et prouvés en Why3. Ce lemme non prouvé était toutefois déjà prouvé dans la formalisation Coq. Une preuve papier partielle est faite dans [10].

Nous avons également implémenté l'algorithme de Danicic et notre algorithme optimisé en OCaml à l'aide d'OCamlgraph. Pour nous assurer de leur correction, nous les avons testés sur une centaine d'exemples (prenant pour oracle l'implémentation certifiée, extraite grâce à la formalisation Coq et beaucoup plus lente). Ensuite, les deux versions ont été exécutées sur plusieurs milliers de graphes aléatoires générés par OCamlgraph, avec le nombre de noeuds variant entre 10 et 6500. Les résultats (cf. [10]) montrent que l'algorithme de Danicic explose pour les graphes avec quelques centaines de nœuds, alors que notre algorithme reste efficace sur des graphes de plusieurs milliers de nœuds.

4 Conclusion

Nous avons réalisé une formalisation en Coq d'une théorie de dépendances de contrôle faibles incluant la définition d'une notion de dépendance, d'une notion de stabilité correspondante et d'un algorithme pour le calcul de la clôture proposé par Danicic et al. La version actuelle de cette formalisation comprend plus de 8000 lignes de code Coq. Elle présente un intérêt à la fois théorique et pratique : une version certifiée peut servir d'oracle de tests. Nous avons également proposé un nouvel algorithme et réalisé sa preuve dans l'outil Why3. Enfin, nous avons réalisé une étude expérimentale qui montre un gain de performance important de notre technique par rapport à l'algorithme initial de Danicic. Dans le présent article, nous avons présenté nos motivations et quelques éléments clefs de ces contributions. Un article plus complet a été soumis [10]. Les formalisations complètes et les versions implémentées des deux algorithmes sont disponibles en ligne[1].

Références

[1] Why3, a tool for deductive program verification, GNU LGPL 2.1. http://why3.lri.fr.

[2] Formalization of relaxed slicing, 2016. http://perso.ecp.fr/~lechenetjc/slicing/.

[3] Richard W. Barraclough, David Binkley, Sebastian Danicic, Mark Harman, Robert M. Hierons, Akos Kiss, Mike Laurence, and Lahcen Ouarbya. A trajectory-based strict semantics for program slicing. *Theor. Comp. Sci.*, 411(11–13) :1372–1386, 2010.

[4] José Bernardo Barros, Daniela Carneiro da Cruz, Pedro Rangel Henriques, and Jorge Sousa Pinto. Assertion-based slicing and slice graphs. In *SEFM 2010*, 2010.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer, 2004.

[6] Sebastian Danicic, Richard W. Barraclough, Mark Harman, John Howroyd, Ákos Kiss, and Michael R. Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theor. Comput. Sci.*, 412(49) :6809–6842, 2011.

[7] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[8] Mark Harman, David Binkley, and Sebastian Danicic. Amorphous program slicing. *Journal of Systems and Software*, 68(1) :45–64, 2003.

[9] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Cut branches before looking for bugs : Sound verification on relaxed slices. In *FASE'16 (Part of ETAPS'16)*, pages 179–196, 2016.

[10] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Fast computation of arbitrary control dependencies. 2018. Submitted.

[11] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. Coq a dit : fromage tranché ne peut cacher ses trous. In *Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, 2016.

[12] The Coq Development Team. The Coq proof assistant, v8.6, 2017. http://coq.inria.fr/.

[13] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[14] Mark Weiser. Program slicing : Formal, psychological and pracical investigation of an automatic program abatraction method. phd dissertion. university of michigan. *Ann Arbor, Michigan*, 1979.

[15] Mark Weiser. Program slicing. In *ICSE 1981*, 1981.

Merci à


list
ceatech


PROOF IN USE
SPARK-2014.ORG/PROOFINUSE


GPL
Génie de la Programmation et du Logiciel
GdR
Groupement de Recherche


OCaml PRO


TRUST IN SOFT