



HAL
open science

JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs

Timothy Bourke, Delphine Demange

► **To cite this version:**

Timothy Bourke, Delphine Demange. JFLA 2023 - 34èmes Journées Francophones des Langages Applicatifs. 2023, Journées Francophones des Langages Applicatifs. hal-03962188v1

HAL Id: hal-03962188

<https://inria.hal.science/hal-03962188v1>

Submitted on 30 Jan 2023 (v1), last revised 22 Feb 2023 (v2)

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

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

34^{èmes} Journées Francophones
des
Langages Applicatifs

du 31 janvier au 3 février 2023 à Praz-sur-Arly (Haute-Savoie)

Inria



Préface

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. Le spectre des travaux présentés aux JFLA est très large : il touche les aspects les plus théoriques de la conception des langages applicatifs jusqu'à ses applications industrielles. La 34^{ème} édition des JFLA se déroulera à L'Alisier, à Praz-sur-Arly (Haute-Savoie).

Cette année, nous avons sélectionné 12 articles longs de recherche, 2 articles courts de recherche, et 3 démonstrations de prototypes. Cette sélection a été faite par les membres du comité de programme, que nous remercions chaleureusement, à partir de 21 soumissions initiales.

Nous aurons aussi le grand plaisir d'écouter un exposé invité de Sylvie Boldo sur l'agrégation externe d'informatique, un exposé invité de Xavier Leroy sur les effets en OCaml 5 et un cours invité de Cyril Cohen sur les bibliothèques de *Mathematical Components*.

Contributions sélectionnées. Les contributions sélectionnées pour ces JFLA couvrent des sujets allant des bibliothèques OCaml aux métathéories des langages, en passant par de la preuve formelle appliquée à des problèmes pratiques, et par des analyses statiques par interprétation abstraite.

Une extension de la bibliothèque OCaml `unionFind` aux références satisfaisant un mécanisme de retour arrière est proposée pour traiter efficacement la vérification de typage d'une extension de System F. Une autre bibliothèque OCaml est présentée qui permet la modélisation, la simulation et la compilation des programmes quantiques de bas niveau.

Pour un langage de programmation décrivant des modèles probabilistes synchrones, c'est-à-dire, des fonctions qui définissent et manipulent de suites de distributions, une méthode d'inférence est proposée pour traiter les paramètres constants des modèles.

En matière de métathéorie des langages, une première étude des modules génératifs ML propose un aller-retour entre les modules OCaml et un encodage en F^ω , identifiant au passage un système intermédiaire permettant d'éviter le fossé d'expressivité entre le langage des modules et celui des signatures. Une deuxième étude considère deux modèles de calcul équipés de leur modèle de réalisabilité, et s'intéresse à une traduction avec passage de continuation entre ces deux calculs, pour déterminer si elle peut préserver leurs réalisations.

Plusieurs contributions traitent de l'interprétation, concrète ou abstraite, des langages de programmation. Une première fait état d'une technique pour implémenter des interprètes monadiques en combinant, de façon modulaire et dans Coq, les signatures, les calculs et les règles de raisonnement. Une deuxième présente un outil pour générer des interprètes OCaml à partir d'un langage minimaliste de description de sémantiques. Une autre propose une interprétation abstraite des sémantiques squelettiques. Une dernière présente une technique d'interprétation abstraite pour un langage fonctionnel typé du premier ordre avec un traitement spécifique des types algébriques récursifs.

Plusieurs cas d'application de preuves formelles sont aussi présentés, qui utilisent des cadres de formalisation variés. F^* est utilisé dans l'analyse de la sécurité de TreeSync, un sous-protocole d'authentification pour une messagerie de groupe sécurisée. Why3 sert de base à un logiciel pour prouver la sûreté des logiciels appris par des techniques d'intelligence artificielle. Coq est utilisé dans plusieurs contributions. Une formalisation est faite en Coq de la propriété de sûreté des équipements nucléaires, par analyse des modes de défaillance et de leurs effets. Pour un cadre d'optimisation de programmes par transformations source-à-source vérifiées en Coq, une nouvelle représentation des arbres de syntaxe abstraite intègre des invariants et les preuves associées. Un compilateur vérifié pour un langage synchrone est étendu à deux nouvelles structures d'activation, en complétant l'analyse des dépendances associées. Et une investigation est faite de la

problématique d'intégration au compilateur C formellement vérifié CompCert de flottants non conformes à la norme IEEE-754 et utilisés par certaines cibles embarquées.

D'autres contributions se concentrent sur les outils de preuve eux-mêmes. Une approche vise à rendre la preuve de programmes avec pointeurs à la portée des prouveurs automatiques en projetant une structure récursive sur un domaine numérique. Un traducteur automatique de Metamath vers Dedukti offre deux encodages possibles, permettant ainsi différents compromis en termes d'interopérabilité.

Remerciements Nous remercions en premier lieu les auteurs de tous les articles soumis aux JFLA 2023, sans qui les journées n'auraient pas lieu.

Nous tenons aussi à exprimer nos vifs remerciements aux rapporteurs du comité de programme et aux rapporteurs externes. Vous trouverez leurs noms sur les pages suivantes.

Nous remercions également nos chers orateurs invités, qui nous font l'honneur de partager leur expérience et leurs connaissances avec les participants des journées.

Pour l'organisation et la gestion financière, nous remercions les services d'Inria Paris, notamment Tiphaine Leblanc et Fodil Zaidi, et du CNRS à Rennes, notamment Antoine L'Azou. Leur soutien nous a été indispensable. Concernant la publication des actes, nous remercions le Centre pour la Communication Scientifique Directe, et notamment Hélène Lowinger.

Nous remercions enfin les sponsors, listés sur la couverture arrière des actes. Leur implication a permis de subventionner la venue de plusieurs doctorantes, doctorants, étudiantes et étudiants. Elle témoigne également de l'importance des travaux et résultats de recherche scientifique présentés aux JFLA, que leurs retombées pratiques, voire industrielles, soient immédiates ou bien envisagées à plus long terme.

Programmatiquement vôtre,
Timothy Bourke et Delphine Demange.

Comité de programme

Timothy Bourke	Inria, ÉNS de Paris
Delphine Demange	Univ Rennes, Inria, CNRS, IRISA
François Bobot	CEA List
Lélio Brun	National Institute of Informatics
Raphaëlle Crubillé	CNRS, LIS, Aix-Marseille
Pierre-Evariste Dagand	CNRS, IRIF, Paris
Stefania Gabriela Dumbrava	Institut Polytechnique de Paris / ENSIIE
Benjamin Farinier	Université Rennes 1 / IRISA
Aymeric Fromherz	Inria de Paris
Diane Gallois-Wong	Nomadic Labs
Assia Mahboubi	Inria U. de Rennes, Nantes
Gabriel Radanne	LIP ENS Lyon / Inria U. GA, Lyon
Laurence Rideau	Inria U. Côte d'Azur, Sophia Antipolis
Pierre Roux	Onera, Toulouse
Boris Yakobowski	AdaCore, Paris

Rapporteurs externes

Germán Andrés Delbianco
Nicolas Ayache
Guillaume Claret
Guillaume Geoffroy
Boubacar Sall

Table des matières

Exposés et cours invités	
À propos de l'agrégation d'informatique	2
<i>Sylvie Boldo</i>	
Tutoriel Mathematical Components	3
<i>Cyril Cohen</i>	
Théorie et pratique des effets en OCaml 5	4
<i>Xavier Leroy</i>	
<hr/>	
Articles longs	
Effectful programming across heterogeneous computation—Work in Progress	7
<i>Jean Abou Samra, Martin Bodin et Yannick Zakowski</i>	
Filtrer sans s'appauvrir : Inférer les paramètres constants des modèles réactifs probabilistes	24
<i>Guillaume Baudart, Grégoire Bussone, Louis Mandel et Christine Tasson</i>	
An AST for Representing Programs with Invariants and Proofs	43
<i>Guillaume Bertholon et Arthur Charguéraud</i>	
Retrofitting OCaml modules : Fixing signature avoidance in the generative case	59
<i>Clément Blaudeau, Didier Rémy et Gabriel Radanne</i>	
Analyse de dépendance vérifiée pour un langage synchrone à flot de données	101
<i>Timothy Bourke, Basile Pesin et Marc Pouzet</i>	
Formal proofs applied to system models	121
<i>Évelyne Contejean et Andrei Samokish</i>	
Do CPS translations also translate realizers?	134
<i>Samuel Gardelle et Étienne Miquey</i>	
Building CFA for λ -calculus from Skeletal Semantics	152
<i>Thomas Jensen, Vincent Rébiscoul et Alan Schmitt</i>	
Traduire l'univers des mathématiques en Dedukti, sans univers	172
<i>Amélie Ledein et Elliot Butte</i>	
Backtracking reference stores	190
<i>Gabriel Scherer</i>	
Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques rékursifs	211
<i>Milla Valnet, Raphaël Monat et Antoine Miné</i>	
Vérification symbolique de protocoles cryptographiques en F^* : application au sous-protocole TreeSync de MLS	243
<i>Théophile Wallez</i>	

Articles courts

Comment dompter un troupeau de flottants sauvages ? 265
Arthur Correnson

L'arithmétique de séparation 274
Jean-Christophe Filliâtre et Andrei Paskevich

Démonstrations

Goose : an OCaml environment for quantum computing 285
Denis Carnier, Arthur Correnson, Christopher McNally et Youssef Moawad

Caractériser des propriétés de confiance d'IA avec Why3 288
Julien Girard-Satabin, Michele Alberti, François Bobot et Zakaria Chihani

Necro ML : Kit de Nécromancie 293
Louis Noizet et Alan Schmitt

Exposés et cours invités

À propos de l'agrégation d'informatique

Exposé invité

Sylvie Boldo

Université Paris-Saclay, Université Paris-Sud, CNRS, Inria

Résumé

Cet exposé présentera l'agrégation externe d'informatique ainsi qu'un retour sur sa première session qui a eu lieu en 2022. Le rapport du jury sur la session 2022 se trouve en ligne.¹ Plus d'informations (épreuves, annales, environnement informatique, FAQ, ...) se trouvent sur le site du jury : <https://agreg-info.org/>

1. https://media.devenirensignant.gouv.fr/file/agreg_externe/84/0/rj-2022-agregation-externe-informatique_1428840.pdf

Mathematical Components

Tutoriel invité

Cyril Cohen

Inria Sophia Antipolis

Résumé

Mathematical Components est un ensemble des bibliothèques de mathématiques formalisées développées avec l'assistant de preuve Coq. Ces bibliothèques ont été utilisées pour des projets de formalisation à grande échelle, tels que celle du théorème des quatre couleurs et du théorème de l'ordre impair (Feit-Thompson). Dans ce cours en deux parties, nous explorerons les bibliothèques Mathematical Components disponibles et je fournirai des éléments pour en lire le contenu, les utiliser et y contribuer.

La première partie du cours donnera des clés de lecture de la bibliothèque, et en particulier comment y trouver et utiliser les résultats recherchés. Pour cela nous passerons en revue les conventions de nommage et d'usage en vigueur, afin de faire des recherches pertinentes dans le contenu des bibliothèques. Nous verrons également comment lire les scripts de preuve écrits avec le langage de tactique `SSReflect`, ainsi que les quelques tactiques indispensables pour utiliser des résultats de la bibliothèque. Cela sera illustré avec quelques exemples lors de sessions d'exercices encadrés.

La deuxième partie du cours sera consacrée aux hiérarchies de structures présentes dans les bibliothèques Mathematical Components et à la manière dont elles sont générées avec `Hierarchy Builder` à partir de la version 2.0+alpha1 de Mathematical Components. Je présenterai les motivations pour l'utilisation de `Hierarchy Builder` par rapport à une gestion manuelle avec des records, des classes ou des modules. Ensuite nous examinerons les différentes façons d'instancier des structures, puis comment ajouter des nouvelles structures en utilisant `Hierarchy Builder`, et enfin comment modifier du code tout en préservant la compatibilité arrière.

Théorie et pratique des effets en OCaml 5

Cours invité

Xavier Leroy

Collège de France

Résumé

La version 5 d'OCaml ajoute, sous forme de bibliothèque, un nouveau trait au langage Caml : les effets définis par l'utilisateur et les gestionnaires d'effets.

La première partie de ce cours illustrera la pratique des effets en OCaml 5 [1, 2]. On peut les voir comme une généralisation des exceptions : lever un effet interrompt le calcul en cours, tout comme lever une exception ; mais au contraire des exceptions, le calcul en cours peut être redémarré plus tard. C'est le gestionnaire de l'effet qui décide quand et avec quelle valeur redémarrer ce calcul.

Une première utilisation des effets est l'inversion du contrôle, comme par exemple transformer un itérateur fonctionnel « à la Caml » en itérateur générateur « à la Java ». Toutes sortes de coroutines peuvent également être définies par l'utilisateur en termes d'effets. Plus généralement, les effets d'OCaml 5 ont la puissance d'expression des continuations délimitées linéaires.

Nous terminerons ce tutoriel en esquissant l'implémentation d'une bibliothèque de fibres (*lightweight threads*) avec entrées/sorties non bloquantes, utilisables en style direct (pas de passage explicite de continuations comme avec la bibliothèque LWT). La bibliothèque EIO en cours de développement [3] pousse cette approche à base de fibres jusqu'à fournir une alternative performante et en style « direct » à LWT.

Dans la deuxième partie du cours, nous aborderons les fondations théoriques des effets et des gestionnaires d'effets. Nous partirons des monades, un concept issu de la théorie des catégories et appliqué à la sémantique dénotationnelle par Moggi en 1989 puis à la programmation fonctionnelle pure par Wadler et d'autres dès 1991. Les monades fournissent un cadre élégant pour décrire de nombreuses sortes d'effets allant de l'affectation à la programmation probabiliste.

Le lambda-calcul « computationnel » de Moggi [4] décrit de manière indépendante de la monade sous-jacente la propagation des effets. Mais la manière dont les effets sont engendrés et gérés est propre à chaque monade. La théorie des effets algébriques de Plotkin et Power [5] fournit un cadre général pour décrire la génération et le traitement des effets et les spécifier de manière algébrique via des équations, un peu comme dans les types abstraits algébriques [6]. Cette théorie débouche naturellement sur les gestionnaires d'effets comme nouvelle construction du langage permettant d'implémenter les effets algébriques.

Le cours se terminera par une brève discussion du typage statique des effets, qui n'est pas encore implémenté en OCaml 5 mais pour lequel plusieurs systèmes de types ont été proposés [7, 8], ainsi que de la preuve par vérification déductive de programmes utilisant des effets [9].

Références

- [1] KC Sivaramakrishnan, *The OCaml system release 5.0, Documentation and user's manual*, section 12.24, «Effect handlers», <https://v2.ocaml.org/releases/5.0/htmlman/effects.html>.
- [2] Daniel Hillerström et KC Sivaramakrishnan, *Concurrent Programming with Effect Handlers*, <https://github.com/ocaml-labs/ocaml-effects-tutorial>.

- [3] Thomas Leonard *et al.*, *EIO : Effects-based direct-style IO for multicore OCaml*, <https://github.com/ocaml-multicore/eio>.
- [4] Eugenio Moggi, *Notions of computations and monads*, Inf. Comput. 93(1), 1991.
- [5] Gordon Plotkin et John Power, *Algebraic Operations and Generic Effects*, Applied Categorical Structures 11 :69–94, 2003.
- [6] Andrej Bauer, *What is algebraic about algebraic effects and handlers?*, arXiv :1807.05923, 2019.
- [7] Andrej Bauer et Matija Pretnar, *An effect system for algebraic effects and handlers*, Logical Methods in Computer Science 10(4), 2014.
- [8] Daan Leijen, *Type directed compilation of row-typed algebraic effects*. In : Principles of Programming Languages (POPL) 2017.
- [9] Paulo Emilio de Vilhena et François Pottier, *A separation logic for effect handlers*, Proc. ACM Program. Lang. 5(POPL) : 1-28, 2021.

Articles longs

Effectful Programming across Heterogeneous Computations

— Work in Progress

Jean Abou Samra¹, Martin Bodin², and Yannick Zakowski³

¹ ENS, Paris, France

`jean@abou-samra.fr`

² Inria, Grenoble, France

`martin.bodin@inria.fr`

³ Inria, Lyon, France

`yannick.zakowski@inria.fr`

Keywords: Formal Semantics, Free Monad, Coq, Algebraic Effects.

Abstract

Monadic programming is a popular way to embed effectful computations in purely functional languages. In particular, the so-called free-monad comes with the promise of extensibility and modularity: computations are seen as syntax arising from a signature of operations they may perform. Popularized in a programming context, the approach is nowadays used for verification in proof assistants as well, as witnessed by frameworks such as FreeSpec or Interaction Trees.

In this work in progress, we investigate the following question. Can we type each sub-computation with their minimal valid operation signature, and seamlessly combine all these monadic computations of different natures? Furthermore, can we leverage this additional precision in typing to derive monadic invariants for free?

We answer positively by suggesting two simple ideas. First, a bind operation between computations living in distinct monads can be defined by transporting them through monad morphisms. Second, to give more structure to the monads we manipulate by indexing them by a semi-lattice as a means to automatically infer the adequate morphisms. We illustrate the benefits on a minimal Coq example: a computation interacting with a memory cell.

1 Introduction

Monads were initially introduced as a semantic model for effectful computations [11]. Quickly, they have spread beyond this formal setup to become a popular programming device allowing for safely encapsulating effects in functional programs [4, 13], taking most famously a central role in Haskell’s design.

Under the lens of programming, a given monad consists of three core components.¹ First, a family of types specifies a domain of computations: the `Option` type constructor captures potentially failing computations, the `List` type constructor can be seen as specifying non-deterministic computations, state-threading functions may encapsulate stateful computations, and so on. Monads are furthermore equipped with two operations: the `ret` construct describes how to embed a pure computation into the domain of computations, while the `bind` operation describes how computations can be sequenced (hence its notation ‘`x ← m ; k`’). Figure 1 illustrates the Coq definition of a stateful monad over a single memory `cell` storing a natural number.

¹In general, monads should also come equipped with a notion of equivalence of computation. We brush this detail under the rug in this presentation, pretending that we may work with Coq’s equality `eq` everywhere. We refer the interested reader to our formal development for a proper setoid-based approach.


```

Definition monad := Type → Type.

Definition cell : Type := nat.
Definition state : monad := fun X => cell → cell * X.
Definition state_ret : ∀ X, X → state X := fun X x c => (c, x).
Definition state_bind : ∀ X Y, state X → (X → state Y) → state Y :=
  fun X Y m k c => let (c', x) := m c in k x c'.

```

Figure 1: The state monad (👉)

```

Definition signature := Type → Type.
Notation "M ~ N" := (∀ X, M X → N X).
Inductive free (E : signature) (X : Type) :=
  | pure (x : X)
  | op Y (e : E Y) (k : Y → free E X).

Variant Rd : signature :=
  | rd : Rd cell.
Variant Wr : signature :=
  | wr (c : cell) : Wr unit.

Fixpoint interp E M (MM: Monad M)
  (h : E ~ M) X (m : free E X) : M X :=
  match m with
  | pure x => ret x
  | op e k =>
    bind (h e) (fun x => interp h (k x))
  end.

Definition h_state
  : Rd +' Wr ~ state :=
  fun _ e c => match e with
  | inl1 rd => (c, c)
  | inr1 (wr c') => (c', tt)
  end.

```

Figure 2: The free monad (👉) and its use for stateful computations (👉)

Perhaps more surprisingly, monads offer a solution to Wadler’s expression problem. The *free monad* (Figure 2) acts as an extensible syntax [12] for effectful computations: effects are thought of as arising from the use of effectful operations described via an interface E . A computation is then essentially encoded as a tree where leaves contain pure computations, and nodes contain operations. The relationship between a node and its children arises from a continuation indexed by the type of answer expected from a valid implementation of the operation. This tree structure is naturally equipped with a monadic structure: leaves directly act as `ret` operations, while `bind` attaches the appropriate continuation to the leaves of the first part of the computation (👉).

The right hand-part of Figure 2 illustrates the approach when declaring computations manipulating the previously mentioned cell. The `rd` event specifies the *read* operation: the fact that one expects to get back the content of the cell following this operation is reflected in its type `rd : Rd cell`. In contrast, a *write* event `wr c` is intended to update the content of the cell, but does not entail any informative answer: its return type is merely an acknowledgement, embodied by the *unit* type. From there, one can write rich programs manipulating this memory cell as elements of the monad `free (Rd +' Wr)` where `+'` is the disjoint sum of both signatures. Read operations give rise to an infinite branching, with a branch per natural number, while write operations have a single child. In concrete syntax, a computation doubling the current content of the cell becomes:

```

Definition double : free (Rd +' Wr) unit := n ← rd;; wr (2 * n).

```

The final crucial ingredient to our story is hidden behind the *free monad*’s name: except for their return type, the operations are free of constraints, no semantics is attached to them! Because of this freedom, one can plug’n’play the monadic implementation of one’s choice: by folding over the tree, substituting nodes for their implementations, the `interp` function lifts

monadic implementations of operations into implementations of computations. For instance, `h_state` provides a typical stateful implementation to read and write operations, such that `interp h_state double` becomes the expected executable function of an initial cell.

Monadic interpreters built atop of the free monad are an increasingly popular way to formalize the semantics of programming languages in proof assistants based on dependent typed theories, such as Coq for instance. Indeed, via variants of the *delay* monad [2, 10], divergence can be internalized, freeing us from Coq’s termination checker when writing monadic definitional interpreters. When applicable, the approach offers the benefits of compositionality — in the traditional sense of denotational semantics —, of modularity — effects are specified in isolation and modularly composed —, and of executability — through extraction to OCaml for instance, testing is made possible, including when dealing with non-terminating programs.

Specifically in the Coq ecosystem, the approach has been implemented in the FreeSpec library [7, 6, 8] and the Interaction Tree (ITree) project [14, 15]. The viability of the method has been shown to scale to realistic settings such as for modelling LLVM IR [16], or to support non-determinism and model concurrency [5, 3]. In this context, *formal reasoning* about these monadic computations takes back a central place, combining the algebraic reasoning enabled by the monadic setup to unary or relational Hoare style reasoning.

The extensibility of the free approach is central to these large scale projects to enable modularity: orthogonal effects are specified in isolation, and theories can be reused across projects. It is additionally used as a means to painlessly link computations arising from different sources: one can always *translate* free computations into free computations over larger interfaces. However, in our experience, these projects have vastly shied away from leveraging the extensibility as a means to facilitate reasoning. Indeed, signatures can be thought of as a primitive type and effect system. As such, one could play the game of typing as precisely as possible every computation. Coming back to our cell example, we should benefit when possible from reflecting in the type of a computation that the cell is never written to, by typing the computation at type `free Rd X` rather than the larger `free (Rd +' Wr) X`. But we should be able to go further: if considering now a stateful computation over several cells, one should be able to reflect in the signature an over-approximation of the read and write location sites of the computation. Speculating even further, invariants could be encoded in dependent types, specifying for instance that the cell may be written to, but only in an increasing fashion.

In this work in progress, we ask the question: is this ambition practical? Can we meaningfully and painlessly combine a zoo of computations of different natures, both as free computations over diverse signatures, but also as their semantic implementations in monads of diverse complexities? Can we leverage this additional typing information to ease the reasoning in two directions: transporting invariants for free, and allowing for reasoning in simpler monadic structures?

We introduce the following early contributions that lead us believe these question should find a positive answer:

- we observe that monadic computations of distinct natures can be bound, provided there is a pair of monad morphisms into a common target (Section 3.1);
- we propose to index the subset of monads of interest by first a partial order (Section 3.2), then a directed set (Section 3.3), in order to remove the need for outrageous explicit type annotations;
- moving to a setup based on the free monad, we propose an axiomatized interface to program with the free and concrete views of the indexing domain of computations (Section 4);

```

Definition read : monad := fun X => data -> X.
Definition read_ret X (x : X) := fun c => x.
Definition read_bind X Y (m : read X) (k : X -> read Y) : read Y :=
  fun c => k (m c) c.

Definition write : monad := fun X => X * option data.
Definition write_ret X (x : X) := (x, None).
Definition write_bind X Y (m : write X) (k : X -> write Y) : write Y :=
  let '(x,mw) := m in
  match k x with
  | (y, None) -> (y, mw)
  | (y, Some w) -> (y, Some w)
  end.

Definition pure : monad := fun X => X.
Definition pure_ret X (x : X) := x.
Definition pure_bind X Y (m : pure X) (k : X -> pure Y) : pure Y := k m.

```

Figure 3: Read only, write only, and pure monads (📖)

- we provide a minimal instance of this interface to the case of a stateful computation over a cell (Section 2), and offer some thoughts as to how this work can lead to nicer reasoning principles (Section 5).

All our results are formalized in Coq². This paper can naturally be read as plain text, but we additionally provide hyperlinks to our source code as a support for the interested reader: those are indicated by a ‘📖’ symbol.

2 Running Example

As a means to guide our intuition, we consider as a minimal concrete example stateful computations over a single memory cell as introduced in Section 1. To work with such effects, one would typically (1) assemble pieces of computations written in the `free (Rd + Wr)` monad, (2) interpret these computations into the `state` monad, and (3) perform any reasoning, of functional correctness for instance, in this structure.

We propose ourselves to leverage three simple pieces of static information we may collect: a computation might only read the cell, it might only write to the cell, or it might perform neither operation. On the syntactic side, a sufficient condition to ensure these invariants is to type a piece of computation in respectively the `free Rd`, `free Wr`, or `free \emptyset` ³ monad. We hence have naturally four signatures in mind, organized into a diamond w.r.t. a form of signature inclusion.

So far, little differs from previous work: we could translate the simpler signatures into the largest one when combining computations. However, this would mean that semantically, we still see all computations through the lens of the overly general state monad. Having made the existence of these four kinds of typed computations explicit, the temptation is great to instead interpret each of them into the simplest structure possible. Figure 3 suggests such structures: read only computations should not need to return an updated cell; write only ones should not

²<https://gitlab.inria.fr/yzakowsk/ordered-signatures/-/tree/jfla23/theories>

³Where \emptyset is the empty signature.

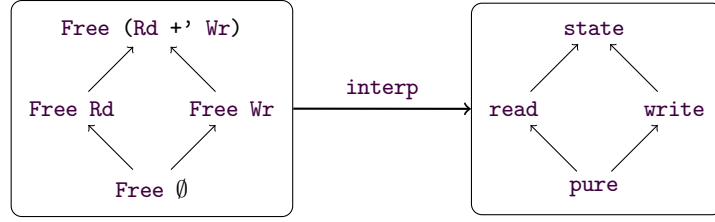


Figure 4: Flavors of stateful computations: syntax and semantics

have to take the initial cell as argument; and pure computation should live in the identity monad. We omit the formal definitions here (👉) but analogues to `h_state` can be defined to interpret each signature at play into its corresponding implementation monad.

We hence have a diamond of free monads, upon which we can climb by translation, and an isomorphic diamond of concrete monads. Although less explicit, the intuition according to which monads get “more general” when climbing in the diamond is valid in the concrete one as well. Here, the arrow between two monads M and N represents a monad morphism, i.e., an operation `fmap` : $M \rightsquigarrow N$ commuting with `ret` and `bind`. For instance, the morphism between the `read` monad and the `state` monad simply exposes that the cell has been left untouched:

```
Definition read_state_map: read ~> state := fun X m c => (c, m c). 📌
```

Figure 4 sums up the eight monads we are tempted to manipulate, four on the syntactic side, connected via interpretation to the four on the semantic side. We aim to identify the right interface necessary to program with no overhead with this heterogeneous syntax, as well as to benefit from the additional typing annotations to compose pieces of reasoning in the simplest possible structures. As a simplistic but illustrative example, we consider the following program.

```
Variable init : cell -> free Wr unit.
Variable fetch : free Rd cell.
Definition main (n : cell) : free (Rd +' Wr) bool :=
  init n ;;
  v1 <- fetch ;;
  v2 <- fetch ;;
  ret (v1 =? v2).
```

The goal is twofold. Can we write this syntax, despite `init` and `fetch` living in different syntax than `main`? And second, can we frame the right theory allowing us to leverage the typing information to prove that `main n` will always return `true` while keeping the definitions of `init` and `fetch` completely opaque? We already intuitively observe that the second goal requires us to know how to combine heterogeneous computations not only across the syntactic diamond, but also across the semantic one: the interpretation of a heterogeneous bind should commute into a heterogeneous bind of interpretations into distinct monads.

3 Monadic Programming

We set aside the free monad temporarily, and first focus on developing the right mathematical framework to painlessly program with monadic computations living in different structures. This amounts to binding together computations from different monads: a value $m : M X$ in a first monad M and a continuation $k : X \rightarrow N Y$ in a second monad N . In other words, we are

trying to define a practical structure with a heterogeneous bind operator with the following type, for several monads M , N , and T .

```
bind : ∀ X Y, M X → (X → N Y) → T Y
```

We furthermore ask this heterogeneous bind operator extends the usual monadic laws: `ret` should be a unit for `bind` on both sides, and `bind` should be associative.

3.1 Monad Morphisms to Transport

We start with a very simple observation: two computations can always be sequenced as soon as we know how to transport both arguments into a common monad. We hence request the existence of monad morphisms from M to T and from N to T and compose `bind` with the appropriate `fmaps`. Coq automatically infers the type of these `fmaps`, but for better readability, we annotate them in grey in the code.

```
Definition bindH M N T (MM : Monad M) (MN : Monad N) (MT : Monad T)
  (MMT : MonadMorphism M T) (MNT : MonadMorphism N T)
  : ∀ X Y, M X → (X → N Y) → T Y :=
  fun X Y m k => v ← fmapM↔T m;; fmapN↔T (k v).
```

Where `Monad M` is a type class constraint asserting the existence of the monadic operations over M , and `MonadMorphism N T` a type class constraint ensuring the existence of a morphism `fmapN↔T` from N to T . In the rest of this subsection, we omit these constraints to lighten the presentation.

By identifying all three monads and using the identity morphism, we show that the heterogeneous bind is indeed an extension of the usual monadic bind.

```
Lemma bindH_extends_bind : ∀ M X Y (m : M X) (k : X → M Y),
  bindHM X → (X → M Y) → M Y m k = bind m k.
```

Furthermore, the monadic laws remain valid, once appropriately generalized. The left and right `ret` laws simply have to manually `fmap` the unbound pure computations.

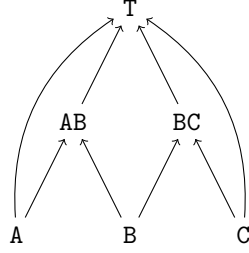
```
Lemma bindH_ret_l : ∀ M N T X Y (x : X) (k : X → N Y),
  bindHM X → (X → N Y) → T Y (ret x) k = fmapN↔T (k x).
```

```
Lemma bindH_ret_r : ∀ M N T X (c : M X),
  bindHM X → (X → N X) → T X c (fun x => ret x) = fmapM↔T c.
```

The associativity of the heterogeneous bind operator is a bit more of a mouthful. We need to consider six monads: three “base” monads A , B , and C for the arguments of the bind (`mx`, `kxy`, and `kzy`), one general monad into which the final result is computed, and two pivot monads AB and BC for the intermediary results. Figure 5 represents these structures, as well as the various morphisms each bind operator requires to state the lemma. We furthermore assume that the three subdiagrams of Figure 5 commute.

```
Lemma bindH_bindH : ∀ A B C AB BC T
  (commAB : ∀ X (m : A X), fmapAB↔T (fmapA↔AB m) = fmapA↔T m)
  (commBC : ∀ X (m : C X), fmapBC↔T (fmapC↔BC m) = fmapC↔T m)
  (commBT : ∀ X (m : B X), fmapAB↔T (fmapB↔AB m) = fmapBC↔T (fmapB↔BC m)),
  ∀ X Y Z (mx : A X) (kxy : X → B Y) (kzy : Y → C Z),
  bindH (bindH mx kxy) kzy = bindH mx (fun x => bindH (kxy x) kzy).
```

One may ponder: have we already solved our programming challenge, can we not write `main` using `bindH`? Well yes, but at great cost! Looking at the type of `bindH`, the unifying monad T


 Figure 5: Monads and their morphisms used to state `bindH_bindH`

```

Variable D : Type.
Variable reify : D → monad.
Notation "'[[ i ]]" := (reify i).
Hypothesis reify_into_monads : ∀ i, Monad [[i]].

Variable reify_le : ∀ i1 i2, i1 ⊆ i2 → MonadMorphism [[i1]] [[i2]].
Definition climbPO i1 i2 (I12 : i1 ⊆ i2) : ∀ X, [[i1]] X → [[i2]] X :=
  fmap (MonadMorphism := reify_le I12).

Hypothesis reify_le_trans :
∀ i1 i2 i3 (I12 : i1 ⊆ i2) (I13 : i1 ⊆ i3) (I23 : i2 ⊆ i3) X (m : [[i1]] X),
  climbPO I23 (climbPO I12 m) = climbPO I13 m.
    
```

Figure 6: Partial Order interface 🍷

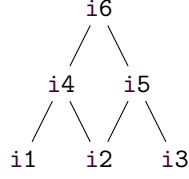
cannot be inferred from the type of the computations `m` and `k`: the programmer would essentially have to annotate it manually at every binding point. Furthermore, while the inference of the morphisms involved via instance declarations would be possible, this definition provides no safeguard to guide the programmer, they would have to foresee those they will need. We can do better.

3.2 A Partial Order to Index

While the `bind` operator from Section 3.1 captures semantically what we are looking for, its generality, working over arbitrary monads, leaves both our type checker and our programmer quite in the dark. Looking back at Figure 4, we had identified the constellation of monads at play starting from observing that we might work with some signatures $(\emptyset, \text{Rd}, \text{Wr}, \text{Rd} + \text{Wr})$, and that these signatures are naturally ordered by inclusion.

We follow this intuition by requesting the user to provide the interface described in Figure 6. We assume a domain of indices `D` equipped with a partial order \subseteq — for our running example, we build the four valued diamond $\{v, r, w, rw\}$. Indices are mapped to the monads of interest via a `reify` function (written `[[_]]`).

Intuitively, the ordering placed over our indices abstracts the ability to transport computations across monadic structures. We formalize this requisite in the `reify_le` field: we request the user to explicit how to map proofs of inequality between indices to monad morphisms between their respective reification. We define `climbPO` as the function associating the corresponding `fmap` to a given inequality. Finally, we must ensure some coherence: given two monads `M` and

Figure 7: Hasse Diagram of the order used in `bindPO_bindPO`

N , we intuitively want to consider at most one way to transport computations from M to N . In particular, the provided morphisms must be compatible with the associativity of the partial order, as spelled out in `reify_le_trans`.

Given such an interface, we can provide a specialization of Section 3.1’s operation: rather than provide a common target monad and morphisms, all we need is to pick an index above the two indexes involved in the computations being sequenced.

```

Definition bindPO i1 i2 i3 (LT1 : i1 ⊆ i3) (LT2 : i2 ⊆ i3) :
  ∀ X Y, [[i1]] X → (X → [[i2]] Y) → [[i3]] Y :=
  bindH (MMT := reify_le LT1) (MNT := reify_le LT2).

```

As before, the extended monadic laws remain valid. But where they used to depend on arbitrary morphisms, they are now specialized to ordered indices: everything relies on the canonical morphism specified by `reify_le`.

```

Lemma bindPO_ret_l : ∀ i1 i2 i3 (I13 : i1 ⊆ i3) (I23 : i2 ⊆ i3),
  ∀ X Y (k : X → [[i2]] Y) (x : X),
  bindPO I13 I23 (ret x) k = climbPO I23 (k x).

```

```

Lemma bindPO_ret_r : ∀ i1 i2 i3 (I13 : i1 ⊆ i3) (I23 : i2 ⊆ i3),
  ∀ X (m : [[i1]] X),
  bindPO I13 I23 m (ret (X := X)) = climbPO I13 m.

```

```

Lemma bindPO_bindPO : ∀ i1 i2 i3 i4 i5 i6,
  (I16 : i1 ⊆ i6) (I46 : i4 ⊆ i6) (I56 : i5 ⊆ i6) (I36 : i3 ⊆ i6)
  (I14 : i1 ⊆ i4) (I24 : i2 ⊆ i4) (I25 : i2 ⊆ i5) (I35 : i3 ⊆ i5),
  ∀ X Y Z (m : [[i1]] X) (k1 : X → [[i2]] Y) (k2 : Y → [[i3]] Z),
  bindPO I46 I36 (bindPO I14 I24 m k1) k2
  = bindPO I16 I56 m (fun x => bindPO I25 I35 (k1 x) k2).

```

As for `bindH_bindH`, the associativity law is based on a hierarchy of monads. But this time, their relation is expressed as an ordering of their indices, represented in Figure 7. Interestingly, the hypotheses of commutativity are now systematic consequences of `reify_le_trans`, and can therefore be omitted.

At its core, no work has of course been saved, all the morphisms at play must still be provided. But the necessary proof obligations are now cleanly identified, and the arguments to `bind` and to the monadic laws are now drawn from a much simpler structure.

We have shed some light on how to structure the constructions at hand, but have not addressed the other running issue: the unifying monad T from Section 3.1 has now become a unifying index $i3$, but it still needs to be provided. We remove this thorn in the side by adding slightly more structure to our indexing domain.

3.3 A Directed Set to Resolve Indecisiveness

The heterogeneous bind still has too much freedom: from having the luxury of picking the target monad \mathbb{T} of its choice, it can now pick its favourite target index $i3$. As a consequence, programmers have to either manually specify $i3$ for each call of `bindPO`, or to explicit the return type of each bind operation.

This freedom feels superfluous when coding: taking inspiration from our running example once again, it seems always safe to look at our index domain as having the structure of a semi-lattice and taking as target the join of the indices involved. In practice, picking the smallest upper bound does not matter for soundness, all we care about is to have a canonical way to chose a binary upper bound: we request the user to additionally equip their partial order with the structure of a directed set.

```
Variable join : D → D → D.
Infix "⊔" := join.
Hypothesis join_le_l : ∀ i1 i2, i1 ⊑ i1 ⊔ i2.
Hypothesis join_le_r : ∀ i1 i2, i2 ⊑ i1 ⊔ i2.
```

The heterogeneous bind specialises `bindPO` with the two inequalities about \sqcup . This time Coq can infer all type arguments from $(m : \llbracket i1 \rrbracket X)$ and $(k : X \rightarrow \llbracket i2 \rrbracket Y)$, discharging a lot of annotations from the programmer.

```
Definition bindL i1 i2 :
  ∀ X Y,  $\llbracket i1 \rrbracket X \rightarrow (X \rightarrow \llbracket i2 \rrbracket Y) \rightarrow \llbracket i1 \sqcup i2 \rrbracket Y :=$ 
  bindPO (join_le_l i1 i2) (join_le_r i1 i2).
```

The `main` function constituting our running example can thus be written with the desired syntax, with no additional notation, provided that the extended interface is provided. We furthermore observe that the user can provide a reification of the indexing diamond into either the free diamond (left part in Figure 4) or the semantic diamond (right part in Figure 4). Depending on their choice, the exact same syntax, depicted to the right will be valid against opaque computations `init` and `fetch` living in respectively `free Wr` and `free Rd`, or in `write` and `read`. Its return type will be directly inferred as respectively `free (Rd +' Wr)` or `state` from the joins involved.

```
Definition main (n : data) :=
  init n ;;
  v1 ← fetch ;;
  v2 ← fetch ;;
  ret (v1 =? v2).
```

We have overlooked one detail in the code above: the typing of `ret (v1 =? v2)` remains ambiguous. However this is easily resolved. By definition, `ret` performs no effect in the monads, hence if the lattice is equipped with a \perp element (typically reified into the `pure` monad), then it is safe to always use this element as an index for `ret`: we can thus redefine `ret` to always live in this “minimal” monad.

In order to guide the type inference when performing a bind, we have added a computational device, `join`, to compute the index governing the monadic type of the result. Pandora’s box is open, we are performing non-trivial dependent programming. The consequences show up the moment we turn our attention to the monadic laws for `bindL`. Things remain straightforward for the two `ret` laws:


```
Lemma bindL_ret_l : ∀ i1 i2,
  ∀ X Y (k : X →  $\llbracket i2 \rrbracket Y$ ) (x : X),
  bindL (ret (M :=  $\llbracket i1 \rrbracket$ ) x) k = climbPO (join_le_r i1 i2) (k x).
```

```
Lemma bindL_ret_r : ∀ i1 i2,
  ∀ X (m :  $\llbracket i1 \rrbracket X$ ),
  bindL m (ret (M :=  $\llbracket i2 \rrbracket$ )) = climbPO (join_le_l i1 i2) m.
```



But the obvious statement for associativity does not even type check!


```
Fail Lemma bindL_bindL_fail : ∀ i1 i2 i3,
  ∀ X Y Z (m :  $\llbracket i1 \rrbracket$  X) (k1 : X →  $\llbracket i2 \rrbracket$  Y) (k2 : Y →  $\llbracket i3 \rrbracket$  Z),
    bindL (bindL m k1) k2 = bindL m (fun x ⇒ bindL (k1 x) k2).
```

Indeed, the expression `bindL (bindL m k1) k2` has type $\llbracket (i1 \sqcup i2) \sqcup i3 \rrbracket Z$ while the expression `bindL m (fun x ⇒ bindL (k1 x) k2)` has type $\llbracket i1 \sqcup (i2 \sqcup i3) \rrbracket Z$. In other words, we need to assume the associativity of our domain of indices to express the associativity of our heterogeneous bind.

```
Hypothesis join_assoc : ∀ i1 i2 i3, i1  $\sqcup$  (i2  $\sqcup$  i3) = (i1  $\sqcup$  i2)  $\sqcup$  i3. 
```

Luckily, `climbP0` allows us to follow any inequality proof along our index domain. It can hence in particular be used to transport equality proofs in our types: by instantiating it over `join_assoc`, we get the `climb_assoc` transport function (we omit its code here) needed to state the bind associativity.

```
Definition climb_assoc : ∀ i1 i2 i3 :  $\llbracket i1 \sqcup (i2 \sqcup i3) \rrbracket \rightsquigarrow \llbracket (i1 \sqcup i2) \sqcup i3 \rrbracket$ . 
```

```
Lemma bindL_bindL : ∀ i1 i2 i3, 
  ∀ X Y Z (m :  $\llbracket i1 \rrbracket$  X) (k1 : X →  $\llbracket i2 \rrbracket$  Y) (k2 : Y →  $\llbracket i3 \rrbracket$  Z),
    bindL (bindL m k1) k2 = climb_assoc (bindL m (fun x ⇒ bindL (k1 x) k2)).
```

At this point, we have reached a first milestone: we have identified a suitable interface to program in a heterogeneous way across a set of monads organised as the reification of a pointed set. We now turn our attention back to the free monad.

4 Support for the Free Monad

In the previous section, we have seen that indexing monads by a directed set whose order maps to monad morphisms between the reified monad provides an adequate interface to support heterogeneous programming among those monads. Looking once again back at Figure 4, one gets the feeling that we did not quite strike the bullseye. First, we had not one but two structured sets of monads in mind — the syntactic world with the various free monads, and the semantic counterpart with their interpretations. Furthermore, signatures themselves seemed to play a specific role. In this section, we provide yet another interface, from which we will derive two instances of the directed one, respectively for the syntax and the semantics. The full interface is depicted on Figure 8: we motivate and walk through its components in this section. As an additional visual support, Figure 9 depicts the data contained in the interface when instantiated to our running example.

A first intuition could be to consider that signatures themselves should form the indexing structure. After all, the disjoint sum (+) could be a candidate for `join`, while signature inclusion, written `<` and defined by the existence of an injection between the signatures, would constitute a valid order. However, disjoint sum is much too crude a `join` — at the very least, we would like to look more closely to a set-theoretical join — and we therefore still need to introduce a proper pointed set `D`. But on the syntactic side, we are indeed indexing signatures rather than monads: this is the purpose of the part of Figure 8 describing the first layer of reification. In contrast to the previous case, two ordered indices must here map to reified signatures such that the smallest one can be injected into the largest one, as witnessed by `Sinj`. We impose three coherence properties on this first layer of reification: the injections should be proof irrelevant, proofs of self relation should map to the identity injection, and proofs by transitivity should map to the composition of their respective injections.

```

(* The directed domain by which we index *)
Variable D : Type.
Djoin_le_l : ∀ d1 d2, d1 ⊆ d1 ⊔ d2;
Djoin_le_r : ∀ d1 d2, d2 ⊆ d1 ⊔ d2;
Djoin_assoc : ∀ d1 d2 d3, d1 ⊔ (d2 ⊔ d3) = d1 ⊔ d2 ⊔ d3;

(* First layer of reification: in terms of signatures *)
Sreify : D → signature;
Notation "'[[ ' i ' ]]" := (Sreify i);
(* To the order must correspond injections *)
Sinj : ∀ d1 d2, d1 ⊆ d2 → [[d1]] -< [[d2]];
(* Coherence properties of these injections *)
Sinj_proper : ∀ d1 d2 (I12 I12' : d1 ⊆ d2), Sinj I12 = Sinj I12';
Sinj_refl : ∀ d (I : d ⊆ d), Sinj I = inject_id;
Sinj_trans : ∀ d1 d2 d3 (I12 : d1 ⊆ d2) (I23 : d2 ⊆ d3),
  (Sinj I12) ∘ (Sinj I23) = Sinj (PreOrder_Transitive _ _ _ I12 I23);

(* Via the free construction, D reifies into the corresponding free monads *)
Freify d := free [[d]];
Notation "'{ ' i ' }'" := (Freify i);
Ffmap d1 d2 (I : d1 ⊆ d2) : {d1} ∼ {d2} := fun m => translate (Sinj I) m;

(* Second layer of reification: in terms of concrete monads *)
Mreify : D → monad;
Notation "'<< ' i ' >>'" := (Mreify i).
Mmonad : ∀ d, Monad <<d>>;
(* To the order must correspond morphisms *)
Mmorph : ∀ d1 d2, d1 ⊆ d2 → MonadMorphism <<d1>> <<d2>>;
(* Coherence properties of these morphisms *)
Mfmap {d1 d2} (I : d1 ⊆ d2) X : <<d1>> ∼ <<d2>> :=
  fmap <<d1>> ∼ <<d2>> (MonadMorphism := Morph (Mmorph I)) X;
Mmorph_proper : ∀ d1 d2 (I J : d1 ⊆ d2), (Mfmap I) = (Mfmap J);
Mmorph_refl : ∀ d (I : d ⊆ d) X (m : <<d>> X), Mfmap I m = m;
Mmorph_trans : ∀ d1 d2 d3 (I12 : d1 ⊆ d2) (I23 : d2 ⊆ d3),
  (Mfmap I12) ∘ (Mfmap I23) = Mfmap (PreOrder_Transitive I12 I23);

(* Both layers are finally connected by handlers *)
Dh : ∀ d, [[d]] ∼ <<d>>;
I d : {d} ∼ <<d>> := interp (Dh d);
(* Climbing in the syntax and then interpreting or interpreting and then
climbing in the semantics should commute *)
I_commut : ∀ d1 d2 (I : d1 ⊆ d2) X (m : {d1} X),
  I (Ffmap I m) = Mfmap I (I m)

```

Figure 8: Interface for free monadic computations (🔥)

Turning our eye to Figure 9, we have so far covered the upper part of the figure. The indexing diamond maps to the one containing the four signatures introduced in Section 2 via `Sreify` (also noted `[[_]]`). By the `free` construction, signatures get mapped to all the free monads at which we might want to write programs — hence by composition, reification into signature induces a reification `Freify` (noted `{_}`) into free monads. Non-explicitly represented are the nature and relationship between the arrows in each of these three boxes: to a proof of inequality corresponds an injection of signatures, to which correspond a monad morphism between the corresponding free monads — the latter is derived for free.

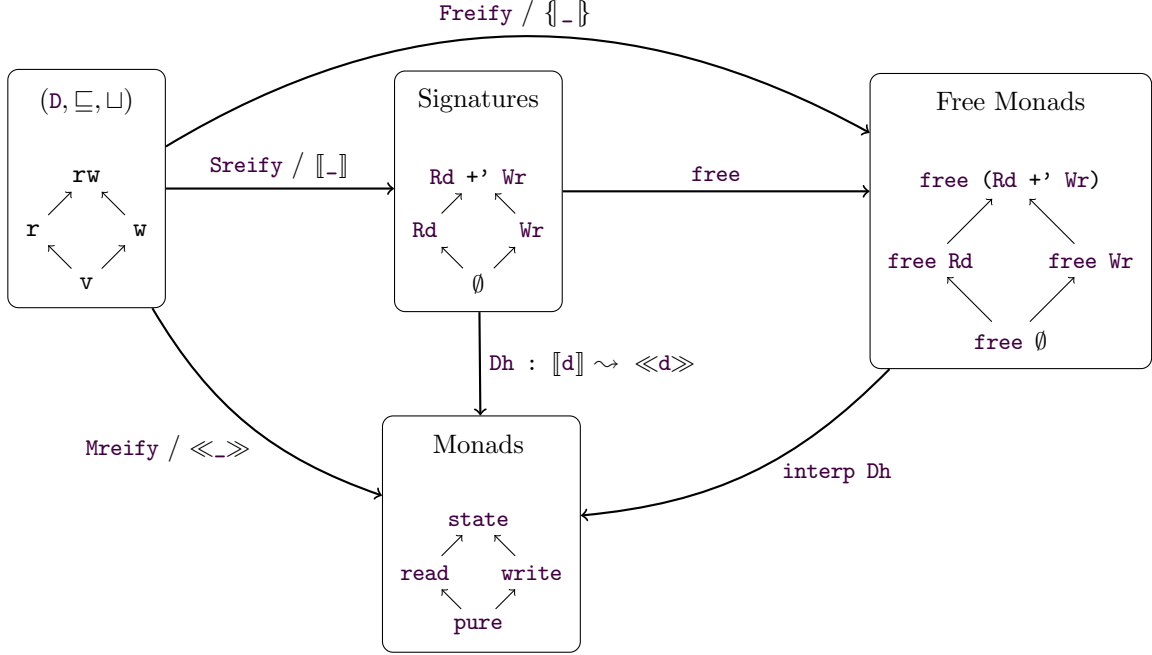


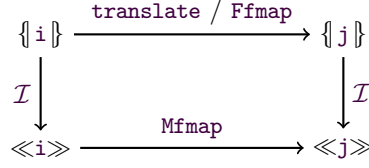
Figure 9: The full picture instantiated on our running example 🍷

We now bring into the picture the monads in which we concretely implement our effectful operations. This is the purpose of the second layer of reification from Figure 8: **Mreify** (also noted \ll_\gg) maps the same domain of indices D to concrete monads. This part of the interface is very close to the one developed in Section 3.3: indices must be reified into monads, and proofs of inequality must map to monad morphisms. The coherence properties are identical to the ones for the first layer of reification. On Figure 9, we have now introduced a fourth diamond, containing the state monad and its simplified versions, and the left arrow.


It remains only to relate the “free” diamond to the concrete one: we need to explain how we can implement the relevant signatures — this is captured by Dh . The user must provide at each index d a handler ($Dh\ d : \ll d \gg$): the **Rd** events must be implemented in the **read** monad, the **Rd +' Wr** ones in the state monad, and so on. Much like the reification into signatures gives rise to a reification into the free monads, this family of handlers entails a bridge \mathcal{I} between the diamond of free monads and the one of concrete monads via the **interp** function, completing the picture.

There remains one unstated coherence condition. Given two indices i and j such that $i \sqsubseteq j$, one may now follow two paths, represented in Figure 10. On one hand, we can translate the signature $\ll i \gg$ into $\ll j \gg$ (property Sin_j in Figure 8), and thus translate any tree in $\{i\}$ into $\{j\}$ —this is the role of the **translate** function in the definition of **Ffmap**. We can then interpret the resulting computation into $\ll j \gg$. On the other hand, we may interpret $\ll i \gg$ into $\ll i \gg$, and then transport $\ll i \gg$ into $\ll j \gg$. Condition $\mathcal{I}_{\text{commut}}$ ensures that this diagram commutes.

From an instance of the interface from Figure 8, we derive three essential facts. The first is an instance of the directed interface indexed by D w.r.t. **Freify**. The second is an instance of the directed interface indexed by D w.r.t. **Mreify**. We can hence sequence heterogeneous com-

Figure 10: Representation of the $\mathcal{I}_{\text{commut}}$ property

putations in both the syntactic and semantic world: let us write these operations respectively Fbind and Mbind . The last important result we derive is a theorem for commuting interpretation with binds in this melting pot:


Theorem $\text{bind}_{\mathcal{I}} : \forall d1\ d2\ X\ Y\ (m : \text{free } \llbracket d1 \rrbracket X) (k : X \rightarrow \text{free } \llbracket d2 \rrbracket Y),$ 
 $\mathcal{I} (\text{Fbind } m\ k) = \text{Mbind } (\mathcal{I}\ m) (\text{fun } x \Rightarrow \mathcal{I}(k\ x)).$

We have fulfilled the core of our contract by axiomatizing an interface allowing us to write the tightly typed version of `main` without additional crust. But even more importantly, it allows us to switch the perspective we have on the semantics of a compound computation, obtained by monadic interpretation, into the composition of the generally simpler implementation of its sub-components. Equipped with the minimal ingredients necessary to express and leverage effect typing information, we now sketch some early avenues to try and take advantage of it.

5 Reasoning

Sections 3 and 4 have depicted the core of our current contribution: a clean specification of the necessary interface to enable (1) programming in the free monad over a variety of signatures, (2) mapping each computation to an adequate structure of implementation, (3) retaining equational reasoning through notably generalized monadic laws and commutation of interpretation with heterogeneous binds. We now reach the current limits of this work in progress: the principled way to leverage typing information has yet to be identified, and no case study at sufficient scale has yet been performed to back up a particular ad-hoc approach. We hence only modestly present a few directions we have been exploring, and ornate them with preliminary remarks.

Recall Section 2's task: can we prove that `main n` will always return `true`. This could be specified extensionally, breaking the monadic abstraction: $\forall c, \text{snd } (\mathcal{I} (\text{call } n)) = \text{true}$. It seems more appealing to introduce a unary Hoare logic over `state`.

Definition `pre_state` := (cell → Prop). 
Definition `post_state X` := (X → cell → Prop).
Definition `Hoare_state X` : `pre_state` → `state X` → `post_state X` → Prop :=
`fun P m Q => ∀ c, P c → let '(c', x) := m c in Q x c'.`

Our goal hence become:

Theorem `main_correct` : $\forall (n : \text{nat}),$
`Hoare_state (fun _ => True) ($\mathcal{I} (\text{main } n)$) (fun v _ => v = true).`

Using $\text{bind}_{\mathcal{I}}$, the computation becomes explicitly the sequence of a computation in `write` with a computation in `read`. Using a cut rule for our Hoare logic, we can eliminate the former at the trivial postcondition, and are hence left with reasoning about the remaining read only computation. A quite adhoc way to conclude is to prove a general law of read only computations: when called twice in sequence, they lead to the same result.

```

Lemma read_twice (m : free Rd nat) :
  ∀ c, I(v1 ← m ;;
        v2 ← m ;;
        ret (v1 =? v2)) c = true.

```



This lemma breaks the abstraction introduced by `read`, but solves our goal immediately. It leverages our ability to keep track of the type of `fetch` to derive a generic invariant of the computation. From this perspective, it is a satisfying result, and the one we currently propose in our formal development.

In the remaining of this section, we discuss potential avenues to go further.

Finer grain invariants breaking abstraction. Although `read_twice` is generic over arbitrary computations (`m : free Rd nat`), it still fills handcrafted specifically for our problem. One may wonder whether the primitive fact expressing that read only computations may not affect the cell may suffice. Similarly, pure and write-only computations can be explicitly proved to not depend on the initial state.

```

Lemma read_invariant_explicit : ∀ X (m : read X) c,
  let (cf,x) := fmap read ~> state m c in c = cf.

```



```

Lemma write_invariant_explicit : ∀ X (m : write X) c c',
  let (cf,x) := fmap write ~> state m c in
  let (cf',x') := fmap write ~> state m c' in
  x = x' ∧ ((cf = c ∧ cf' = c') ∨ cf = cf').

```

```

Lemma pure_invariant_explicit : ∀ X (m : pure X) c c',
  let (cf,x) := fmap pure ~> state m c in
  let (cf',x') := fmap pure ~> state m c' in
  x = x' ∧ cf = c ∧ cf' = c'.

```

However, these invariants require to completely break the abstractions in order to be useful. On a simple example such as `main`, managing to reduce enough the computation to exhibit the sub-parts needed without simply computing the whole thing has proved unreasonably cumbersome for the expected benefit. It would nonetheless be interesting to see if it can be done, and may be worth it when working with coinductive computations such as the ones generated by ITrees, as (internal to Coq) reduction is not possible in this context.

Internalizing `read_state_invariant_explicit` into `hoare_state`. Rather than explicitly exposing the underlying computation, one could try to express the invariants as `state_hoare` triplets valid for any computation lifted from a given monad. Expressing that the state is left unchanged in this style is however quite annoying. One option is to change the postcondition so that it quantifies additionally over the initial state before the execution as well. Another is to use a meta-variable in order to bind as an equality in the precondition the initial value of the cell, and use the same meta-variable in the postcondition.

```

Lemma read_state_invariant : ∀ X v (m : read X),
  Hoare_state (fun c => c = v) (fmap read ~> state m) (fun _ c => c = v)

```



However, in both cases, there seems to be no obvious way to leverage such a rule to ensure that the second call to `fetch` would return the same value.

Furthermore, it is unclear how one would internalize in `state_hoare` the write only invariant `write_invariant_explicit`.

```

Definition post_pure X := (X → Prop).
Definition Hoare_pure X : pure X → post_pure X → Prop :=
  fun m Q ⇒ Q m.

Definition pre_read := (cell → Prop).
Definition post_read X := (X → Prop).
Definition Hoare_read X : pre_read → read X → post_read X → Prop :=
  fun P m Q ⇒ ∀ c, P c → let x := m c in Q x.

Definition post_write X := ((X → Prop) * (cell → Prop)).
Definition Hoare_write X : write X → post_write X → Prop :=
  fun m '(Qx, Qc) ⇒ let '(x, mc) := m in
    match mc with
    | None ⇒ Qx x
    | Some c ⇒ Qx x ∧ Qc c
  end.

```

Figure 11: Hoare-style proof system for pure, read only and write only monads

Hoare-style abstraction at each level. A strong temptation is to avoid breaking any abstraction. We have worked hard to preserve information about the structure in which each computation has been built, in a way compatible with equational reasoning. We would like to follow the usual approach and pair it cleanly with Hoare-style reasoning in the appropriate structure.

Naturally, this suggests that we should associate a Hoare-style proof system to each monad at play. For our running case study, we hence need a diamond of hoare systems: the missing components can be found in Figure 11. While more appealing, similar difficulties arise: for instance, read only computations seem to require a meta-variable to be lifted.

```

Lemma read_invariant : ∀ X v (m : read X) (P : pre_read) (Q : post_read X),
  Hoare_read P m Q →
  Hoare_state (fun c ⇒ P c ∧ c = v) (fmap read ~> state m) (fun x c ⇒ c = v ∧ Q x).

```

Furthermore, we are here working in an ad-hoc fashion, handcrafting proof systems for our case study. In order to state generic laws about these systems, and notably w.r.t. their interactions with `fmap`, we would need to move them to the interface. However identifying a universal shape to monadic deductive systems is non-trivial: it would likely have to rely on porting Maillard et al.'s work on Dijkstra monads [9].

We plan on conducting additional case studies at medium scale to better identify the kind of reasoning we may want to perform, and hopefully derive the adequate framework to do so.

6 Discussion and Future Work

The initial motivation for this work emerged from a Coq formalisation of R [1]. This formalisation is a large monadic interpreter of roughly 18,000 lines of code, with about two thousand `bind` applications. Proving some safety invariant within this formalisation (for instance memory safety) is doable, but requires a substantial proof effort due to the size of the interpreter: invariant lemmas have to be stated and proven for every function. Frustratingly, a lot of these lemmas felt obvious, as most functions only use a small subset of effects.

To illustrate this intuition, we categorised the interpreter functions involved in the formalization by the effects they used: Figure 12 shows the different categories that we identified.

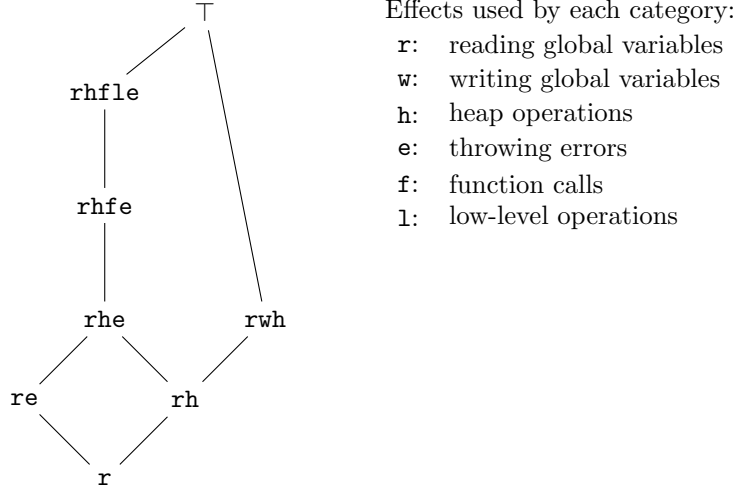


Figure 12: Hasse diagram for the different effects within a R monadic interpreter [1].

An interesting part of this diagram are the categories that are not shown: for instance, we did not find any interpreter functions that could raise an error (effect e) without reading the interpreter’s global variables (effect r), thus the re category is shown in the diagram, but not e . Almost all interpreter functions read these global variables, but only a small number write them: these are the functions initialising the interpreter at the very beginning of the execution, never to be called back again. In figure 12, these initialising functions fall into the rwh category. As a consequence, for most of the interpreter execution, these variables are constant by construction: no interpreter function syntactically call the write operations. The invariant that these global variables are constant after the initialisation phase thus ought to be simple to prove, but it turned out to be cumbersome in practice. We expect the \top category to be almost empty, with the sole exception of the main function that sets up the initialisation and calls the read-eval-print loop.

We believe that the ongoing work we present in this paper may help ease the points observed in this large scale project: the categories of Figure 12 fit well into the directed set of Section 3.3. We could thus imagine rewriting the interpreter within this formalism. As the approach presented in this work is based on a `bind` operation whose usage is very close to the usual monadic `bind`, we do not expect this conversion to require a significant amount of work. During type inference, Coq will compute where each interpreter function falls within the directed set. This enables a first phase of verification for the programmer: for instance, if a function is inferred to be of type \top (which in the context of R is expected to be very rare), something may be wrong and is worth inspecting. Furthermore, each of these categories provides some invariants: the sketches of methodology presented in Section 5 should be improved upon to cleanly state, prove, and leverage these invariants. In the case of this R interpreter, the size of the interpreter is large compared to the number of monads involved: we expect to get similar results to the ones presented in a cumbersome way in [1], but with less work.

References

- [1] Martin Bodin, Tomás Diaz, and Éric Tanter. A trustworthy mechanized formalization of R. In *Dynamic Languages Symposium, DLS*, 2018.
- [2] Venanzio Capretta. General recursion via coinductive types. *Log. Methods Comput. Sci.*, 1(2), 2005.
- [3] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice trees. *POPL*, 2023.
- [4] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In Mary S. Van Deusen and Bernard Lang, editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 71–84. ACM Press, 1993.
- [5] Mohsen Lesani, Li-yao Xia, Anders Kaseorg, Christian J. Bell, Adam Chlipala, Benjamin C. Pierce, and Steve Zdancewic. C4: Verified transactional objects. *Proc. ACM Program. Lang.*, 6(OOPSLA1), apr 2022.
- [6] Thomas Letan and Yann Régis-Gianas. Freespec: specifying, verifying, and executing impure computations in Coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020*, pages 32–46. ACM, 2020.
- [7] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effect handlers in Coq. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik P. de Vink, editors, *Formal Methods - 22nd International Symposium, FM 2018, Held as Part of the Federated Logic Conference, FloC 2018*, volume 10951 of *Lecture Notes in Computer Science*, pages 338–354. Springer, 2018.
- [8] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular verification of programs with effects and effect handlers in Coq. In *Formal Methods - 22nd International Symposium, FM 2018*, pages 338–354, 2018.
- [9] Kenji Maillard, Cătălin Hrițcu, Exequiel Rivas, and Antoine Van Muylder. The next 700 relational program logics. *POPL*, 4, 2020.
- [10] Conor McBride. Turing-completeness totally free. In *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*, pages 257–275, 2015.
- [11] Eugenio Moggi. *Computational Lambda-Calculus and Monads*. IEEE Computer Society Press, 1989.
- [12] Wouter Swierstra. Data types à la carte. *Journal of Functional Programming*, 18(4):423–436, 2008.
- [13] Philip Wadler. Monads for functional programming. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.
- [14] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees. *POPL*, 4, 2020.
- [15] Irene Yoon, Yannick Zakowski, and Steve Zdancewic. Formal reasoning about layered monadic interpreters. *Proc. ACM Program. Lang.*, 6(ICFP):254–282, 2022.
- [16] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. Modular, compositional, and executable formal semantics for LLVM IR. *ICFP*, 5, aug 2021.

Filtrer sans s'appauvrir : inférer les paramètres constants des modèles réactifs probabilistes

Guillaume Baudart,^{1,2} Grégoire Bussone,^{2,3}
Louis Mandel,⁴ et Christine Tasson³

¹ Inria Paris

² École normale supérieure – PSL university

³ Sorbonne Université

⁴ IBM Research

Résumé

ProbZelus étend le langage synchrone Zelus pour permettre de décrire des modèles probabilistes synchrones. Là où la programmation synchrone implémente des fonctions de suites, un langage probabiliste synchrone permet de décrire des modèles qui calculent des suites de distributions. On peut par exemple estimer la position d'un objet en mouvement à partir d'observations bruitées ou estimer l'incertitude d'un capteur à partir d'une suite d'observations. Ces problèmes mêlent des flots de variables aléatoires – les *paramètres d'état* qui changent au cours du temps – et des variables aléatoires constantes – les *paramètres constants*.

Pour estimer les paramètres d'état, les algorithmes d'inférence bayésienne de Monte Carlo séquentiels reposent sur des techniques de filtrage. Le filtrage est une méthode approchée qui consiste à perdre volontairement de l'information sur l'approximation actuelle pour recentrer les estimations futures autour de l'information la plus significative. Malheureusement, cette perte d'information est dommageable pour l'estimation des paramètres constants qui n'évoluent pas au cours du temps. Ce phénomène s'appelle l'appauvrissement.

Inspirés de la méthode d'inférence *Assumed Parameter Filter* (APF), nous proposons (1) une analyse statique, (2) une passe de compilation et (3) une nouvelle méthode d'inférence modulaire pour ProbZelus qui évite l'appauvrissement pour les paramètres constants tout en gardant les performances des algorithmes de Monte Carlo séquentiels pour les paramètres d'états.

1 Introduction

La programmation synchrone flot de données [5] est un paradigme dans lequel les fonctions, appelées nœuds, manipulent des suites infinies, appelées flots. Ces flots progressent au même rythme, de manière synchrone. L'expressivité des langages synchrones est volontairement restreinte, ce qui permet à des compilateurs spécialisés de générer du code efficace et correct par construction avec de fortes garanties sur le temps d'exécution et sur l'utilisation de la mémoire [19]. Le langage industriel Scade est notamment utilisé pour la conception de systèmes embarqués critiques [13].

Les langages probabilistes [6, 14, 17, 18] étendent des langages de programmation généralistes avec des constructions probabilistes. Suivant la méthode bayésienne, un *modèle* probabiliste décrit une distribution de probabilité (la distribution *a posteriori*) à partir d'une croyance initiale (la distribution *a priori*) et d'observations concrètes (les entrées).

Dans la même lignée de langages, ProbZelus [2] est une extension probabiliste du langage synchrone flot de données Zelus [8]. Ce langage permet de décrire des modèles réactifs probabilistes. Là où la programmation synchrone implémente des fonctions de suites, un langage probabiliste synchrone permet de décrire des modèles qui calculent des suites de distributions.

Un exemple est le problème de localisation et cartographie simultanées (SLAM pour *Simultaneous Localization and Mapping*) [20]. Un robot cherche à inférer à la fois sa position et une carte de son environnement. La position est un *paramètre d'état* représenté par un flot de variables aléatoires. À chaque instant, une nouvelle position doit être estimée à partir de la position précédente et des nouvelles observations. La carte est un *paramètre constant* représenté par une variable aléatoire dont la valeur est progressivement affinée à chaque nouvelle observation à partir d'une croyance initiale, la distribution *a priori*. Ce type de problème qui mêle paramètres constants et paramètres d'état qui changent au cours du temps est appelé *State-Space Models* (SSM) [10]. Tout programme probabiliste synchrone peut s'exprimer comme un SSM.

Pour estimer les paramètres d'état, les algorithmes d'inférence de Monte Carlo séquentiels (SMC pour *Sequential Monte Carlo*) reposent sur des techniques de filtrage [10,15]. Le filtrage est une méthode approchée qui consiste à perdre volontairement de l'information sur l'approximation actuelle pour recentrer les estimations futures autour de l'information la plus significative. Ces méthodes sont donc particulièrement bien adaptées au contexte réactif où un système en interaction avec son environnement ne s'arrête jamais et doit donc s'exécuter avec des ressources bornées. Toutes les méthodes d'inférence de ProbZelus appartiennent à cette famille [1–3]. Malheureusement, cette perte d'information est dommageable pour l'estimation des paramètres constants qui n'évoluent pas au cours du temps. Ce phénomène s'appelle l'appauvrissement.

Pour éviter ce problème, l'algorithme d'inférence *Assumed Parameter Filter* (APF) [16] propose, à chaque instant, de décomposer l'inférence en deux temps : (1) estimation des paramètres d'état, et (2) mise à jour des paramètres constants. Cette méthode suppose que les paramètres constants sont bien identifiés et que le modèle est écrit sous une forme qui permet de mettre à jour les distributions des paramètres constants sachant la valeur des paramètres d'état.

Dans cet article, nous proposons une nouvelle méthode d'inférence fondée sur APF pour ProbZelus. Nous présentons en particulier les contributions suivantes : Section 3 une analyse statique pour identifier les paramètres constants, Section 4 une passe de compilation pour générer un modèle exploitable par APF, Section 5 une implémentation de l'algorithme APF pour les modèles ProbZelus. Nous évaluons cette nouvelle méthode d'inférence sur un ensemble de modèles ProbZelus. Le code et une annexe contenant la preuve du théorème principal sont disponibles à l'adresse suivante : <https://github.com/rpl-lab/jfla23-apf>.

2 Motivation

Pour motiver notre approche, considérons l'exemple d'un radar adapté de [10, Section 2.4.1] illustré Figure 1. On cherche à estimer la position x_t d'un bateau au cours du temps à partir d'une mesure de distance (par exemple à l'aide d'un sonar), et d'une mesure angulaire. On suppose que le bateau dérive à vitesse constante θ et on cherche également à estimer cette vitesse. Ce modèle est donc un *State-Space Model* (SSM). On s'intéresse à la distribution d'un paramètre d'état (le flot de variables aléatoires x_t) et d'un paramètre constant (la variable aléatoire θ).

2.1 Un modèle de radar en ProbZelus

Un premier modèle probabiliste décrit le mouvement du bateau, c'est-à-dire à chaque instant $t > 0$, la distribution *a priori* des variables θ et x_t en fonction de la position précédente x_{t-1} .

```
proba move x0 = theta, x where
  rec init theta = sample(mv_gaussian(zeros, i2))
  and x = x0 → sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
```

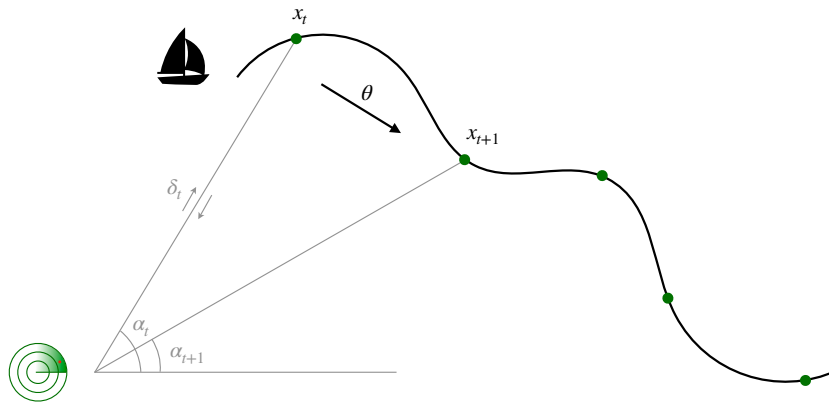


FIGURE 1 – Le radar cherche à estimer la position courante du bateau x_t et sa vitesse de dérive constante θ à partir du délai de rebond d'un sonar δ_t et d'une observation angulaire α_t .

Le mot-clé `proba` introduit un modèle probabiliste appelé `move` qui prend en argument la position initiale x_0 et renvoie, à chaque instant, la vitesse `theta` et la position courante x . Les variables x_0 , `theta`, et x sont des vecteurs à deux dimensions dans \mathbb{R}^2 . Le paramètre constant `theta` est introduit par le mot-clé `init`. Ici, on suppose *a priori*, que `theta` est échantillonnée dans une gaussienne multivariée centrée sur $(0, 0)^T$ et de covariance I_2 , la matrice identité de dimension 2 (on suppose que les deux composantes de la vitesse ne sont pas corrélées). La définition de la position x utilise les opérateurs d'initialisation \rightarrow et de délai unitaire `pre`. Au premier instant, $x_0 = x_0$. Puis, aux instants $t > 0$ suivants, x_t est échantillonnée dans une gaussienne centrée sur la valeur précédente x_{t-1} traduite de `theta` et de covariance $0.5 * I_2$. Le paramètre d'état x est donc un flot de variables aléatoires.

Le modèle de radar suivant définit, à chaque instant, la distribution *a posteriori* de la vitesse du bateau et de sa position à partir des distributions *a priori* générées par `move`, et de deux *observations* bruitées : `delta` le délai de rebond du sonar et `alpha` la mesure angulaire.

```
proba radar (delta, alpha) = theta, x where
  rec theta, x = move x0
  and d = sqrt((vec_get x 0) ** 2. +. (vec_get x 1) ** 2.)
  and a = atan(vec_get x 1 /. vec_get x 0)
  and () = observe(gaussian(2. * d /. c, delta_noise), delta)
  and () = observe(gaussian(a, alpha_noise), alpha)
```

À partir de la vitesse `theta` et de la position courante $x = (x_1, x_2)^T$ estimée, on calcule la distance $d = \sqrt{x_1^2 + x_2^2}$, et l'angle $a = \text{atan}(x_2/x_1)$. La construction `observe` permet ensuite de contraindre le modèle en supposant (1) que le temps de rebond observé `delta` a été échantillonné dans une gaussienne centrée sur $2d/c$ (où c est la vitesse du signal envoyé par le sonar), et (2) que la mesure angulaire `angle` a été échantillonnée dans une gaussienne centrée sur a .

Finalement, le nœud `main` prend en argument les flots d'observations `delta` et `alpha` et lance l'inférence sur le modèle `radar` pour calculer à chaque instant une approximation `d_theta` de la distribution de la vitesse `theta` et une approximation `d_x` de la distribution de la position x .

```
node main (delta, alpha) = d_theta, d_x where
  rec d = infer radar (delta, alpha)
  and d_theta, d_x = Dist.split d
```

Données : le modèle `model`, l'entrée y_t et le résultat précédent μ_{t-1} .

Résultat : μ_t une approximation de la distribution de p_t .

pour chaque particule $i = 1$ à N **faire**

```

|    $p_{t-1}^i = \text{sample}(\mu_{t-1})$ 
|    $p_t^i, w_t^i = \text{model}(y_t \mid p_{t-1}^i)$ 
 $\mu_t = \mathcal{M}(\{w_t^i, p_t^i\}_{1 \leq i \leq N})$ 
renvoyer  $\mu_t$ 

```

Algorithme 1 : Filtre particulaire.

Le mot-clé `node` introduit un nœud déterministe classique. La fonction `Dist.split` sépare une distribution sur des paires (ici `theta`, et `x`) en une paire de distributions. On peut ensuite analyser ces distributions en calculant moyennes, quantiles, etc. ou en affichant un nuage d'échantillons tirés dans la distribution comme dans la Figure 2.

2.2 Le filtrage et ses limites

Les méthodes d'inférence proposées par ProbZelus [1–3] appartiennent à la famille des algorithmes de Monte Carlo séquentiels [10, 15].

Ces méthodes reposent sur un ensemble de simulations indépendantes, appelées *particules*. Chaque particule renvoie une valeur de sortie associée à un score. Le score représente la qualité de la simulation. Un grand nombre de particules permet d'approximer la distribution recherchée. La construction `sample(d)` tire aléatoirement une valeur dans la distribution `d`, et la construction `observe(d, x)` modifie le score courant de la particule en lui ajoutant la *vraisemblance* de l'observation `x` par rapport à la distribution `d`. À chaque instant, l'opérateur `infer` accumule les valeurs calculées par chaque particule pondérées par leurs scores pour approximer la distribution *a posteriori*.

Filtre particulaire. Si le modèle fait appel à l'opérateur `sample` à chaque instant, par exemple pour estimer la position du bateau, la méthode précédente implémente une simple marche aléatoire pour chaque particule. À mesure que le temps avance, il devient de plus en plus improbable qu'une des marches aléatoires coïncide avec le flot d'observations. Le score associé à chaque particule dégringole rapidement vers 0.

Pour régler ce problème, les méthodes de Monte Carlo séquentielles ajoutent une étape de filtrage. L'Algorithme 1 décrit ainsi l'exécution d'un instant pour un filtre particulaire. À chaque instant t , une particule $1 \leq i \leq N$ correspond à une valeur possible des paramètres (*i.e.*, variables aléatoires) p_t^i du modèle. On commence par échantillonner un nouvel ensemble de particules dans la distribution obtenue à l'instant précédent. Les particules les plus probables sont ainsi dupliquées et les moins probables sont éliminées. On recentre ainsi l'inférence autour de l'information la plus significative tout en conservant le même nombre de particules tout au long de l'exécution. Sachant l'état précédent p_{t-1}^i , chaque particule exécute ensuite un pas du modèle pour obtenir un échantillon des paramètres p_t^i associé à un score w_t^i . À la fin de l'instant, on construit une distribution μ_t où chaque particule est associée à son score. $\mathcal{M}(\{w_t^i, p_t^i\}_{1 \leq i \leq N})$ est une distribution multinomiale, où la valeur p_t^i est associée à la probabilité w_t^i .¹

Appauvrissement. Malheureusement, cette approche engendre une perte d'information dommageable pour l'estimation de paramètres constants. Sur l'exemple du radar, au premier

1. On note $\bar{w}_t^i = w_t^i / \sum_{i=1}^N w_t^i$ les scores normalisés.

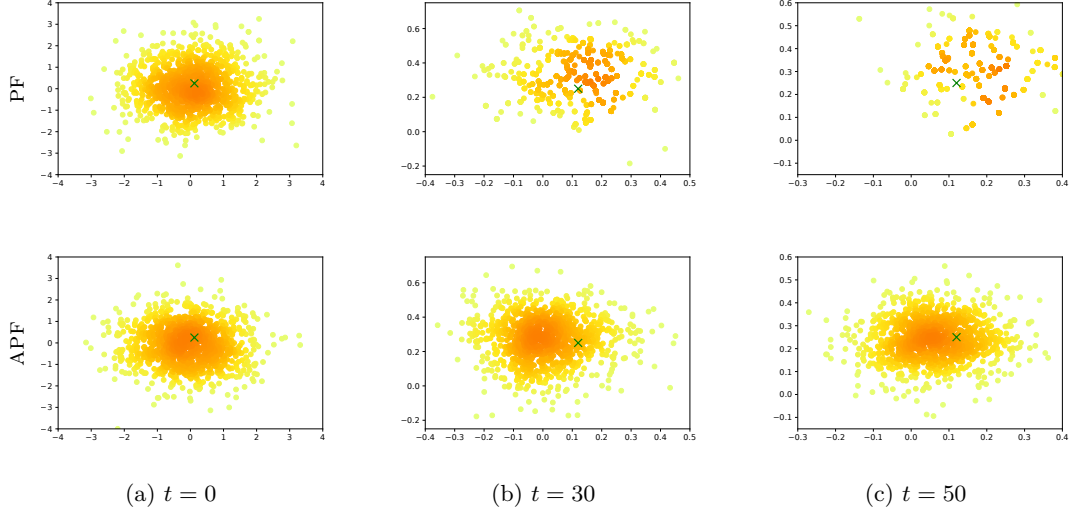


FIGURE 2 – Estimations du paramètre θ de l'exemple du radar au cours du temps avec un filtre particulaire (PF, en haut) et *Assumed Parameter Filter* (APF, en bas). La vraie vitesse de dérive est indiquée par une croix verte. Le gradient de couleur représente la densité de points. L'échelle change au cours du temps. Les résultats peuvent différer d'une exécution à l'autre.

instant, chaque particule tire une valeur aléatoire pour le paramètre θ . À chaque instant, les particules dupliquées partagent la même valeur pour θ . La quantité d'information utile pour estimer θ décroît donc à chaque nouveau filtrage et, au bout d'un certain temps, il ne reste plus qu'une seule valeur possible. La partie haute de la Figure 2 illustre graphiquement ce phénomène.

Plus formellement, on peut montrer que le filtre particulaire souffre d'appauvrissement sur les paramètres constants quel que soit le modèle.

Proposition 1 (Appauvrissement). *Considérons un filtre particulaire avec $N \in \mathbb{N}^*$ particules pour estimer un paramètre constant θ à valeur dans V . À chaque instant, on note $\Theta_t \in V^N$ l'estimation obtenue à l'instant $t \in \mathbb{N}$, par définition un ensemble de valeurs possibles pour θ . Il existe presque sûrement un instant $T \in \mathbb{N}$ tel que $\forall t > T, |\Theta_t| = 1$.*

Démonstration. On note à chaque instant $t \in \mathbb{N}$, $\Theta_t = \{\theta_t^i\}_{1 \leq i \leq N}$ l'ensemble des valeurs possibles et $w(\theta_t^i) \in]0, \infty[$ le score associé à la valeur θ_t^i . À chaque filtrage, θ_t^i est dupliquée avec la probabilité $w(\theta_t^i)$ (le score normalisé de θ_t^i). Pour $t \in \mathbb{N}$, s'il reste au moins deux particules distinctes $|\Theta_t| \geq 2$, on note A la valeur de Θ_t associée au score minimal qui a donc le plus de chances d'être éliminée : $A = \operatorname{argmin}_{a \in \Theta_t} \sum_{\theta_t^i = a} w(\theta_t^i)$. On a donc $\forall 1 \leq i \leq N, \mathbb{P}(\theta_{t+1}^i = A) \leq \frac{1}{2}$ et :

$$\mathbb{P}(\exists a. a \in \Theta_t \text{ et } a \notin \Theta_{t+1}) \geq \mathbb{P}(A \notin \Theta_{t+1}) = \prod_{i=1}^N \mathbb{P}(\theta_{t+1}^i \neq A) \geq \frac{1}{2^N} = \epsilon > 0.$$

Comme ϵ ne dépend que de N , $\forall n \in \mathbb{N}$, on a : $\mathbb{P}(\Theta_{t+n} = \Theta_t) \leq (1 - \epsilon)^n$. Il existe donc presque sûrement $t' > t$ tel que $\Theta_{t'} \subsetneq \Theta_t$. Par récurrence sur la taille de Θ_0 , il existe presque sûrement $T \in \mathbb{N}$ tel que $|\Theta_T| = 1$. \square

Données : le modèle `model`, l'entrée y_t et le résultat précédent μ_{t-1} .

Résultat : μ_t une approximation de la distribution de x_t (état) et θ (constant).

pour chaque particule $i = 1$ à N faire

```

|  $x_{t-1}^i, \Theta_{t-1}^i = \text{sample}(\mu_{t-1})$ 
|  $\theta^i = \text{sample}(\Theta_{t-1}^i)$ 
|  $x_t^i, w_t^i = \text{model}(y_t \mid \theta^i, x_{t-1}^i)$ 
|  $\Theta_t^i = \text{Udpate}(\Theta_{t-1}^i, \lambda \theta, \text{model}(y_t \mid \theta, x_{t-1}^i, x_t^i))$ 
 $\mu_t = \mathcal{M}(\{w_t^i, (x_t^i, \Theta_t^i)\}_{1 \leq i \leq N})$ 
renvoyer  $\mu_t$ 

```

Algorithme 2 : *Assumed Parameter Filter* [16]

2.3 Assumed Parameter Filter

Plutôt que d'échantillonner au début de l'exécution un ensemble de valeurs pour les paramètres constants qui s'appauvrira à chaque filtrage, dans l'algorithme *Assumed Parameter Filter* (APF) [16], chaque particule calcule une distribution symbolique de paramètres constants. Lors de l'exécution, l'inférence alterne ensuite entre une passe d'échantillonnage pour estimer les paramètres d'état, et une passe d'optimisation qui permet de mettre à jour les paramètres constants. On évite ainsi l'appauvrissement pour l'estimation des paramètres constants. La partie basse de la Figure 2 illustre les résultats de APF sur l'exemple du radar.

Plus formellement, l'Algorithme 2 décrit l'exécution d'un instant de APF. À chaque instant t , une particule $1 \leq i \leq N$ correspond maintenant à une valeur possible des paramètres d'état x_t^i et une distribution de paramètres constants Θ_t^i . Comme pour le filtre particulaire, on commence par échantillonner un ensemble de particules dans la distribution obtenue à l'instant précédent. On échantillonne ensuite une valeur θ^i dans Θ_{t-1}^i . Sachant la valeur des paramètres constants θ^i et l'état précédent x_{t-1}^i , on peut exécuter un pas du modèle pour obtenir un échantillon des paramètres d'état x_t^i associé à un score w_t^i . Il reste alors à calculer Θ_t^i en explorant les autres valeurs possibles pour θ sachant que la particule a choisi la transition $x_{t-1}^i \rightarrow x_t^i$. À la fin de l'instant, comme pour le filtre particulaire de l'Algorithme 1, on construit une distribution μ_t où chaque particule est associée à son score.

La difficulté principale est qu'à chaque instant, l'Algorithme 2 doit ré-exécuter plusieurs fois le modèle en fixant la valeur de certaines variables aléatoires. D'abord pour calculer les paramètres d'état sachant la valeur échantillonnée θ^i , puis pour mettre à jour la distribution Θ_t^i en explorant plusieurs valeurs possibles pour θ , sachant la transition $x_{t-1}^i \rightarrow x_t^i$.

Pour implémenter APF pour des modèles ProbZelus arbitraires, nous proposons donc une analyse statique pour identifier les paramètres constants et leur distributions *a priori* (Section 3) et une passe de compilation qui transforme ces paramètres en entrée supplémentaire du modèle (Section 4).

3 Analyse Statique

Dans cette section, nous présentons un noyau de ProbZelus. L'ensemble du langage, y compris les structures de contrôle de haut niveau comme les automates hiérarchiques, peut être compilé vers ce noyau [12]. Nous présentons ensuite l'analyse statique qui permet d'identifier les paramètres constants et leur distribution *a priori* sur ce noyau.

```

d ::= let x = e | node f x = e | proba f x = e | d d
e ::= c | x | (e, e) | op(e) | fα(e) | last x
    | e where rec init x = e and ... and init x = e and p = e and ... and p = e
    | present e -> e else e | reset e every e | sample(e) | observe(e, e) | infer(f(e))

```

FIGURE 3 – Syntaxe du noyau de ProbZelus

3.1 Un noyau de ProbZelus

La syntaxe du noyau de ProbZelus est présentée dans la Figure 3. Un programme est une série de déclarations d . Une déclaration peut donner un nom au résultat d'une expression (`let x = e`) ou définir un nœud déterministe (`node f x = e`) ou probabiliste (`proba f x = e`). Le nom de chaque déclaration est unique. Une expression peut être une constante (c), une variable (x), une paire $((e, e))$, l'application d'une primitive ($op(e)$), un appel de nœud ($f^\alpha(e)$), un délai (`last x`), un ensemble d'équations locales mutuellement récursives (`e where rec E`), une structure de contrôle (`present e -> e else e`), une structure de réinitialisation (`reset e every e`) ou une construction probabiliste (`sample(e)`, `observe(e, e)` ou `infer(f(e))`).

On impose que toutes les variables initialisées soient définies par une équation, quitte à ajouter `x = last x`, et que les équations soient ordonnancées. Dans une expression de la forme `e where rec (init xi = ei)1 ≤ i ≤ k and (yj = e'j)1 ≤ j ≤ n`, on a donc $\{x_i\}_{1 \leq i \leq k} \subset \{y_j\}_{1 \leq j \leq n}$ et e'_j ne peut dépendre de $y_{j'}$ avec $j \leq j'$ qu'à travers l'opérateur `last`.

Ce noyau est proche de celui utilisé pour définir la sémantique formelle de ProbZelus [2]. La différence principale est qu'il est possible d'initialiser une variable avec une expression arbitraire là où le noyau original limitait la construction `init` à des constantes : `init x = c`. On autorise ainsi `init x = sample(d) and x = last x` pour définir des paramètres constants. Par ailleurs, on autorise la définition des variables globales dans les déclarations. Enfin, l'appel de nœud $f^\alpha(e)$ est ici annoté avec un nom d'instance α ajouté automatiquement par le compilateur qui permet de différencier les appels à une même fonction. Par exemple, l'expression $f(0) + f(1)$ devient dans ce noyau $f^{\alpha_1}(0) + f^{\alpha_2}(1)$.

Sémantique. La sémantique formelle de ProbZelus est définie dans [2]. Dans un environnement γ qui associe les noms de variables à leur valeur, les expressions déterministes sont des machines à états caractérisées par un état initial $\llbracket e \rrbracket_\gamma^{\text{init}}$ de type S et une fonction de transition $\llbracket e \rrbracket_\gamma^{\text{step}}$ de type $S \rightarrow O \times S$. La fonction de transition prend en entrée un état et renvoie une sortie et l'état suivant. On obtient un flot de données en itérant la fonction de transition depuis l'état initial.

Les expressions probabilistes sont elles aussi caractérisées par un état initial de type S , mais la fonction de transition $\llbracket e \rrbracket_\gamma^{\text{step}}$ de type $S \rightarrow \Sigma_{O \times S} \rightarrow [0, \infty)$ renvoie une mesure qui associe un score positif à chaque ensemble mesurable de paires (sortie, état suivant) $\in \Sigma_{O \times S}$.

3.2 Identifier les paramètres constants

L'analyse statique est définie par un jugement $\emptyset, \emptyset \vdash prog : H, C$ qui, pour un programme $prog$, construit un environnement H associant à chaque nœud probabiliste un type ξ qui identifie les paramètres constants du nœud et leur distribution *a priori* (l'environnement C contient les variables globales constantes). Pour un nœud `proba f x = e`, le type ξ est une fonction telle que chaque variable x dans $dom(\xi)$ est un paramètre constant dans e de distribution *a priori* $\xi(x)$.

$$\begin{array}{c}
\frac{}{C \vdash c : \emptyset} \quad \frac{}{C \vdash x : \emptyset} \quad \frac{C \vdash e_1 : \xi_1 \quad C \vdash e_2 : \xi_2}{C \vdash (e_1, e_2) : \xi_1 \cup \xi_2} \quad \frac{C \vdash e : \xi}{C \vdash \text{op}(e) : \xi} \\
\\
\frac{C \vdash e : \xi}{C \vdash f^\alpha(e) : \xi \cup \{\alpha \leftarrow f_prior\}} \quad \frac{}{C \vdash \text{last } x : \emptyset} \quad \frac{C \vdash e_1 : \xi}{C \vdash \text{present } e_1 \rightarrow e_2 \text{ else } e_3 : \xi} \\
\\
\frac{C \vdash e_2 : \xi}{C \vdash \text{reset } e_1 \text{ every } e_2 : \xi} \quad \frac{C \vdash e : \xi}{C \vdash \text{sample}(e) : \xi} \quad \frac{C \vdash e_1 : \xi_1 \quad C \vdash e_2 : \xi_2}{C \vdash \text{observe}(e_1, e_2) : \xi_1 \cup \xi_2} \\
\\
\frac{C \vdash e : \xi \quad \forall 1 \leq i \leq k, C \vdash e_i : \xi_i \quad \forall 1 \leq j \leq n, C \vdash e'_j : \xi'_j \quad \xi' = \{x_i \leftarrow d_i \mid x_i = y_j \wedge e'_j = \text{last } y_j \wedge \vdash^p e_i : d_i \wedge C \vdash^l d_i\}}{C \vdash e \text{ where rec } (\text{init } x_i = e_i)_{1 \leq i \leq k} \text{ and } (y_j = e'_j)_{1 \leq j \leq n} : \xi \cup (\cup_{i=1}^k \xi_i) \cup (\cup_{j=1}^n \xi'_j) \cup \xi'} \\
\\
\frac{C \vdash e : \xi}{H, C \vdash \text{proba } f \ x = e : H \cup \{f \leftarrow \xi\}, C} \quad \frac{}{H, C \vdash \text{node } f \ x = e : H \cup \{f \leftarrow \emptyset\}, C} \\
\\
\frac{C \vdash^l e}{H, C \vdash \text{let } x = e : H, C \cup \{x\}} \quad \frac{H, C \vdash d_1 : H_1, C_1 \quad H_1, C_1 \vdash d_2 : H', C'}{H, C \vdash d_1 \ d_2 : H', C'}
\end{array}$$

FIGURE 4 – Identification des paramètres constants avec leurs distributions *a priori*.

$$\frac{}{\vdash^p \text{sample}(d) : d} \quad \frac{\vdash^p e : d}{\vdash^p e \text{ where rec } E : d \text{ where rec } E}$$

FIGURE 5 – Distribution échantillonnée.

Le jugement de typage des expressions est de la forme $C \vdash e : \xi$ et repose sur deux jugements auxiliaires : le jugement $\vdash^p e : d$ qui extrait la distribution d échantillonnée par e , et le jugement $C \vdash^l e$ qui garantit que l'expression e est constante et ne dépend que des variables constantes définies dans l'ensemble C .

Paramètres constants. Le jugement principal $C \vdash e : \xi$ est défini Figure 4. Pour simplifier l'analyse, on suppose que toutes les variables et les instances ont des noms différents. Une passe de renommage des variables permet facilement de satisfaire cette précondition. Toutes les règles parcourent les sous-expressions pour collecter les paramètres constants. La règle pour `reset` e_1 `every` e_2 parcourt uniquement e_2 car l'expression e_1 peut être réinitialisée et donc n'est pas constante. Pour une raison similaire, la règle pour `present` $e_1 \rightarrow e_2$ `else` e_3 ne parcourt que e_1 . La règle pour $f^\alpha(e)$ associe dans ξ le nom d'instance α avec la distribution *a priori* de f que l'on suppose définie dans la variable `f_prior` (qui sera ajoutée par la compilation). La règle pour l'expression e `where rec` $(\text{init } x_i = e_i)_{1 \leq i \leq k}$ `and` $(y_j = e'_j)_{1 \leq j \leq n}$ ajoute dans ξ les variables x_i qui respectent les conditions suivantes.

1. Elle n'est pas modifiée par la fonction de transition $(x_i = y_j \wedge e'_j = \text{last } y_j)$.
2. Elle échantillonne une distribution d_i ($\vdash^p e_i : d_i$).
3. La distribution d_i est constante et ne dépend que des constantes dans C ($C \vdash^l d_i$).

$$\begin{array}{c}
\frac{}{C \vdash^l c} \quad \frac{x \in C}{C \vdash^l x} \quad \frac{C \vdash^l e_1 \quad C \vdash^l e_2}{C \vdash^l (e_1, e_2)} \quad \frac{C \vdash^l e}{C \vdash^l \text{op}(e)} \quad \frac{C \vdash^l e_1 \quad C \vdash^l e_2 \quad C \vdash^l e_3}{C \vdash^l \text{present } e_1 \rightarrow e_2 \text{ else } e_3} \\
\\
\frac{C \vdash^l e_1 \quad C \vdash^l e_2}{C \vdash^l \text{reset } e_1 \text{ every } e_2} \quad \frac{C \cup \{x_i\}_{1 \leq i \leq k} \vdash^l e \quad C \cup \{x_j\}_{1 \leq j < i} \vdash^l e_i}{C \vdash^l e \text{ where rec (init } x_i = e_i)_{1 \leq i \leq k} \text{ and } (x_j = \text{last } x_j)_{1 \leq j \leq k}}
\end{array}$$

FIGURE 6 – Expressions constantes.

L'analyse d'un nœud probabiliste f type le corps e du nœud et ajoute ce type dans l'environnement H associé au nom f . Les nœuds déterministes n'ont pas de variables aléatoires par définition. Les déclarations globales doivent être constantes, leurs noms sont ajoutés dans l'ensemble C des variables constantes.

Distributions. Le jugement $\vdash^p e : d$, défini Figure 5, garantit que l'expression e échantillonne une distribution d . La règle principale est donc celle de `sample`(d) qui renvoie la distribution d . La règle pour `where rec` autorise l'utilisation de variables locales pour définir la distribution échantillonnée.

Constantes. Le jugement $C \vdash^l e$, défini Figure 6, garantit que l'expression e est constante en vérifiant que toutes les sous-expressions sont constantes et ne dépendent que de variables constantes.

Exemple. Illustrons l'analyse statique sur l'exemple du nœud `move` de la Section 2. L'équation `theta = last theta` est ajoutée automatiquement par le compilateur.

```

proba move x0 = theta, x where
  rec init theta = sample(mv_gaussian(zeros, i2))
  and x = x0 → sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))
  and theta = last theta

```

La règle `proba` type le corps du nœud en utilisant la règle `where rec`. Le typage de chaque sous-expression renvoie un ensemble vide (aucun paramètre constant local n'est défini). Le système détecte que `theta` est un flot constant (`init theta = ... and theta = last theta`), donc l'expression `sample(mv_gaussian(zeros, i2))` est analysée avec le jugement \vdash^p qui renvoie la distribution `mv_gaussian(zeros, i2)`. Cette distribution est typée avec succès par le jugement \vdash^l dans l'environnement C qui contient les constantes `zeros` et `i2`. Ainsi le corps du nœud `move` a le type $\{ \text{theta} \leftarrow \text{mv_gaussian}(\text{zeros}, \text{i2}) \}$ et le nœud a le type $\{ \text{move} \leftarrow \{ \text{theta} \leftarrow \text{mv_gaussian}(\text{zeros}, \text{i2}) \} \}$.

4 Compilation

Pour exécuter l'algorithme APF présenté en Section 2.3, il faut maintenant compiler les modèles probabilistes pour que les paramètres constants deviennent une entrée supplémentaire. On pourra ainsi ré-exécuter le modèle en explorant différentes valeurs pour mettre à jour la distribution des paramètres constants. La compilation est définie Figure 7 par induction sur la syntaxe du noyau. On commence par typer le programme complet, c'est-à-dire une liste de déclarations $p = d_1 \dots d_n$ dans un environnement vide : $\emptyset, \emptyset \vdash p : H, C$. La fonction de compilation \mathcal{C} est ensuite paramétrée par l'environnement de typage H pour les déclarations, et le type ξ pour les expressions (voir Section 3).

$$\begin{aligned}
\mathcal{C}_\xi(c) &= c \\
\mathcal{C}_\xi(x) &= x \\
\mathcal{C}_\xi((e_1, e_2)) &= (\mathcal{C}_\xi(e_1), \mathcal{C}_\xi(e_2)) \\
\mathcal{C}_\xi(\text{op}(e)) &= \text{op}(\mathcal{C}_\xi(e)) \\
\mathcal{C}_\xi(\text{last } x) &= \text{last } x \\
\mathcal{C}_\xi(\text{present } e_1 \rightarrow e_2 \text{ else } e_3) &= \text{present } \mathcal{C}_\xi(e_1) \rightarrow \mathcal{C}_\xi(e_2) \text{ else } \mathcal{C}_\xi(e_3) \\
\mathcal{C}_\xi(\text{reset } e_1 \text{ every } e_2) &= \text{reset } \mathcal{C}_\xi(e_1) \text{ every } \mathcal{C}_\xi(e_2) \\
\mathcal{C}_\xi(\text{sample}(e)) &= \text{sample}(\mathcal{C}_\xi(e)) \\
\mathcal{C}_\xi(\text{observe}(e_1, e_2)) &= \text{observe}(\mathcal{C}_\xi(e_1), \mathcal{C}_\xi(e_2)) \\
\mathcal{C}_\xi\left(e \text{ where } \begin{array}{l} \text{rec } (\text{init } x_i = e_i)_{1 \leq i \leq k} \\ \text{and } (y_j = e'_j)_{1 \leq j \leq n} \end{array}\right) &= \mathcal{C}_\xi(e) \text{ where } \begin{array}{l} \text{rec } (\text{init } x_i = \mathcal{C}_\xi(e_i))_{1 \leq i \leq k, x_i \notin \text{dom}(\xi)} \\ \text{and } (y_j = \mathcal{C}_\xi(e'_j))_{1 \leq j \leq n, y_j \notin \text{dom}(\xi)} \end{array} \\
\mathcal{C}_\xi(f^\alpha(e)) &= \begin{cases} f(\mathcal{C}_\xi(e)) & \text{si } f \text{ est déterministe} \\ f_model(\alpha, \mathcal{C}_\xi(e)) & \text{si } \alpha \in \text{dom}(\xi) \\ f_model(\alpha, \mathcal{C}_\xi(e)) \text{ where } & \text{sinon} \\ \quad \text{rec init } \alpha = \text{sample}(f_prior) \\ \quad \text{and } \alpha = \text{last } \alpha \end{cases} \\
\mathcal{C}_\xi(\text{infer}(f(e))) &= \text{APF.infer}(f_prior, f_model, e) \\
\mathcal{C}_H(\text{let } x = e) &= \text{let } x = e \\
\mathcal{C}_H(\text{node } f \ x = e) &= \text{node } f \ x = \mathcal{C}_\emptyset(e) \\
\mathcal{C}_H(\text{proba } f \ x = e) &= \text{let } f_prior = \text{img}(\xi) \quad \text{avec } H(f) = \xi \\ &\quad \text{proba } f_model \ (\text{dom}(\xi), x) = \mathcal{C}_\xi(e)
\end{aligned}$$

FIGURE 7 – Fonction de compilation d'un programme ProbZelus pour APF. La fonction de compilation est paramétrée par le type ξ pour les expressions et l'environnement de typage H pour les déclarations (voir Section 3).

Modèles. Pour un modèle `proba` $f \ x = e$, si après l'analyse statique $H(f) = \xi$ (i.e., $C \vdash e : \xi$), alors f est compilé en deux déclarations :

- `let` $f_prior = \text{img}(\xi)$: la distribution *a priori* des paramètres constants dans e , et
- `proba` $f_model \ (\text{dom}(\xi), x) = \mathcal{C}_\xi(e)$: un nouveau modèle qui prend en argument supplémentaire $\text{dom}(\xi)$ pour les paramètres constants.

Expressions. La fonction de compilation d'une expression est paramétrée par le type ξ du nœud qui la contient. Cette fonction supprime les définitions de paramètres constants, $x \in \text{dom}(\xi)$, qui sont passés en entrée du nouveau modèle. Les premiers cas dans la Figure 7, de c à `observe`, implémentent une simple descente récursive sur les sous-expressions. Le cas `where/rec` supprime effectivement les paramètres constants en ne gardant que les équations définissant les variables x telles que $x \notin \text{dom}(\xi)$.

Exemple. Après compilation, le nœud `move` de l'exemple de la Section 2 devient :

```

let move_prior = mv_gaussian(zeros, i2)
proba move_model (theta, x0) = theta, x where
  rec x = x0 → sample(mv_gaussian(pre x +@ theta, 0.5 *@ i2))

```

Appel de nœud. La difficulté principale est de faire remonter les paramètres constants introduits par les appels de nœud. On distingue trois cas.

- Si le nœud est déterministe, il n'y a pas de paramètre constant possible.
- Si les paramètres constants de l'appel de nœud sont également des paramètres constants dans le contexte d'appel, on a $\alpha \in \text{dom}(\xi)$ et on se contente de remplacer l'appel à f^α par un appel à `f_model` en utilisant le nom frais d'instance pour les paramètres constants.
- Sinon, il existe des paramètres constants dans f qui ne sont pas constants dans le contexte d'appel car l'instance f^α est utilisée à l'intérieur d'un `reset/every` ou d'un `present/else`. Dans ce cas, on redéfinit localement ces paramètres en échantillonnant leur distribution *a priori* `f_prior`.

Par ailleurs, dans les nœuds déterministes, on substitue l'appel à `infer(f(e))` par un appel à `APF.infer(f_prior, f_model, e)`.

Exemple. Le nœud radar de la Section 2 devient :

```
let radar_prior = move_prior
proba radar_model (i_move_0, (delta, alpha)) = theta, x where
  rec theta, x = move_model (i_move_0, x0)
  and d = sqrt((vec_get x 0) ** 2. +. (vec_get x 1) ** 2.)
  and a = atan(vec_get x 1 /. vec_get x 0)
  and () = observe(gaussian(2. *. d /. c, delta_noise), delta)
  and () = observe(gaussian(a, alpha_noise), alpha)
```

La variable `i_move_0` est le nom frais associé à l'instance de `move` appelée dans `radar`. Il faut enfin substituer l'appel à `infer` par un appel à `APF.infer` dans le nœud `main`.

```
node main (delta, alpha) = d_theta, d_x where
  rec d = APF.infer(radar_prior, radar_model, (delta, alpha))
  and d_theta, d_x = Dist.split d
```

Correction. La propriété de correction établit que l'inférence appliquée sur un modèle et son compilé définissent les mêmes flots de distributions pour la sémantique idéale de ProbZelus.

Pour tout programme `prog` tel que $\emptyset, \emptyset \vdash \text{prog} : H, C$, pour tout modèle probabiliste m de `prog`,

$$\llbracket \text{infer}(m(\text{obs})) \rrbracket \equiv \llbracket \text{APF.infer}(m_{\text{prior}}, m_{\text{model}}, \text{obs}) \rrbracket$$

En pratique, le calcul de la distribution *a posteriori* est un problème intractable, on utilise donc des méthodes approchées. Nous montrons ici que la compilation préserve la sémantique idéale du modèle, mais permet d'utiliser APF pour calculer une meilleure approximation de la distribution définie par le modèle.

Notations. Pour une mesure μ , on note $\bar{\mu}$ la distribution normalisée $\mu/\mu(\top)$ où \top représente l'ensemble du domaine de définition de μ . On note $f_*(\mu)$ la mesure image par une fonction mesurable f (par exemple, les mesures images par les fonctions de projection π_1 et π_2 permettent de séparer une distribution sur des paires en une paire de distributions). On note $\lambda U. \delta(v)(U)$ la mesure de Dirac au point v ($\delta(v)(U) = 1$ si $v \in U$ et 0 sinon). Enfin on note $\int \mu(dt) f(t)$ l'intégrale de la fonction f le long de la mesure μ où t est la variable d'intégration.

La sémantique de `APF.infer` est définie en appliquant `infer` après avoir échantillonné le paramètre constant α au début de l'exécution.

$$\llbracket \text{APF.infer}(m_{\text{prior}}, m_{\text{model}}, \text{obs}) \rrbracket = \llbracket \text{infer}(m_{\text{model}}(\alpha, \text{obs}) \text{ where } \text{rec } \text{init } \alpha = \text{sample}(m_{\text{prior}}) \text{ and } \alpha = \text{last } \alpha) \rrbracket$$

Deux expressions probabilistes sont équivalentes si elles définissent les mêmes flots de distributions. La sémantique est exprimée en terme d'état initial et de fonction de transition, l'équivalence s'exprime donc avec une relation de bisimilarité [22] initialisée sur les états initiaux et propagée à travers les fonctions de transition [9]. La bisimilarité sur les environnements γ_1 et γ_2 met en rapport les nœuds probabilistes avec leur version compilée et garantit que toutes les autres valeurs sont identiques.

Définition 1 (Bisimilarité d'environnements). Pour tout programme *prog*, pour tous γ_1, γ_2 , $\gamma_1 \equiv \gamma_2$ si pour toute variable $x \in \text{dom}(\gamma_1) \cap \text{dom}(\gamma_2)$, $\gamma_1(x) = \gamma_2(x)$ et pour tout nœud probabiliste `proba` $m \ x = e$ tel que $C \vdash e : \xi$ dans *prog*.

$$\left\{ \begin{array}{l} \gamma_1(m_init) = \{\{e\}\}_{\gamma_1}^{\text{init}} \\ \gamma_1(m_step) = \lambda v. \{\{e\}\}_{\gamma_1[x \leftarrow v]}^{\text{step}} \end{array} \right. \text{ et } \left\{ \begin{array}{l} \gamma_2(m_model_init) = \{\{C_\xi(e)\}\}_{\gamma_2}^{\text{init}} \\ \gamma_2(m_model_step) = \lambda(\theta, v). \{\{C_\xi(e)\}\}_{\gamma_2[\text{dom}(\xi) \leftarrow \theta][x \leftarrow v]}^{\text{step}} \\ \gamma_2(m_prior) = \text{img}(\xi) \end{array} \right.$$

La bisimilarité sur les expressions probabilistes met en rapport les flots de mesures renvoyées à chaque instant par les fonctions de transition.

Définition 2 (Bisimilarité probabiliste). Soient e_1 et e_2 des expressions probabilistes, et une bisimulation $\mathcal{P} \subseteq S \text{ dist} \times S \text{ dist}$ entre les *distributions* d'états. $\{\{e_1\}\} \equiv_{\mathcal{P}} \{\{e_2\}\}$ lorsque :

- pour tout $\gamma_1 \equiv \gamma_2$, $(\delta(\{\{e_1\}\}_{\gamma_1}^{\text{init}}), \delta(\{\{e_2\}\}_{\gamma_2}^{\text{init}})) \in \mathcal{P}$ et
- pour tout $\gamma_1 \equiv \gamma_2$, si $(\sigma_1, \sigma_2) \in \mathcal{P}$

$$\left\{ \begin{array}{l} \mu_1 = \lambda U. \int \sigma_1(ds_1) \{\{e_1\}\}_{\gamma_1}^{\text{step}}(s_1)(U), \ o_1 = \pi_{1*}(\bar{\mu}_1), \ \sigma'_1 = \pi_{2*}(\bar{\mu}_1) \\ \mu_2 = \lambda U. \int \sigma_2(ds_2) \{\{e_2\}\}_{\gamma_2}^{\text{step}}(s_2)(U), \ o_2 = \pi_{1*}(\bar{\mu}_2), \ \sigma'_2 = \pi_{2*}(\bar{\mu}_2) \end{array} \right. \text{ alors } \left\{ \begin{array}{l} (\sigma'_1, \sigma'_2) \in \mathcal{P} \\ o_1 = o_2 \end{array} \right.$$

Nous prouvons la correction de la compilation dans un cadre simplifié.

Théorème 1. *Pour tout programme prog tel que $\emptyset, \emptyset \vdash \text{prog} : H, C$, pour tout modèle probabiliste m de prog sous la forme `proba` $m \ x = e$ `where` `rec` `init` $\theta = \text{sample}(d)$ `and` $\theta = \text{last}$ θ , avec $H(m) = \{\theta \leftarrow d\}$, où e ne contient pas d'appel de nœud, alors $\llbracket \text{infer}(m(\text{obs})) \rrbracket \equiv \llbracket \text{APF.infer}(m_prior, m_model, \text{obs}) \rrbracket$.*

Les contraintes sur m ne sont pas excessives : e peut contenir des définitions locales, θ peut être un tuple de paramètres constants, et on peut inliner les appels de nœuds. De plus, on peut imaginer une analyse statique qui permet de vérifier que m est sous la bonne forme.

Démonstration. Vue la sémantique de `infer` et `APF.infer`, il suffit de montrer que : $\{\{m(\text{obs})\}\} \equiv_{\mathcal{P}} \{\{m_model(\alpha, \text{obs}) \text{ where } \text{rec } \text{init } \alpha = \text{sample}(m_prior) \text{ and } \alpha = \text{last } \alpha\}\}$ pour une bisimulation \mathcal{P} . Comme $H(m) = \{\theta \leftarrow d\}$, alors $C \vdash e : \emptyset$ car le seul paramètre constant, θ , est défini à l'extérieur de e et donc $C_\emptyset(e) = e$. On pose $(\sigma_1, \sigma_2) \in \mathcal{P}$ ssi il existe σ tel que $\sigma_1 = \lambda U. \int \sigma(dm, ds) \delta(\cdot, (m, (\cdot), s))(U)$ et $\sigma_2 = \lambda U. \int \sigma(dm, ds) \delta(m, (\cdot), ((\cdot), (\cdot)), s))(U)$. On vérifie alors que \mathcal{P} relie les états initiaux et se propage à travers les fonctions de transition ce qui suppose de dérouler et faire commuter les intégrales imbriquées introduites par les environnements locaux. \square

5 Implémentation

Il reste maintenant à implémenter la construction `APF.infer`. Dans cette section, nous décrivons d'abord comment implémenter les méthodes de Monte Carlo séquentielles (SMC) pour les machines à états produites par la compilation d'un programme synchrone. Nous détaillons ensuite l'implémentation de *Assumed Parameter Filter* comme un cas particulier de SMC.

5.1 Méthodes de Monte Carlo séquentielles

Zelus compile les nœuds en machines à états. Une machine est une structure qui définit les méthodes suivantes : `alloc` alloue la mémoire nécessaire pour stocker l'état, `reset` réinitialise l'état en place, et `step` implémente la fonction de transition qui met à jour l'état en place. La mémoire est donc allouée statiquement avant l'exécution. Pour implémenter le filtrage, le compilateur génère également une méthode `copy` qui copie l'état d'une particule source dans celui d'une particule cible. En OCaml, on a ainsi² :

```
type ( $\alpha$ ,  $\beta$ ) cnode = Cnode: { alloc : unit  $\rightarrow$  's;
                             reset : 's  $\rightarrow$  unit;
                             step  : 's  $\rightarrow$   $\alpha \rightarrow \beta$ ;
                             copy  : 's  $\rightarrow$  's  $\rightarrow$  unit; }  $\rightarrow$  ( $\alpha$ ,  $\beta$ ) cnode
```

Les nœuds probabilistes sont également des machines à état mais la fonction `step` prend un argument supplémentaire `prob` qui contient les informations nécessaires à l'inférence :

```
type ( $\alpha$ ,  $\beta$ ) pnode = (prob *  $\alpha$ ,  $\beta$ ) cnode
```

Les méthodes de Monte Carlo séquentielles utilisent un ensemble de simulations indépendantes, les *particules*, associées à un score pour approximer la distribution recherchée. En pratique, au cours d'une simulation les constructions probabilistes `sample` et `observe` utilisent l'argument supplémentaire `prob` pour mettre à jour le score de la particule.

```
type prob = {id : int; scores : float array}

let sample (prob, d) = draw d
let observe (prob, d, x) = prob.scores.(prob.id)  $\leftarrow$  prob.scores.(prob.id) +. logpdf d x
```

Le type `prob` contient un identifiant de particule `id`, et le tableau de scores `scores`. `sample` tire un échantillon aléatoire dans la distribution `d`, et `observe` incrémente le score de la particule avec la vraisemblance de l'observation, *i.e.*, la valeur de la densité au point d'observation $d_{\text{pdf}}(x)$. On calcule les scores en échelle logarithmique pour des raisons de stabilité numérique.

L'opérateur `infer` est un nœud d'ordre supérieur qui transforme une machine probabiliste de type `(α , β) pnode` en une machine déterministe de type `(α , β dist) cnode` qui calcule un flot de distributions. L'implémentation de `infer` suit de près l'Algorithme 1 présenté dans la Section 2.

```
type  $\alpha$  infer_state = {mutable sigma :  $\alpha$  dist}

let infer n particle =
  let Cnode {alloc; reset; step; copy} = particle in
  let i_alloc () = {sigma = dirac (alloc ())} in
  let i_reset state =
    let s = Dist.draw state.sigma in
    reset s; state.sigma  $\leftarrow$  dirac (s)
  in
  let i_step state obs =
    let particles = Array.init n (fun _  $\rightarrow$  let p = alloc () in
                                             let s = Dist.draw state.sigma in
                                             copy s p; p) in
    let scores = Array.make n 0. in
    let values = Array.mapi (fun i s  $\rightarrow$  step s ({id = i; scores}, obs)) particles in
```

2. Voir le fichier `lib/std/ztype.ml` du compilateur Zelus <https://github.com/inria/zelus>.

```

state.sigma ← multinomial particles scores;
multinomial values scores
in
let i_copy src dst = dst.sigma ← src.sigma in
Cnode {alloc = i_alloc; reset = i_reset; step = i_step; copy = i_copy}

```

L'opérateur `infer` prend en arguments un nombre de particules `n` et une machine probabiliste `particle`. L'état mémoire de `infer` est une distribution d'états possibles initialisé par la fonction `i_alloc` avec une distribution de Dirac sur l'état initial des particules. La fonction de transition `i_step` commence par échantillonner un tableau de particules dans la distribution d'états : c'est le filtrage. On itère ensuite la fonction de transition des particules `step` sur le tableau `particles` pour mettre à jour l'état de chaque particule et calculer les valeurs de sortie. On utilise un tableau `scores` pour stocker les scores obtenus. Pour chaque particule l'argument `prob` prend la valeur `{id = i; scores}` qui permet de lier la case `scores.(i)` à la particule `i`. On peut enfin mettre à jour l'état de `infer` avec une distribution multinomiale sur les particules, et renvoyer la distribution de sortie : une distribution multinomiale sur les valeurs.

Un filtre particulaire est un cas particulier simple de SMC où chaque particule se contente d'exécuter le modèle pour obtenir une valeur de sortie et un score.

```
let infer_pf n model = infer n model
```

Remarque. Cette implémentation suit de près la sémantique idéale de ProbZelus [2] mais la fonction de transition alloue de la mémoire à chaque itération. En pratique, le nombre de particules est fixé. On peut donc allouer statiquement toute la mémoire nécessaire dans l'état et utiliser uniquement des modifications en place dans la fonction de transition.

5.2 Assumed Parameter Filter

Assumed Parameter Filter est également un cas particulier de SMC qui utilise des particules plus avancées. La fonction de transition des particules suit l'Algorithme 2 décrit dans la Section 2.3. En plus de l'état et du score, chaque particule maintient une distribution de paramètres constants Θ^i qui est mise à jour à chaque instant.

Pour mettre à jour Θ^i , il faut être capable de mesurer la vraisemblance d'une exécution où seule la valeur des paramètres constants θ a changé : $\Theta_t^i = \text{Udpate}(\Theta_{t-1}^i, \lambda\theta, \text{model}(y_t \mid \theta, x_{t-1}^i, x_t^i))$ dans l'Algorithme 2. Il faut donc trouver un moyen de rejouer un pas d'exécution en fixant la valeur de tous les paramètres d'état.

Mémoriser les échantillons. On peut implémenter l'opérateur `sample` comme une machine à état et utiliser l'état mémoire pour mémoriser les valeurs échantillonnées lors de la première exécution du modèle : $x_t^i, w_t^i = \text{model}(y_t \mid \theta^i, x_{t-1}^i)$ dans l'Algorithme 2. Malheureusement, l'exécution du modèle modifie l'état du programme en place. Si on ne fait pas attention, la fonction `Udpate` utilise donc $\lambda\theta, \text{model}(y_t \mid \theta, x_{t-1}^i, x_t^i)$ au lieu de $\lambda\theta, \text{model}(y_t \mid \theta, x_{t-1}^i, x_t^i)$.

Pour résoudre ce problème, nous proposons de stratifier la mémoire. Avant l'exécution du modèle, on copie superficiellement l'état qui contient des pointeurs vers l'état *profond* de chaque `sample`. Après l'exécution, on peut ainsi réutiliser cette copie de l'état où seul l'état des `sample` a été modifié. L'état de `sample` ne stocke donc pas directement la valeur échantillonnée, mais un pointeur vers une mémoire partagée. On distingue alors deux modes d'exécution gardés par une variable booléenne `replay`. Si `replay` est faux, l'appel à `sample(d)` tire un nouvel échantillon dans `d` et met à jour la mémoire partagée. Si `replay` est vrai, la valeur est déjà connue et l'appel à `sample(d)` se comporte comme `observe(d, m)` où `m` est le contenu de la mémoire partagée.

```

type prob = {id : int; scores : float array; replay : bool}

let sample =
  let alloc () = ref (ref None) in
  let reset state = !state := None in
  let step state (prob, dist) =
    match prob.replay with
    | false → let v = Dist.draw dist in !state := Some (v); v
    | true → let v = Option.get (! !state) in observe (prob, (dist, v)); v in
  let copy src dst = dst := !src in
  Cnode { alloc; reset; copy; step; }

```

La méthode `copy` implémente la copie superficielle qui ne duplique que les pointeurs en préservant l'état profond.

On peut maintenant définir une fonction `apf_particle` qui prend en argument le résultat de la compilation de la Section 4 et qui construit une particule qui implémente l'algorithme APF.

```

type ( $\alpha$ ,  $\beta$ ) apf_state = {p_state:  $\alpha$  ; mutable d_theta :  $\beta$  dist}

let apf_particle m_model m_prior =
  let Cnode {alloc; reset; step; copy} = m_model in
  let p_alloc () = {p_state = alloc (); d_theta = m_prior} in
  let p_reset state = reset state.p_state; state.d_theta ← m_prior in
  let p_step ({p_state; d_theta} as state) (prob, obs) =
    let start_state = alloc () in copy p_state start_state;
    let theta = Dist.draw d_theta in
    let o = step p_state ({ prob with replay = false }, (theta, obs)) in
    let tmp_state = alloc () in
    state.d_theta ← update
      (fun theta →
        copy start_state tmp_state;
        let prob = {id = 0; replay = true; scores = [|0.|]} in
        let _ = step tmp_state (prob, (theta, obs)) in
        prob.scores.(prob.id))
    d_theta;
  o
in
let p_copy src dst = copy src.p_state dst.p_state; dst.d_theta ← src.d_theta in
Cnode {alloc = p_alloc; reset = p_reset; step = p_step; copy = p_copy}

```

En plus de l'état du modèle, l'état de chaque particule contient une distribution de paramètres constants `d_theta` initialisée par la fonction `p_alloc` avec `m_prior`. La fonction de transition `p_step` d'une particule se décompose ensuite de la manière suivante :

1. Copier superficiellement l'état de départ `p_state` dans une variable `start_state`.
2. Échantillonner une valeur `theta` dans `d_theta`.
3. Exécuter la fonction de transition de `m_model`, `step` avec `replay = false` pour mettre à jour l'état et mémoriser les valeurs de tous les paramètres d'état.
4. Mettre à jour la distribution `d_theta` en utilisant une fonction qui prend en argument `theta`, copie superficiellement l'état `start_state` dans une variable `tmp_state` et renvoie le score obtenu en ré-exécutant `step` à partir de `tmp_state` avec `replay = true` et un score réinitialisé pour rejouer l'instant courant en réutilisant les valeurs des paramètres d'état.
5. Renvoyer la sortie `o` calculée à l'étape 3.

Finalement, `APF.infer` est un cas particulier de SMC qui utilise `apf_particle`.

```
let apf_infer n m_prior m_model = infer n (apf_particle m_model m_prior)
```

Modularité. Notre implémentation ne fait pas d'hypothèse sur la fonction *Update* de l'Algorithme 2 qui peut donc implémenter diverses heuristiques. L'article original [16] se concentre sur la méthode de correspondance des moments. À chaque instant, on tire un ensemble d'échantillons aléatoires dans `d_theta` et on mesure les scores obtenus. `d_theta` devient alors la distribution gaussienne qui approxime le mieux ces échantillons pondérés.

Nous avons également implémenté l'échantillonnage préférentiel, une méthode d'inférence sans filtrage. Au premier instant, on tire un ensemble de valeurs aléatoires pour les paramètres constants. Puis, à chaque instant, on met à jour le score associé à chacune de ces valeurs.

6 Évaluation

Dans cette section, nous validons expérimentalement notre implémentation d'APF sur un ensemble de modèles ProbZelus. Notre évaluation répond aux questions suivantes : **(QR1)** Quel est l'impact d'APF sur la précision des estimations à ressources fixées ? **(QR2)** APF résout-il le problème d'appauvrissement sur les estimations de paramètres constants ?

Méthode expérimentale. Pour chacun des exemples, on mesure l'évolution sur 500 pas d'exécution pas de l'erreur quadratique moyenne (*Mean Square Error*, MSE) pour les paramètres d'état et les paramètres constants. Pour mesurer l'appauvrissement, on regarde également l'évolution de la taille effective d'échantillonnage (*Effective Sample Size*, ESS), une métrique qui indique le nombre d'échantillons utiles. Les expériences ont été réalisées avec un processeur Intel Core i5-6200U (2.30GHz) avec 8Go de mémoire vive.

On compare les performances d'APF avec deux autres méthodes d'inférence : l'échantillonnage préférentiel (*Importance Sampling*, IS), une méthode de Monte Carlo élémentaire sans filtrage, et le filtre particulaire (*Particle Filter*, PF), une méthode de Monte Carlo séquentielle avec filtrage à chaque instant. Pour chacune des méthodes d'inférence, on fixe les ressources à 10^4 particules. Sauf mention contraire, APF répartit les ressources disponibles avec 100 particules de filtrage et 100 particules d'échantillonnage pour le calcul de la correspondance des moments.

Pour comparer l'ESS d'APF avec celui des autres méthodes, on génère ensuite le même nombre d'échantillons en tirant une valeur aléatoire dans la distribution des paramètres constants pour chaque particule d'échantillonnage. Si n_e est le nombre de particules d'échantillonnage et n_f le nombre de particules de filtrage utilisées par APF, chaque particule génère donc n_e échantillons qui sont tous associés au score de la particule. On obtient donc à chaque instant $n_e * n_f$ échantillons, soit le nombre d'échantillons produit par les autres méthodes.

Nous avons sélectionné des modèles qui illustrent plusieurs types de dépendances entre paramètres d'état et paramètres constants. **Coin** infère le biais d'une pièce à partir d'une série de jets [2]. Ce modèle n'a pas de paramètre d'état. APF utilise 10^4 particules d'échantillonnage. **Kalman** estime la position d'un agent à partir d'observations bruitées [2]. Ce modèle n'a pas de paramètre constant. APF utilise 10^4 particules de filtrage. **Sin** est similaire à Kalman, mais le modèle de mouvement fait intervenir une relation non-linéaire entre un paramètre constant et la position. C'est le modèle utilisé pour motiver APF dans l'article original [16]. **Split** fusionne deux problèmes indépendants : l'inférence d'un paramètre constant à partir d'observations bruitées et un filtre de Kalman. **Radar** est le modèle de radar de la Section 2.

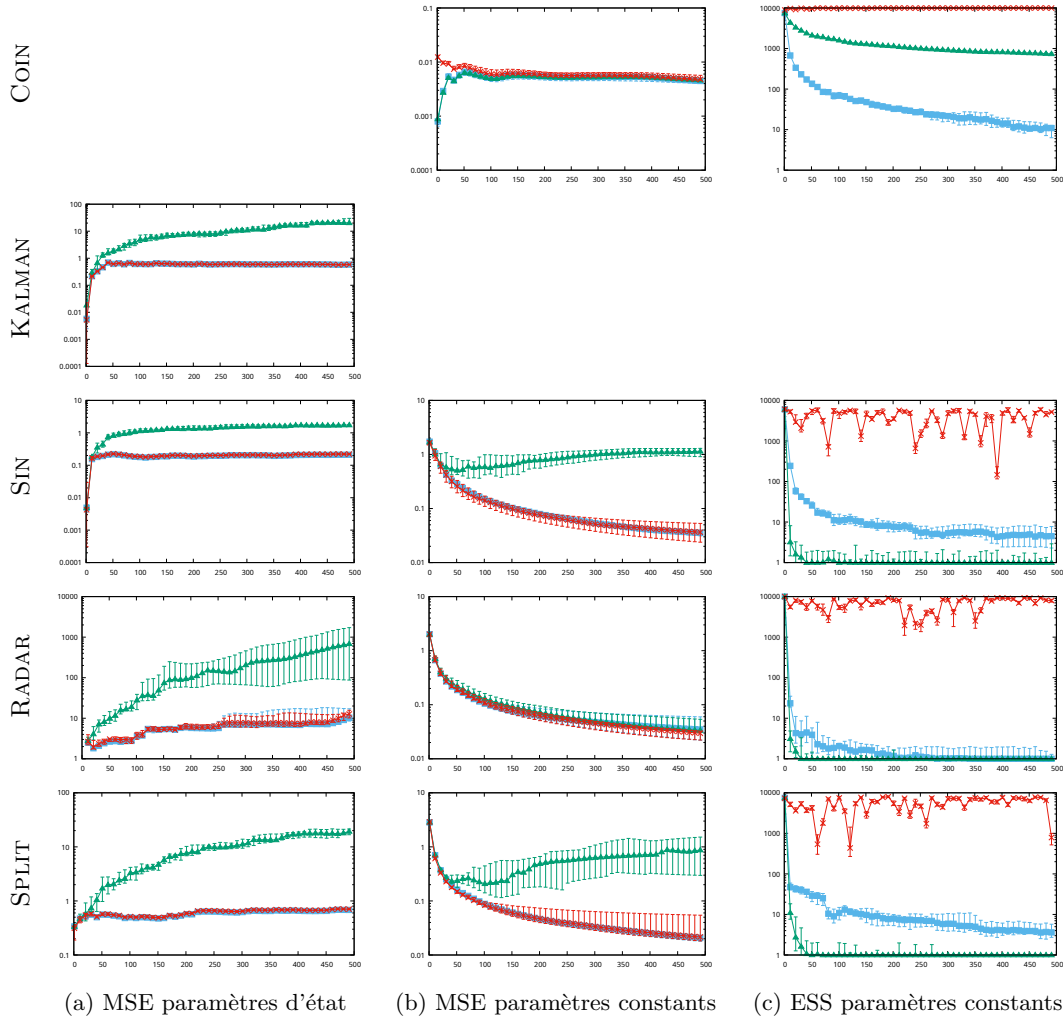


FIGURE 8 – ■ PF, ▲ IS, × APF. Pour chacun des 5 modèles on mesure l'évolution sur 500 pas d'exécution du *mean square error* (MSE) pour les paramètres d'état et les paramètres constants, et le *effective sample size* (ESS) des paramètres constants. Les marqueurs indiquent les valeurs médianes et les barres d'erreur correspondent aux quantiles 90% et 10% sur 10 exécutions.

QR1 : Précision. Les résultats de précision sont présentés Figures 8a et 8b. On remarque que, sur tous les exemples, la précision d'APF est équivalente à celle de PF pour l'estimation des paramètres d'état et des paramètres constants. Les courbes bleue et rouge sont quasiment identiques. On note cependant une plus grande variabilité dans les résultats d'APF sur les paramètres constants, notamment sur les derniers pas d'exécution. Ce comportement s'explique par le phénomène d'appauvrissement : après autant de filtrages successifs, l'estimation de PF pour les paramètres constants ne repose plus que sur une unique valeur et on a perdu toute mesure sur l'incertitude de l'estimation.

QR2 : Appauvrissement. Les résultats d'appauvrissement sont présentés Figure 8c. L'ESS pour APF reste quasiment constant autour du nombre d'échantillons total. Tous les échantillons générés ajoutent de l'information sur la distribution recherchée.

Sans surprise, on observe que l'ESS pour PF décroît strictement à chaque nouveau filtrage. L'ESS pour IS s'effondre lui aussi mais pour une raison différente. Aucune particule n'est éliminée, mais la plupart des valeurs tirées initialement deviennent tellement improbables que leur score tombe à 0. Ces particules n'ajoutent plus d'information pour le calcul de la distribution et ne sont donc plus comptées dans l'ESS.

Coût. Sur les trois exemples précédents où APF répartit les ressources entre filtrage et échantillonnage APF est en moyenne 1.43x plus lent que PF (Sin : 1.32x, Radar : 1.65x, Split : 1.33x). Pour les modèles avec un seul type de paramètre où APF utilise toutes les ressources soit en filtrage, soit en échantillonnage, APF est en moyenne 2.63x plus lent que PF. Ce ralentissement s'explique par le fait que APF ré-exécute plusieurs fois une même itération du modèle pour mettre à jour la distribution des paramètres constants puis générer les échantillons, là où le filtre particulière n'exécute le modèle qu'une fois par instant. Ces performances sont correctes si l'appauvrissement n'est pas acceptable et que l'on cherche à estimer précisément l'incertitude sur la valeur des paramètres constants.

7 Conclusion

Nous avons montré comment estimer les paramètres constants des modèles ProbZelus en évitant le problème d'appauvrissement grâce à l'algorithme d'inférence *Assumed Parameter Filter*. Notre implémentation repose sur une nouvelle analyse statique, une nouvelle passe de compilation, et un nouveau moteur d'inférence pour ProbZelus. Nos résultats expérimentaux montrent que cette méthode d'inférence permet d'éviter le problème d'appauvrissement pour les paramètres constants sans pénaliser la précision des estimations pour les paramètres d'état.

Travaux connexes. L'analyse statique de la Section 3 est inspirée d'analyses spécifiques aux langages synchrones flot de données, en particulier l'analyse d'initialisation [11], qui vérifie que tous les flots sont bien définis au premier instant, et le typage des arguments statiques de Zelus, qui vérifie que certaines valeurs peuvent être calculées dès la compilation [7].

La compilation présentée en Section 4 se concentre uniquement sur les paramètres constants. C'est le moteur d'exécution qui se charge de fixer la valeur des paramètres d'état à l'aide d'une mémoire partagée pour rejouer un instant. Une alternative possible est de se rapprocher de la compilation des modèles hybrides de Zelus [4] et ajouter des entrées/sorties pour toutes les variables aléatoires introduites dans le modèle. On pourrait ainsi simplifier le moteur d'exécution au prix d'une compilation plus avancée.

La méthode d'inférence privilégiée de ProbZelus est *Delayed Sampling* [2], une méthode de Monte Carlo séquentielle qui calcule autant que possible des solutions analytiques exactes et n'échantillonne une variable aléatoire que lorsque le calcul symbolique échoue [21]. *Delayed Sampling* peut ainsi éviter l'appauvrissement si les paramètres constants n'ont jamais besoin d'être échantillonnés. Malheureusement, le moteur symbolique ne sait exploiter que des relations de conjugaison entre variables aléatoires, ou des relations affines [1]. On pourrait cependant combiner APF et *Delayed Sampling* pour améliorer la précision de ces deux méthodes.

Remerciements. Nous remercions les relecteurs et le suiveur pour leurs conseils. Ce travail a été en partie financé par le projet ReaLiSe, un projet Émergence(s) de la Ville de Paris.

Références

- [1] Eric Atkinson, Charles Yuan, Guillaume Baudart, Louis Mandel, and M. Carbin. Semi-symbolic inference for efficient streaming probabilistic programming. In *OOPSLA*, 2022.
- [2] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive probabilistic programming. In *PLDI*. ACM, 2020.
- [3] Guillaume Baudart, Louis Mandel, and Reyhan Tekin. Jax based parallel inference for reactive probabilistic programming. In *LCTES*, San Diego, USA, June 2022.
- [4] Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Divide and recycle : types and compilation for a hybrid synchronous language. In *LCTES*, April 2011.
- [5] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. The synchronous languages 12 years later. *Proc. IEEE*, 91(1) :64–83, 2003.
- [6] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro : Deep universal probabilistic programming. *J. Mach. Learn. Res.*, 20 :28 :1–28 :6, 2019.
- [7] Timothy Bourke, Francois Carcenac, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. A synchronous look at the simulink standard library. *ACM Trans. Embed. Comput. Syst.*, 16(5s) :176 :1–176 :24, 2017.
- [8] Timothy Bourke and Marc Pouzet. Zélus : a synchronous language with ODEs. In *HSCC*, pages 113–118. ACM, 2013.
- [9] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. In *CMCS*, volume 11 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.
- [10] Nicolas Chopin and Omiros Papaspiliopoulos. *An introduction to sequential Monte Carlo*. Springer, 2020.
- [11] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer*, 6(3) :245–255, 2004.
- [12] Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Mixing signals and modes in synchronous data-flow systems. In *EMSOFT*, 2006.
- [13] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. SCADE 6 : A formal language for embedded critical software development. In *TASE*, pages 1–11. IEEE Computer Society, 2017.
- [14] Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen : a general-purpose probabilistic programming system with programmable inference. In *PLDI*, pages 221–236. ACM, 2019.
- [15] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo samplers. *J. Royal Statistical Society : Series B (Statistical Methodology)*, 68(3) :411–436, 2006.
- [16] Yusuf Bugra Erol, Yi Wu, Lei Li, and Stuart Russell. A nearly-black-box online algorithm for joint parameter and state estimation in temporal models. In *AAAI*, 2017.
- [17] Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing : A language for flexible probabilistic inference. In *Proceedings of Machine Learning Research*, 2018.
- [18] Noah D. Goodman and Andreas Stuhlmüller. The design and implementation of probabilistic programming languages, 2014. Accessed April 2020.
- [19] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.
- [20] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. Fastslam : A factored solution to the simultaneous localization and mapping problem. In *AAAI*, 2002.
- [21] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. In *AISTATS*, 2018.
- [22] David Michael Ritchie Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, 1981.

An AST for Representing Programs with Invariants and Proofs

Guillaume Bertholon^{1,2} and Arthur Charguéraud^{2,1}

¹ Université de Strasbourg, CNRS, ICube, France

² Inria, Strasbourg, France

Abstract

Deductive verification enables one to check that a program satisfies its specification. There are mainly two approaches: either the user provides invariants in the form of annotations and use a tool to extract proof obligations, like in, e.g., Why3; or the user verifies the program through interactive proofs, like in, e.g., CFML, by providing invariants during the proof steps.

We are interested in expressing in Coq the representation of a program, accompanied with not only its invariants but also its proof terms. Concretely, we present an AST for representing source code and specification in a deep embedding style, and embedded lemmas in shallow embedding style. Such lemmas can be established using the full capabilities of the prover.

We develop a way to build these ASTs from source code using CFML-style interactive tactics. We also develop a way to build these ASTs by extracting proof obligations from source code already annotated with its invariants. Besides, we provide a way to validate our ASTs by reifying them as Coq proof terms.

This work is a first step towards a long term project to devise a trustworthy, user-guided, source-to-source optimization framework. On the one hand, we may need to exploit invariants to justify the correctness of code transformations. On the other hand, to be able to chain transformations, we also need every transformation to update the program annotations.

1 Introduction

Producing programs without bugs is known to be a hard task. In order to guarantee the absence of bugs, programmers can use deductive verification. Deductive verification is a technique that allows programmers to state the formal specification, and statically check that they hold on the program. The link between the code and the specification must be guaranteed by a proof. In practice however, verifying a program requires a lot of effort. Indeed, the amount of effort grows even larger when targeted program grow in size or in complexity. The formal verification of realistic high performance code, appears prohibitively costly.

In our long term project, we aim to develop a methodology for deriving programs that are both highly optimized and formally verified. Our plan is to start from unoptimized code that is easier to verify, and progressively refine this code into a more performant version. Such source-to-source code refinement process has already been shown in the OptiTrust framework to be expressive enough to match the performance of a numerical simulation optimized by hand [3]. However, OptiTrust does not yet provide formal guarantees about the correctness of the output code. To ensure that a source-to-source transformation preserves correctness, it may be needed to exploit the existing invariants of the input code. Therefore to support chain of transformation, each transformation will need to maintain program invariants.

In this paper, we present an abstract syntax tree (AST) data structure for representing programs, together with their invariants and proofs. We also give practical methods to build these

ASTs, and to validate the proof they carry. However, we leave to future work the implementation of practical transformations, e.g., function inlining, arithmetic rewriting (such as replacing a division by a shift), loop reordering, or even replacing a part of the program by a semantically equivalent but arbitrary code. Nonetheless, our AST provides a way for transformations to access those invariants and proofs and maintain both while changing the code.

Before explaining in more details how our AST is represented, let us recall the ways in which invariants are provided and proofs are constructed in state-of-the-art deductive verification tools. Typically, the specification of a program is expressed with a Hoare triple stating what pre-condition must be true before the execution, and what post-condition is true after the execution. In addition, one needs to describe the mutable state, and express the fact that pointers may or may not overlap, i.e., do not alias. These memory properties can be stated concisely using Separation Logic [11]. Given a specification, the verification of a program includes three kind of steps: (1) semantic steps, which process one construct of the source code; (2) structural steps, which refine invariants at a given program point, possibly exploiting the rules of separation logic; and (3) pure steps, which do not involve the program or the data it manipulates directly but are mathematical arguments needed to conclude. Throughout the paper, we call such pure steps *proof leaves*, since they can only appear at the end of proof branches.

These proof steps can be organized in different ways depending on the approach followed to construct a proof that a program satisfies its specification. There are mainly two kind of approaches.

- One can use an annotation strategy. In this case, the user starts with adding annotations to the program. These annotations must contain the specification but also useful invariants that are hard to deduce (e.g. loop invariants). Then, a tool extracts proof leaf goals to ensure that assertions are satisfied using the semantics of the programming language. Doing so, it automatically applies semantics and structural proof steps. These generated proof leaves can then be discharged using automated provers such as SMT solvers. If the solvers are unable to conclude, the user can add more annotations, as intermediate assertions. This is the approach used by, for instance, Why3 [5].
- Or, the user can verify the program through an interactive proof. There, the tool let the user interactively choose a semantic or structural step and update the remaining program code and pre-condition accordingly. In particular, at any point the user can either weaken the pre-condition or strengthen the post-condition. Some of the steps generate proof leaves to justify their correctness. Such proof leaves can also be proven interactively; they correspond to standard Coq lemmas. This second approach is used, for instance, by CFML [4].

Both approaches have their strengths for representing program with invariants and proofs. The annotation approach anchors the invariants directly at the relevant place in the code, in a very natural way. The interactive proof approach gives a fully interactive construction method. Moreover it produces statements that can be easily related to the formal semantics of the programming language, in a *foundational* way.

In Section 2, we explain how programs with invariants can be described in both approaches. In Section 3, we present our AST data structure as a Coq inductive type. As we will argue, our AST can be either viewed as a *deep embedding* of a proof derivation; or as an annotated source code, furthermore decorated with proof environments and proof terms. All proof leaves remain represented by means of a *shallow embedding*, that is, as Coq proof terms. In Section 4, we explain how instances of our AST can be built using either of the two aforementioned

approaches: either via annotation and extraction of proof obligations, or via fully interactive processing. In section §5, we present a *reification* technique for validating that an instance of our AST indeed satisfies the reasoning rules of the program logic. Concretely, the AST is translated into a Coq proof term using lemmas and semantic definitions from CFML. This reification procedure will be helpful in future work to validate the correctness of any source-to-source transformation step.

2 Representing Proofs

As said in the introduction, our AST for representing program with invariants and proofs will be inspired by the two classical approaches for doing program verification, namely program annotation and interactive proofs.

Before getting into the details of our AST, let's review how proofs that an imperative program satisfies a specification are usually made in both styles.

2.1 Program Annotations

The program annotation approach gives a direct link between code and attached invariants. This is an important property for transformations because we need to find which invariants need a modification when updating the code.

Let's consider as example program, the function `ref_move`.

```
let ref_move r n =
  let m = get r in
  let s = ref m in
  set r n;
  s
```

This function can be specified by the following lemma:

$$\forall r, n, m. \{r \hookrightarrow m\} (\text{ref_move } r \ n) \{\lambda s. r \hookrightarrow n \star s \hookrightarrow m\}$$

This lemma indicates that under the pre-condition that the cell at address `r` contains the value `m`, the function will return a new pointer `s` that points to `m` and that `r` will now point to `n`. Here, the symbol \star denotes the separating conjunction of the logic. It implies in particular that $r \neq s$. Following the annotation approach, we then decorate the source code with separation logic assertions at all key program points, yielding the program below.

```
let ref_move r n =
  r \hookrightarrow a
  let m = get r in
  [m = a] \star r \hookrightarrow a
  let s = ref m in
  [m = a] \star r \hookrightarrow a \star s \hookrightarrow a
  set r n;
  r \hookrightarrow n \star s \hookrightarrow m
  s
  \lambda s. r \hookrightarrow n \star s \hookrightarrow m
```

With such an annotated program, one can run a tool to extract, for each block of instruction between annotation, a set of proof leaf obligations that is sufficient for guaranteeing the specification. Typically, such tools are implemented using weakest precondition (WP) computation. With separation logic style specifications, this technique is used by tools such as VeriFast [12], and Viper [10].

In more complicated examples, the memory state contains representation predicates to express recursive data-structures. For instance, a mutable linked list with head at address p , storing the values described by the list L , could be described by a predicate $p \rightsquigarrow \text{Mlist } L$ recursively defined as:

$$p \rightsquigarrow \text{Mlist } L \equiv \text{match } L \text{ with} \\
\begin{array}{l}
| \text{nil} \Rightarrow [p = \text{null}] \\
| x :: L' \Rightarrow \exists q. (p.\text{head} \hookrightarrow x) \star (p.\text{tail} \hookrightarrow q) \star q \rightsquigarrow \text{Mlist } L'.
\end{array}$$

To manipulate those representation predicates, one may need to add some ghost instructions in the annotated program specifically to change the vision of the memory state, by unfolding or folding back a `Mlist` predicate.

The advantage of the annotated program representation for program transformations is that intermediate assertions are attached to program points, so it is easy to tell which assertions need to be updated when local changes are applied to the code. For example, for a transformation that inlines a function, we can specialize the invariants of the body of the function and use these specialized invariants for the inlined code. All the rest of the code can keep its invariants unchanged. As another example, consider the reordering of independent let-bindings. We need to generate new intermediate annotations, but all the annotations before the first let-binding and in the continuation of the second let-binding can be directly reused. In that example, checking that we are able to generate the intermediate annotations effectively proves the absence of dependence between the two bindings. In most cases, such a check is purely syntactic and only use the commutativity of the separating conjunction.

2.2 Interactive Construction of Derivations

The interactive proof strategy provides a way to build a *derivation tree*. These trees perfectly capture the proof steps applied, their required hypotheses, and the links with the program semantics. This is an important property for proof validation, and gives a practical method for the construction of the AST. In this approach, proof are carried out inside an interactive proof assistant, by exploiting reasoning rules of the program logic expressed as lemmas.

Foundational systems such as CFML [4], VST [6] or Iris [7] are built using rules stated as lemmas. These lemmas are proved with respect to the programming language semantics and the logical framework. This foundational approach reduces the trusted code base.

In the derivation trees produced by such frameworks, every node corresponds to a logic rule. Reasoning rules with hypotheses correspond to nodes with sub-trees in the derivation. Interactive verification frameworks typically provide tactics for applying the reasoning rules. They thereby give the illusion to the user of following the code line by line, while building the proof. When invoking these tactics, the user provides the program invariants that cannot be automatically inferred.

Our AST features constructors to match both *structural reasoning rules* and *semantic reasoning rules*. In these reasoning rules, we choose to make the context, written Γ , explicit. This context consists of a list of bindings, for variables and pure facts (i.e., hypotheses). In many presentation, the context, which corresponds to the Coq context, is left implicit, but we find it useful to make context explicit in order to better explain the transitions that affect the context.

Below, the symbol \vdash denotes the heap entailment: $H \vdash H'$ holds iff all heaps satisfying H also satisfy H' . Post-conditions are represented by functions from program return values to heap predicates and we define their point-wise entailment $Q \vdash Q'$ as $\forall v : \text{Val}, Q v \vdash Q' v$. The notation $[P]$ corresponds to the pure heap predicate P . The judgement $\Gamma \vdash \{H\} t \{Q\}$ asserts that the term t admits H as a pre-condition and Q as a post-condition, in the environment Γ . For details, we refer to Charguéraud's survey [2].

We consider the 4 following structural rules of Separation Logic.¹

$$\begin{array}{c}
\text{CONSEQUENCE} \\
\frac{\Gamma \models H \vdash H' \quad \Gamma \models \{H'\} t \{Q'\} \quad \Gamma \models Q' \vdash Q}{\Gamma \models \{H\} t \{Q\}}
\end{array}
\qquad
\begin{array}{c}
\text{FRAME} \\
\frac{\Gamma \models \{H\} t \{Q\}}{\Gamma \models \{H \star H'\} t \{Q \star H'\}}
\end{array}$$

$$\begin{array}{c}
\text{PROP} \\
\frac{\Gamma, - : P \models \{H\} t \{Q\}}{\Gamma \models \{[P] \star H\} t \{Q\}}
\end{array}
\qquad
\begin{array}{c}
\text{EXISTS} \\
\frac{\Gamma, x : T \models \{H\} t \{Q\}}{\Gamma \models \{\exists x : T. H\} t \{Q\}}
\end{array}$$

There is one semantic rule per term construct. We show below a few of them.

$$\begin{array}{c}
\text{VAL} \\
\frac{}{\Gamma \models \{[]\} v \{\lambda r. [r = v]\}}
\end{array}
\qquad
\begin{array}{c}
\text{LET} \\
\frac{\Gamma \models \{H\} t_1 \{Q'\} \quad \Gamma, v : \text{Val} \models \{Q' v\} ([v/x] t_2) \{Q\}}{\Gamma \models \{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}}
\end{array}$$

$$\begin{array}{c}
\text{IF} \\
\frac{\Gamma, - : b = \text{true} \models \{H\} t_1 \{Q\} \quad \Gamma, - : b = \text{false} \models \{H\} t_2 \{Q\}}{\Gamma \models \{H\} (\text{if } b \text{ then } t_1 \text{ else } t_2) \{Q\}}
\end{array}$$

3 Definition of an AST for Representing Programs with Invariants and Proofs

We want to benefit from the strength of both program annotation and proof derivation in our AST. From the annotated program vision we want to keep the ability to read back the program and find where to apply a transformation, and which invariants need a change. From the logical derivation vision, we want to keep the ability to give the invariants along the proof, and the foundational relation with the semantics of the programming language. In this section, we present our AST data structure expressed a Coq inductive type, which can be interpreted both as an annotated program and as a derivation.

In order to manipulate and transform invariants, we will need an explicit representation not only of the program syntax, but also of program invariants and program proofs that we can manipulate. Therefore, we need program invariants to be represented using a *deep embedding*. On the contrary, the proof leaves (i.e., proof elements that do not refer to program code) correspond to Coq proof terms, and are encoded using a *shallow embedding*. The next section explains what is a deep embedding and how it compares to a shallow embedding.

3.1 Shallow vs Deep Embeddings

To represent the syntax of a programming language in Coq (or other similar proof assistants), there are two methods. Either one defines the language as a new inductive type with a fixed

¹We can notice that the PROP and the EXISTS rules are essentially the same, we can exploit this fact encoding $[P]$ by $(\exists - : P. [])$.

set of constructors, or one directly use Coq functions. The first style is called *deep embedding* and the second one is called *shallow embedding*. The question of whether to use a deep or a shallow embedding actually applies not only to the representation of program syntax, but also to the representation of program invariants, and to the representation of proofs.

In a deep embedding, the predefined list of constructors allows for pattern matching. However, binders and contexts need to be manually managed, for example with a definition of substitution and an explicit encoding of variable (either De Bruijn or identifier strings). Similarly, typing is not provided unless explicitly implemented. In a shallow embedding however, since one defines usual Coq terms, binders and typing are directly provided by the proof assistant. As these terms can contain arbitrary elements with arbitrary types, we cannot pattern-match them, and thus have no way to inspect, modify or reuse sub-terms.

Regarding program syntax, a shallow embedding represents program functions as Coq functions, whereas a deep embedding represents it as data constructor carrying name of its arguments and a description of the body. Regarding program invariants, a shallow embedding corresponds to a term in `Prop`, whereas a deep embedding correspond to a value from a custom inductive type. Regarding proofs, shallow embedded proofs correspond to the usual Coq proof terms, whereas deeply embedded proofs correspond to trees in a custom inductive data type, where each constructor correspond to a reasoning rule. Note that deeply embedded proofs are not natively verified.

Recall that our motivation is to define program transformation. Such transformations need to manipulate the program, its invariants and its proofs. Such manipulations can include the identification or the modification of sub-terms that cannot be done with a shallow embedding. We therefore need to use deep embeddings in this work.

In our AST, program syntax is essentially represented using a deep embedding. Program invariants are also represented using a deep embedding. This corresponds to a (simplified) representation of Coq terms, as we show in Section 3.2. For representing the proof terms, we need to be able to manipulate the part of the proofs that correspond to the application of the reasoning rules of the program logic. We represent the application of these rules using a deep embedding. Regarding proof leaves, we have a choice: we can use either a deep or a shallow embedding. It would be technically possible to encode the full calculus of construction in a deep embedded way, as it is done in “Coq in Coq” [1]. However, providing a construction method for such proofs would require reimplementing all Coq tactics. Instead, we choose to represent the proof leaves as *shallowly embedded closed lemmas with deeply embedded arguments*, as we detail in Section 3.4.

In Section 3.3, we will show how we manage the explicit binders in our deep embedded proofs. In Section 3.5, we present our AST as a deep embedding of the proof with strict restrictions on the rules allowed inside the derivation. Those restrictions make the proof look like an annotated program.

3.2 A Deep Embedding of Coq Terms

As we have seen in the previous section, we need a deep embedding for expressing program invariants. Those invariants might contain arbitrary logical formulae. Therefore, we choose to represent them with a simplified deep embedding of Coq terms, as shown below.

```
Inductive coq :=
  | coq_sort (S: sort)
  | coq_val (v: val)
  | coq_var (x: var)
```

```

| coq_app (u1 u2: coq) (* <[ u1 u2 ]> *)
| coq_forall (z: bind) (Uz P: coq) (* <[ ∀z: Uz, P ]> *)
| coq_fun (z: bind) (Uz u: coq) (* <[ fun z: Uz, u ]> *)
| coq_eq (u1 u2: coq) (* <[ u1 = u2 ]> *)

```

This deep embedding mostly consists of lambda calculus, with an additional constructor (`coq_val`) to quickly include values of the manipulated programming language. The variables (type `var`) are defined as Coq strings. We prefer explicit naming to De Bruijn indices to ease printing of the terms and retain user given names. This means, however, that we need to compare terms up to alpha equivalence. The binders are defined with the type `bind` and can either contain a fresh variable name, or be anonymous and not introduce any new variable in the context. Free variables are interpreted in a global environment. This environment contains the encoding of the separation logic, and all the relevant predicates for expressing the specification, with definitions close to their CFML analogues. Advanced constructions, such as recursive functions are not handled because we are not yet sure that they will be useful for our application.

For the goals of this paper, we do not need to formalize the typing judgement of these deep embedded Coq terms. As explained in Section 5, proofs and the invariants they refer to will be ultimately type checked during their reification.

Our embedded terms can be written in a human readable manner thanks to the use of notation. For example, the predicate $\exists L. [\forall i, 0 \leq i < 5 \rightarrow L[i] = 0] \star p \rightsquigarrow \text{Mlist } L$ can be written as follows²:

```
<[ ∃'L: "list", \[ ∀'i: "nat", 0 ≤ 'i < 5 → "get" 'L 'i = 0 ] ★ "MList" 'L 'p ]>.
```

3.3 Manipulation of the Proof Environment

In our AST which represents proof trees, we call *environment* the context of all entities syntactically introduced by the parents of a node and therefore available at that point. Since this AST represents a proof that a program respects a specification, we have several kinds of entities in the context:

- Program variables, which are manipulated by the program instructions;
- Pure facts, which represent mathematical properties true at a given program point;
- Ghost variables, which are additional variables used to state specifications;
- Linear resources, which are linear heap predicates in Separation Logic; each resource can be consumed only once.

These environments can be represented by 3 distinct binding lists. One is used for program variables and is called *program environment*. Another is used for ghost variables and pure facts (which can be seen as pure variables with a type in `Prop`) and is called *pure environment*. The last one is for linear resources and is called *linear environment*.

Each element of the environment can be accessed by a name. This includes linear resources as in Iris [7]. The type of each environment entity is defined using the previously introduced deep embedding, and can refer to the entities defined before itself (considering that linear resources always come last).

²With notation such as $H \star H' \equiv \text{"hstar"} \ H \ H'$, and $\exists x: ux, H \equiv \text{"hexists"} \ ux \ (\text{fun } x: ux \Rightarrow H)$

Our AST will not contain an explicit representation of environments because their contents can be computed by traversing the AST. However, such explicit representation of the environments is used for the construction of our AST. Moreover, since nodes in the AST correspond to reasoning rules, they act on this environment, either by using or adding entities in it.

Environments are extended with bindings when, for example, entering the scope of a let-construct. Consider the program snippet `let p = ref 2 in c`, in a given environment E . Suppose that the body `ref 2` is specified using the post condition $\exists n, [\text{even } n] \star p \hookrightarrow n$. In the continuation c the environment E is extended with a program variable p , a ghost variable n , a pure fact of type `even n` and its linear environment is replaced by a single binding of type $p \hookrightarrow n$.

To assign names to these bindings, in our AST we need to introduce annotations for the binders, with square brackets to denote pure arguments and curly braces to denote linear resources.

```
let p [n: nat] [Pn: even n] {Hp: p↔n} = ref 2 in c
```

This annotated let-construct, adds p as a program variable, n and Pn to the pure environment and sets the pure environment to only contain H_p when processing the c continuation branch. Dually, it requires that the let body returns a value p such that the environment $[n: \text{nat}] [Pn: \text{even } n] \{H_p: p \leftrightarrow n\}$ can be built.

Concretely, we will represent these annotations in our AST using the record `binds` shown below.

```
Record binds := {
  binds_vars: list bind;
  binds_pure: list (bind * coq);
  binds_linear: list (bind * coq)
}.
```

Two instances of this `binds` record are also involved in function specifications: one describes the input of the function, including its pre-condition, the other describes its output, including its post-condition. Annotated functions are represented using the record shown below, where `annot_trm` is explained further on.

```
Record annot_fn := {
  annot_fn_input: binds;
  annot_fn_body: annot_trm;
  annot_fn_output: binds
}.
```

3.4 Shallow Embedding of Proof Leaves

Proof leaves appear as hypotheses of the derivation rules applied in our AST, by definition, they correspond to the proof obligations that do not directly involve the program.

Proving these leaves can involve the application of arbitrary mathematical theorems. Therefore, it is very likely that all code transformations would consider all the proof leaves as axioms, and at most reuse, maybe combine, but never modify or inspect what is inside the leaf. Because of that, we choose to encode proof leaves with a shallow embedding in our AST. This brings two main advantages: it removes the need to support most of the features of the calculus of construction in the deep embedded proof rules, and it gives the user the full set of Coq tactics to build the lemmas in the leaves.

However, incorporating a shallow embedded proof inside a deep embedded one brings some issues. Most of the proof leaf obligations refer to the deep embedded proof environment. There is no hope to use the deep embedded environment from a shallow embedded proof since the shallow proof needs to be closed to pass the type checking.

One approach could be to build a shallow lemma taking the environment as argument. This has the main drawback that environment can contain bindings of any type, and we then need to guarantee that all the accesses fetch a variable with the right type, which would be quite heavy or impractical. We prefer to implement shallow embedded proof leaves as Coq functions that take all the relevant elements of the environment as arguments.

Since our shallow embedded goals have arbitrary types, we are forced to store their proofs in dependant pairs. This dependant pair does not enforce by itself a link between the proven lemma type and the deep embedded proof obligation but we will show in Section 5 how to check this property.

Concretely, we represent prove leaves as follows, where we call `pure_lemma` a dependant pair, and `proof_leaf` a shallow embedded lemma instantiated with deep embedded arguments. These arguments are directly given in the Coq term deep embedding..

```
Record pure_lemma := {
  pure_lemma_stmt: Type;
  pure_lemma_proof : pure_lemma_stmt
}.
```

```
Record proof_leaf := {
  proof_leaf_lemma: pure_lemma;
  proof_leaf_args: list coq
}.
```

To keep maximal freedom in transformations, lemmas should only take as argument useful hypotheses; indeed, a modified piece of code could have a smaller environment than the original.

3.5 An AST Representing Both a Proof Derivation and an Annotated Program

We are now ready to present an AST that can be viewed either as a deep embedding of a proof derivation, or as an annotated program decorated with its invariants and proofs.

In our AST, we allow incomplete proofs. This means that annotations are optional. This includes, for instance, all the proof leaves.

When interpreting our AST as a proof derivation, all the constructors in our AST correspond to one derivation rule, either structural or semantic. On the contrary, when interpreting our AST as an annotated program, semantic constructors become annotated program syntax, and structural constructors are ghost instructions. In particular, if we ignore all the annotations and skip all the structural rules, we can directly recover the source code of the program.

The AST can be given by the following type, where `val_or_var` contains either a program value or a program variable name and `hyp` and `hvar` are aliases to `var` to respectively denote a name bound in the pure environment and a name bound in the linear environment.

```
Inductive annot_trm :=
  (** Semantic **)
  | annot_trm_val (tv: val_or_var)
  | annot_trm_call (fn: var) (args: list val_or_var) (spec: option (hyp*list coq))
```

```

| annot_trm_let (binds: binds) (body: annot_trm) (cont: annot_trm)
| annot_trm_if (uvc: val_or_var) (hyp_name: option bind) (brt: annot_trm) (brf:
  annot_trm)
| ... (* One rule for each construction in the program language *)
  (** Structural **)
| annot_trm_frame (frame: list hvar) (cont: annot_trm)
| annot_trm_change_pre (pure_clear: list hyp) (linear_in: list hvar)
  (output: binds) (change_lemma: option proof_leaf) (cont: annot_trm)
| annot_trm_change_post (pure_clear: list hyp) (linear_in: list hvar)
  (output: binds) (change_lemma: option proof_leaf) (cont: annot_trm)

```

The semantic constructors correspond both to annotated program constructions and to the application of a semantic rule.

- The constructor `annot_trm_val` represents a return value at the end of a program expression. It corresponds as well to the `VAL` semantic rule.

$$\frac{\text{VAL}}{\Gamma \models \{\{\}\} v \{\lambda r. [r = v]\}}$$

As this rule has no parameters except the return value itself, there is no need for any annotation on the `annot_trm_val` constructor. However, it only occurs in fully proven programs with an empty linear environment and a matching post-condition: in particular, all linear resources must have been framed out above in the AST.

- The constructor `annot_trm_let` represents a let-construct. Without annotation its `binds` argument contains only program variables. However, when annotated, it contains as well new environment bindings for the continuation, that correspond to the environment specification of the body. The `annot_trm_let` constructor also corresponds to the following `LET` semantic rule where Q' is given in the `binds` argument.

$$\frac{\text{LET} \quad \Gamma \models \{H\} t_1 \{Q'\} \quad \Gamma, v : \text{Val} \models \{Q' v\} ([v/x] t_2) \{Q\}}{\Gamma \models \{H\} (\text{let } x = t_1 \text{ in } t_2) \{Q\}}$$

- Similarly `annot_trm_if`, is both an if-construct and the application of the `IF` rule, where `hyp_name` is the name given to the new hypothesis p generated by that rule application.

$$\frac{\text{IF} \quad \Gamma, p : b = \text{true} \models \{H\} t_1 \{Q\} \quad \Gamma, p : b = \text{false} \models \{H\} t_2 \{Q\}}{\Gamma \models \{H\} (\text{if } b \text{ then } t_1 \text{ else } t_2) \{Q\}}$$

- `annot_trm_call` encodes a function call with a provided list of arguments. It does not have a corresponding rule because it is supposed to use a function specification. This specification can be given in the annotation `spec` as an instantiated lemma from the pure environment (either local or global).

Structural constructors can appear anywhere in a derivation and correspond to structural rules, or ghost instructions:

- `annot_trm_frame` constructor encodes an application of the `FRAME` rule.

$$\frac{\text{FRAME} \quad \Gamma \models \{H\} t \{Q\}}{\Gamma \models \{H \star H'\} t \{Q \star H'\}}$$

The `frame` argument contains a list of linear resource bindings that correspond to H in the rule above. H' can be automatically deduced by taking the remaining elements in the current context. Q is similarly deduced by subtracting H' from the output specification at this point.

- There is no constructor for the rules `PROP` or `EXISTS` since we apply them implicitly when adding bindings to the pure environment.
- To be able to pinpoint easily which parts of the environment are modified when using the `CONSEQUENCE` rule, we split it into two different constructors `annot_trm_change_pre` and `annot_trm_change_post`, one for each side of the Hoare triple. Moreover, we identify with `linear_in` which subset of the linear environment will be modified, and with `pure_clear` which pure bindings will be cleared in the process, while giving binding names to all the introduced entities with `output`. This corresponds to the following rules that can be derived from the `CONSEQUENCE` rule, where Γ is a pure environment, and Γ' is a subset of Γ .

$$\frac{\text{CHANGE-PRE} \quad \Gamma \models H \vdash H' \quad \Gamma' \models \{H' \star R\} t \{Q\}}{\Gamma \models \{H \star R\} t \{Q\}} \quad \frac{\text{CHANGE-POST} \quad \Gamma' \models \{H\} t \{Q' \star R\} \quad \Gamma \models Q' \vdash Q}{\Gamma \models \{H\} t \{Q \star R\}}$$

This is the only construction introducing a shallow proof leaf, to prove the heap entailment hypothesis. With an empty `linear_in` list, it can be used to add arbitrary assertions in the environment at any program point, as long as it is provable.

With Coq notation, and if we mask proof leaves, we can actually print a term of type `annot_trm` as an annotated program.

This representation of a proof derivation is a good candidate for the kind of proof manipulation required by source-to-source transformations because by replacing any sub-tree in a derivation with another valid sub-tree of the same specification, we maintain the validity of the full tree, while changing the source code of a branch. It means that transformations can be local and ignore all the derivation rules above a relevant application point.

The AST does not by itself enforce its validity: at this stage, nothing checks the typing of the environment variables or that proof leaves assert what they should. This is not a problem since we show in the two next sections how to build a self-consistent tree and how to verify its validity.

4 Building Annotated Programs

In the previous section, we described an AST that can be seen either as an annotated program or as a proof derivation. This section show that we can use these two views to build the trees. Given a program annotated with invariants but without proofs, one may want to build the same annotated program with its proof leaves filled in, using the annotation and extraction workflow presented in section §2.1. Besides, given a program without any annotation, one may want to

interactively build, in CFML fashion, the proof derivation that ensures the program follows its specification, as presented in section §2.2.

We explain how to perform both kinds of tasks, interactively in Coq. We start by presenting the CFML approach. Indeed, we will see how the case where program invariants are already present can also be handled with CFML style tactics, with the only difference being taking into account the existing invariants.

4.1 Building Interactively

We first describe how to interactively build a derivation, in CFML fashion, since we reuse most of the components in our proposed system.

From the user point of view, a CFML proof follow a simple workflow. At each step, the theorem prover shows the program that still need to be processed along with the environment at this program point. It also shows what invariants need to hold after the execution. Seeing these goals, the user will apply CFML tactics to apply structural of semantic rules at the current program point. These tactics produce sub-goals corresponding either to sub-programs verification or proof leaves.

Internally, these CFML tactics call shallow embedded lemmas. These lemmas produce a Coq proof term corresponding to a proof in the shallow embedding of Separation Logic and program semantics. Since this output proof term is in `Prop`, there is no simple way to inspect it, to extract an instance of our AST. Technically Ltac manipulation, to transfer this shallow proof in a deep embedding of Coq is possible (as it is done in the MetaCoq [13] plugin for instance), but the proof terms would have no canonical form anyway, so we cannot convert them into our AST.

Instead, we provide our own set of tactics. These new tactics display the same kind of goals as CFML and generate the same kind of proof obligations. However, their output is an AST in our format. This AST corresponds to a deep embedding of the derivation, except for proof leaves which are shallowly embedded.

The challenge in designing those tactics is that the proof obligations that we would like to show to the user are propositions, in `Prop`. However, their output is an instance of our AST, in `Type`. There is no fundamental difficulty, we simply need to find an appropriate way to state the proof obligations on which the tactics apply. To that end, we introduce a predicate, written `refine_pred E t F \hat{t}` , where t is the input program, E and F correspond to pre- and post-conditions represented as environments with named elements; and \hat{t} is an *eval* that gets refined along the proof. When the proof is completed the \hat{t} variable gets instantiated with the desired output.

We display the `refine_pred E t F \hat{t}` predicate to the user using a notation that hides \hat{t} . What remains looks seemingly like a CFML proof obligation. We show below an example where E and F are displayed with a notation that separates the three components of the environments (program variable, pure facts, and linear resources).

```
PRE
  'x, 'p
  ----
  ['n : int]
  ["Px": 'x = 3]
  ["Pn": 'n = 1]
  ----
  {"Pp": 'p  $\leftrightarrow$  'n}
```

```

CODE
  let `y = get `p in
  `x + `y
POST
  `r
  ----
  ["Pr": 4 = `r]
  ----
  {"Pp": `p ↔ 1}

```

This example corresponds to the following Hoare triple:

$$\{\exists n.[x = 3] \star [n = 1] \star p \leftrightarrow n\} \text{let } y = \text{get } p \text{ in } x + y \{\lambda r.[4 = r] \star p \leftrightarrow 1\}.$$

During an interactive proof all goals except proof leaves are displayed using such notation for `refine_pred`.

Reasoning steps of separation logic in such proofs are carried out using CFML style tactics. For instance, the tactic `xlet` `["Py": `y = 3] {"Pp": `p ↔ 1}` applies to the proof obligation shown above. As in CFML, this tactic generates two sub-goals (one for the body, one for the continuation). The argument provided to `xlet` describes the post-condition for the body. It is optional, and if not provided, an evar is introduced for this post-condition, which like in CFML gets instantiated during the verification of the body.

Internally, `xlet` applies to a goal of the form `refine_pred E (let x = t_body in t_cont) F t̂`. It produces two sub goals `refine_pred E t_body E' t̂_b` and `refine_pred E'' t_cont F t̂_c`, where `t̂_b` and `t̂_c` are newly generated evars, and `E''` corresponds to the concatenation of `E` with `E'` in the program and pure environments and only `E'` in the linear environment. While doing so, it instantiates `t̂` as `(let x E' = t̂_b in t̂_c)`.

At some points in the proof, the user reaches proof leaves, that is, proof obligations not establishing Hoare triples. Such proof leaves are represented in our AST using the type `proof_leaf` introduced in Section 3.4. Recall that `proof_leaf` contains a *closed* shallowly embedded lemma and a list of deeply embedded arguments.

Our framework present proof leaves obligations as the predicate `prove E G p̂`, where `E` is an environment without linear resources, `G` is the deep embedding of the property we want to prove and `p̂` is an evar for the desired value of type `proof_leaf`. We provide a tactic `xprove` that applies to a goal of the form `prove E G p̂`. This tactic takes as argument a subset of the keys of `E` that corresponds to the hypotheses the user wants to exploit to establish a proof of `G`. The tactic then leaves a standard Coq proof obligation of a shallow version of `G` obtained using the reification mechanism described further on in Section 5. The resolution of this goal yields a proof term, which ends up being recorded in our AST.

4.2 Completing an Annotated Program with Proofs

In the previous section, we saw how to build an AST fully annotated with invariants and proofs starting from an unannotated program. In this section, we explain how to build a similar AST starting from a program annotated with invariants. Concretely, we need to fill the missing proof leaves. This consists of extracting proof obligations and proving them, like e.g. in `Why3`.

Our implementation leverages the same mechanisms based on `refine_pred` as in the previous section. The key difference is that the source program contains invariants. In particular, there is no need to provide a post-condition when calling tactics such as `xlet` because this post-condition already appears as a program annotation. More generally, CFML style tactics, can

all be invoked without arguments simply following the structure of the annotated program. As a result, all the tactics processing the code can be automatically handled.

A number of these tactics produce proof leaves. These proof leaves correspond to what is called extracted proof obligations in the tools taking the annotation and extraction approach.

There is another application of refining a partially complete AST into a fully complete one, related to program transformations that we intend to work on in the long term. The user may wish to introduce an arbitrary piece of code into an already verified program. Then, thanks to this construction method, we can extract proof obligations concerning exclusively the new piece of code.

5 A Reification Process to Validate our AST

The AST we presented in Section 3, does not include any guarantees about the validity of the proof it represents. Indeed, nothing prevents using an unbound environment variable or using a rule that do not apply at a given program point. In Section 4, we gave a methodology to build derivations. The derivations built that way are in principle well formed. However, their validity directly depends on the implementation of the construction tactics. Exactly like in Coq, we do not want our tactics to be inside the trusted code base, since they can have rather complicated implementations, and therefore may contain bugs.

A standard approach to provide a stronger validity guarantee on deep embedded proofs is to define a validity predicate. Such predicate ensures that each node in a derivation correspond to valid rule application. This is what we initially tried, but we quickly noticed that establishing and maintaining a validity predicate across transformations would require a lot of work.

To keep strong guarantees without too much effort, we introduce in this section a reification procedure. This will translate a complete proof derivation expressed in our AST into a standard shallow embedded CFML lemma. The obtained Coq proof is therefore directly using the defined semantics of our programming language, in a foundational way.

During reification, both the proof derivation and the invariants represented inside are translated in their shallow embedded counterparts. For specifying the type of the root of the derivation, we also extract an unannotated version of the program syntax as a CFML deeply embedded program term.

One technical aspect of the reification process is that we need to maintain the mapping between binding names from the AST environment, and corresponding Coq variables that appear in the produced proof term. This can be done by maintaining, during the reification process, a table between names in the deep embedding and identifiers in the shallow version. Since invariants can also refer to a global environment, we also need to provide a Coq implementation for each of these global terms.

When reaching proof leaves, our reification procedure will use the pure lemma stored inside the leaf and call it with reified versions of its deeply embedded arguments. This is enough to force the Coq type checker to verify that the lemma indeed proves the obligation it is applied on.

Concretely, the reification process is implemented as a recursive Ltac function parameterized by the mapping between binding names and Coq variables. On the proof parts, it applies rules expressed as CFML lemmas simply following the derivation. For invariants, it follows the structure of the deeply embedded term to progressively refine the corresponding shallow embedded Coq term.

In order to improve the user experience, when extracting program invariants, we want to reuse the names of the deeply embedded binders for the shallow embedded ones. This is possible

only if our Ltac procedure can convert a Coq string into a Coq identifier. Such translation cannot be expressed natively in Ltac but can be implemented using a Coq plugin or a piece of Ltac2 code.

In the end, the Ltac function implementing the reification process is not part of the trusted code base. What goes in the trusted base is the Coq kernel, the language semantics, and the definition of triples regarding the semantics. In addition, for each program, one needs to trust that the shallow embedding of the specification is correct. This shallow specification can be read as the type of the reified verification proof, but it also corresponds to a shallow embedded version of the specification that appears at the root of the annotated AST.

6 Conclusion

In this paper, we aimed at providing a representation for programs annotated with their invariants and proofs, such that we can easily manipulate them.

We achieved that using a deep embedding of the proof derivation with shallow proof leaves and explicit environment manipulations. We provided a method for constructing these annotated programs either interactively or by filling missing proofs.

In the long term, we want to propose an interactive framework for code transformation that preserves proofs and can be used for source-to-source optimization. Using it, one should be able to write a naive version of a program, verify it, and then progressively transform it into a highly optimized code. After the last step we should be able to provide a proof of the final program. By combining such framework with verified compilers such as CakeML [8] or CompCert [9], we could then obtain binaries that are both trustworthy and efficient.

References

- [1] Bruno Barras and Benjamin Werner. Coq in coq. *Available on the WWW*, 1997.
- [2] Arthur Charguéraud. Separation logic for sequential programs (functional pearl). *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.
- [3] Arthur Charguéraud, Begatim Bytyqi, Damien Rouhling, and Yann A Barsamian. OptiTrust: an Interactive Framework for Source-to-Source Transformations. working paper or preprint, September 2022.
- [4] Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11*, page 418–430, New York, NY, USA, 2011. Association for Computing Machinery.
- [5] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792, pages 125–128, March 2013.
- [6] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In Sophia Drossopoulou, editor, *Programming Languages and Systems*, page 353–367, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
- [7] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the Foundations of the Rust Programming Language. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.
- [8] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A Verified Implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, page 179–191, New York, NY, USA, 2014. Association for Computing Machinery.

- [9] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, jul 2009.
- [10] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. Viper: A Verification Infrastructure for Permission-Based Reasoning. In Barbara Jobstmann and K. Rustan M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation*, page 41–62, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
- [11] O’Hearn, Reynolds, and Yang. Local reasoning about programs that alter data structures. In *CSL: 15th Workshop on Computer Science Logic*. LNCS, Springer-Verlag, 2001.
- [12] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82:77–97, 2014. Special Issue on Automated Verification of Critical Systems (AV-oCS’11).
- [13] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, 64(5):947–999, Jun 2020.

Retrofitting OCAML modules

Fixing signature avoidance in the generative case

Clément Blaudeau¹, Didier Remy¹, and Gabriel Radanne²

¹ Cambium, Inria, France

² CASH, Inria, EnsL, UCBL, CNRS, LIP, France

Abstract

ML modules offer large-scale notions of composition and modularity. Provided as an additional layer on top of the core language, they have proven both vital to the working OCaml and SML programmers, and inspiring to other use-cases and languages. Unfortunately, their meta-theory remains difficult to comprehend, requiring heavy machinery to prove their soundness. Building on a previous translation from ML modules to F^ω , we propose a new comprehensive description of a generative subset of OCAML modules, embarking on a journey right from the *source* OCAML module system, up to F^ω , and back. We pause in the middle to uncover a system, called *canonical* that combines the best of both worlds. On the way, we obtain type soundness, but also and more importantly, a deeper insight into the *signature avoidance problem*, along with ways to improve both the OCAML language and its typechecking algorithm.

1 A powerful but imperfect module system

Modularity is a key technique to break down a complex program into parts at different levels of abstraction. Instead of dealing with technical details and complex invariants at all times, programmers can split the code-base into manageable parts, called *modules*, and structure the relationship between those modules by specifying their interfaces and interactions. Code might be packed into a module to make a component, such as the implementation of a data-structure, reusable and often polymorphic—effectively factorizing development. Controlling the interactions between modules through a cautious choice of interfaces is not only a tool for correctness; using abstraction, it is also a way to enforce invariants and to allow for several teams of developers to work independently on different parts of the same program while enjoying a language-level guarantee that they respect the invariants of other parts.

A wide variety of techniques can be used to apply modularity concepts to software development: simple compilation units, classes, packages, crates, etc. In languages of the ML-family, modularity is provided by *modules*, which form a language layer built on top of the core language. The interactions between modules are controlled statically by a strict type system, making modularity work in practice and with little run-time overhead. A module is described by its interface, called a *signature*, which serves as both a light specification and an API.

The OCAML module system is especially rich and still under development for new features. It provides both developer-side and user-side abstraction mechanisms: developers can control the outside view of a module by explicitly restricting its interface, while users can abstract over a module with a given interface using a *functor*. The signature language allows to both *restrict* and *control* the interface of a module, specifically by *hiding* fields (which corresponds to the distinction between *public* and *private* fields in Object-Oriented Programming), or by abstracting type components—keeping types accessible while hiding their definitions.

The OCAML module system is renowned for its expressiveness, ease of use, and the properties it can statically enforce. All sizable OCAML projects use modules to access libraries or define

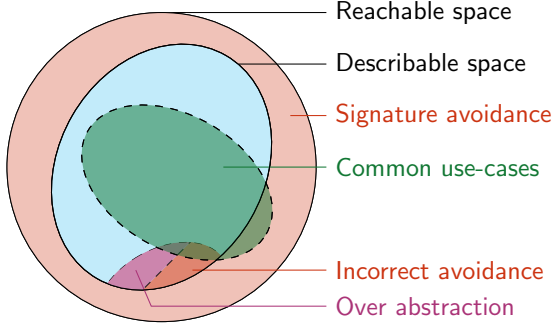


Figure 1: A representation of the mismatch between the *reachable space* of module expressions (outer-most circle) and the *describable space* of signatures (inner ellipse). The common use-cases of OCAML are mainly within the area where the type-checker behaves correctly. In some cases, the current OCAML typechecker can lose type-equalities while still being in the describable space. This may lead to (1) producing a signature where some type fields are unnecessarily made abstract (over-abstraction) or (2) failing at inferring the signature (incorrect avoidance).

parametric instances of data structures (sets, hashtables, streams, etc.). Several successful projects have made heavy use of modules, as in MirageOS [Madhavapeddy et al., 2013] where modules and functors are even assembled on demand using a DSL [Radanne et al., 2019]. Despite the successes and the interest of the community regarding ML modules, giving it a formal type-theoretic definition and establishing its properties has proven to be a difficult task.

Besides the academic interest, having a formal semantics is made even more necessary to envision extensions such as modular implicits [White et al., 2014] where new modules could be built automatically from their signatures by applications of functors to other modules.

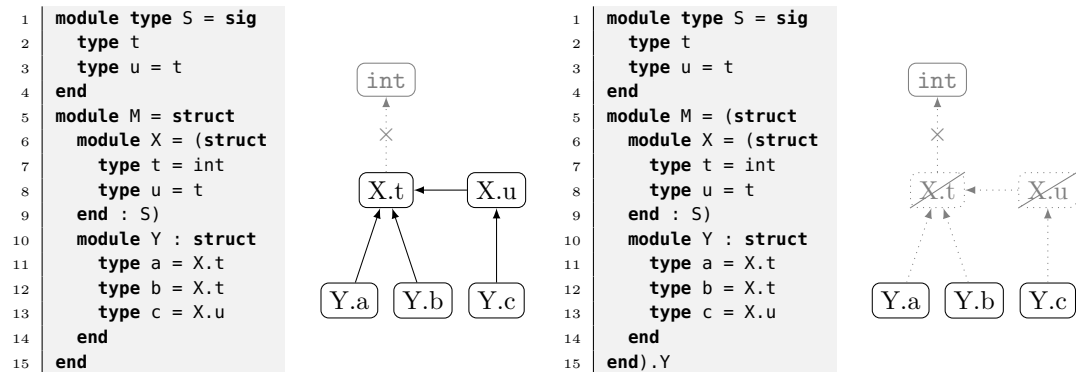
1.1 The signature avoidance problem

The *signature avoidance* problem is a key issue of ML-module systems. It originates from a mismatch, illustrated in Figure 1, between the expressiveness of the module and signature languages: the *reachable space* of possible module expressions is larger than the *describable space* of signatures: some modules can't be described by a signature. This issue can be solved either by sticking to the describable space and ensuring that the typechecker correctly covers it or by extending signatures to make the reachable and describable spaces coincide.

This mismatch is caused by the interaction of three mechanisms. First, type abstraction, which is key to control access and protect invariants by typing, creates new types that are only compatible with themselves (or aliases of themselves). Second, sharing abstract types between modules, which is essential for module interactions, produces trees of type-alias equalities. Finally, hiding type components (through either explicit projections or implicit subtyping during functor applications) can remove abstract type aliases from scope, while other components (values or types) kept *in scope* may refer to them (for instance, removing an abstract type t while keeping a value of type $t \text{ list}$). In some situations, there is no possible signature to infer for a module; in other situations, there are several incompatible ones.

We illustrate this abstraction mechanism in Example 1. Each source code comes with a representation of the type equalities and aliases as a *tree of type equalities*. The connected components of the tree represent equal, interchangeable types that all belong to the same equivalence class. They can be rooted with base types (`int`, `bool`, `string`, etc.) or constructed over other types (like `int list`, `int \rightarrow int`, `int \times bool`, etc.) or previously defined *abstract types*. Abstracting a type effectively removes a link in the type-sharing tree, which splits a connected component.

Example 1 precisely illustrates this potential loss of type sharing when using module oper-



Example 1: Two examples of modules and associated type-sharing trees.

ations. On the left-hand side, restricting the signature of the module X to the module type S removes the link between $X.t$ and int (grayed out in the type-sharing tree): this becomes hidden to the typechecker outside of the body of the module X . On the right-hand side, the projection on the submodule Y removes $X.t$ and $X.u$. The types $Y.a$, $Y.b$, and $Y.c$ are pointing to *out-of-scope* types: they must be changed or deleted. This shows why expressing membership to an equivalence class through paths is fragile: the removal of type fields can split connected components apart, resulting in a loss of type-sharing or incorrect signatures.

Strategies for solving signature avoidance When a type declaration is referring to an out-of-scope type, there are mainly three strategies to correct the signature: (1) abstracting the type (effectively ignoring the previous link in the tree), (2) rewriting the type equalities using in-scope aliases (effectively rewiring the tree to maintain connected components), or (3) extending the signature syntax with existential types. The first strategy can lead to loss of type sharing, but is easy to implement—it is the one currently in use in the OCAML typechecker. The cases where the second strategy succeeds constitute the *solvable* cases of signature avoidance. The OCAML type-checker only tries to follow directed edges of the tree until it finds an accessible type, but does not follow reverse edges and thus does not have a notion of *connected components*. In [Example 1](#), it would fail at finding that $Y.a$, $Y.b$ and $Y.c$ are aliases. Sometimes, no *in-scope* alias is available and signature avoidance cannot be solved without an extended syntax: those are the *general* cases of signature avoidance.

Signature avoidance in practice OCAML developers usually get around this limitation by explicitly naming modules before using them, which adds always-accessible root points to the graph. The module syntax of OCAML actually encourages this approach by limiting the places where inlined, unnamed modules and signatures can be used. In particular, projection on an unnamed module (as done in [Example 1](#)) is forbidden. However, explicit naming is sometimes cumbersome, which can limit the usability of module-based programming patterns such as modular implicits. It also prevents a fine-grained management of types shown in public APIs.

1.2 Related work

Previous work has provided solid formal foundations for ML-modules. The link between abstract types in ML-module systems and existential types in F^ω was already explored by [Mitchell](#)

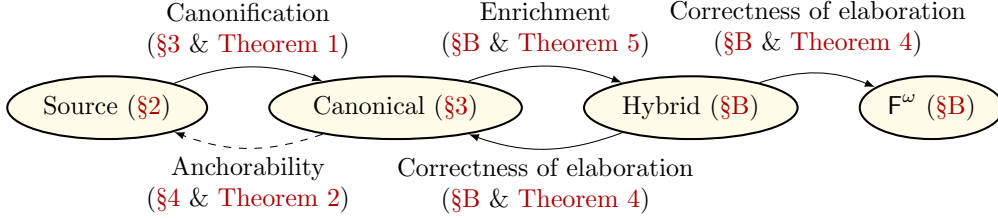


Figure 2: The different systems: their links, with theorems and sections.

and Plotkin [1985]. This vision was opposed by MacQueen [1986], who considered existential types to be too weak, and proposed using a restriction of dependent types (strong sums) to describe module systems. Further work, notably on phase separation by Harper et al. [1989], supported the idea that dependent types may actually be too powerful (thus, unnecessarily complex) for module systems. SML modules were first described by Harper et al. [1989]. Two approaches for the formalization and improvement of abstract types in SML were later concomitantly described by Leroy [1994] (through manifest types) and Harper and Lillibridge [1994] (via an adapted F^ω with translucent sums). The OCAML module system itself was specified by Leroy [1994, 2000], and later with an extension to applicative functors [Leroy, 1995].

The use of existential types to interpret signatures by Russo [2004] opened the door for a simplified link between modules and F^ω . Type generativity is also explored by DREYER [2007], using stamps in place of existential types. A similar, but logically-based approach was later developed by Montagu and Rémy [2009] introducing the concept of open existential types. Pushing Russo’s idea further, a closer correspondence between ML-modules and F^ω was achieved by Rossberg et al. [2014] by elaborating a significant subset of the SML module syntax into F^ω . The type system is safe by construction, inheriting the property from F^ω . A limit of this approach is that, since the definition of modules is only given by translation, the programmer must think in terms of the elaboration and only sees the elaborated types instead of the usual signatures. This makes direct reasoning on the source program more difficult. Their work, called the *F-ing approach*, also covered some difficult points like applicative functors and first-class modules, and was partially mechanized in Coq. Moving one step further, Rossberg [2018] achieved a unification of the core and module languages (thus, un-stratified) using F^ω as the underlying programming language and seeing module constructs as syntactic sugar. More recently, Cray [2020] used involved focusing techniques to solve the signature avoidance problem in the singleton-type approach (for SML modules) in a manner that turns out to have some similarities with that of *F-ing*.

1.3 A journey into OCaml modules

Our approach is strongly inspired by previous works, which we re-explore in the context of OCAML modules. Our goal is to provide a simpler and comprehensive type system for OCAML modules, which could first serve as a specification, then help correct and improve the current implementation, and, finally, serve as a basis for future extensions of the module language. Our contributions are summarized in Figure 2.

We start by presenting a (mostly standard) self-contained specification of the OCAML module system in §2, adapted from previous works. Critically, like the current OCAML implementation, this presentation suffers from the signature avoidance problem.

Then, we build on the insights of the *F-ing* [Rossberg et al., 2014] translation into F^ω to present a new set of typing rules. However, reasoning only *by elaboration* in F^ω is hard, as the elaboration introduces encoding layers and, crucially, uses a very different way of handling

abstract types, making the correspondence between source and F^ω terms and types rather obscure. Instead, as hinted in the work of Russo [2004], we introduce a light extension of the syntax for signatures with F^ω -style quantifiers, which we call *canonical signatures*. They act as a middle point between the path-based approach of the source and the quantified approach of F^ω , effectively splitting the translation effort in two steps. We introduce in §3 a new type system for OCAML modules, called *canonical*, where the source module expressions are typed with canonical signatures and source signatures translated into canonical ones. This system is simpler than the source one and doesn't suffer from the signature avoidance problem. Using our one-way translation, we revisit and explain the ad-hoc techniques (strengthening, equivalence) of the source presentation. We finally give the main result of this section: all source typing derivations can be translated into canonical typing derivations (Theorem 1).

The reverse translation of signatures (from the canonical system to the source one) is not always possible, as the former is more expressive than the latter. In §4 we explore *when and how* we can translate canonical signatures back into OCAML signatures, which uncovers precisely why the signature syntax lacks expressivity. This reverse process, called *anchoring*, is one of our main contributions. We then show that we can restrict the canonical system at one specific point to *mimic* the source system and only produce signatures that are valid regarding the source typing (Theorem 2). Some details are also given in the appendix §A.

The link between the canonical system and F^ω is conceptually easier while presenting some technical challenges. For lack of space, it is only detailed in the appendix §B. We introduce an *hybrid* system that produces both canonical and F^ω objects in correspondence. The hybrid system acts a central justification, as it can yield both the canonical system and the *F-ing* style elaboration by projections. The canonical system was actually designed by erasing the terms from the F^ω encoding which can still be seen as implicit *proof terms* justifying the canonical rules.

In this work, we restricted our system to a generative subset of the language. We believe our approach also extends to applicative functors and other features (first-class modules, module aliases, abstract signatures, etc.), which are left for future work. Our long term goal is to extend OCAML signatures following the canonical system, thus completely solving the signature avoidance problem. The canonical system would then be a standalone *source* system which can be used by programmers to reason about OCaml programs, with the F^ω elaboration ensuring its soundness.

2 A generative subset of the OCaml module system

In this section we present a module system that models a generative subset of OCaml. We present its grammar, several auxiliary judgments used, and end with its typing judgment.

2.1 Grammar and syntactic choices

Figure 3 gives the grammar for a generative subset of the module language. The language of modules is built on top of a core language of expressions e and types τ which we leave abstract, except for value identifiers x and type identifiers t , so that we can extend expressions with qualified variables and types with qualified types.

Syntactic choices The language of module expressions and signatures is rather standard, except for a few minor design choices. We consider the following conventions: module related meta-variables use *uppercase letters*, M , X , etc. while lowercase letters are used for expressions

Path		Signature	
$P ::= A.X$	(Direct access)	$S ::= Q.T$	(Module type)
$P.X$	(External Access)	$(Y : S) \rightarrow S$	(Functor signature)
Y	(Functor parameter)	$\text{sig}_A \overline{D} \text{ end}$	(Structural signature)
$Q ::= A P$	(Prefix)	Declarations	
Module Expression		$D ::= \text{val } x : \tau$	(Value)
$M ::= P$	(Path)	$\text{type } t = \tau$	(Type)
$M.X$	(Projection)	$\text{module } X : S$	(Module)
$(P : S)$	(Sealing)	$\text{module type } T = S$	(Module type)
$(Y : S) \rightarrow M$	(Functor)	Environment	
$P(P')$	(Application)	$\Gamma ::= \emptyset$	(Empty)
$\text{struct}_A \overline{B} \text{ end}$	(Structure)	$\Gamma, (Y : S)$	(Functor Argument)
Binding		$\Gamma, (A.I : D)$	(Declaration)
$B ::= \text{let } x = e$	(Value)	Core language	
$\text{type } t = \tau$	(Type)	$e ::= Q.x$	(Qualified variable)
$\text{module } X = M$	(Module)	\dots	(Other expression)
$\text{module type } T = S$	(Module type)	$\tau ::= Q.t$	(Qualified type)
Identifier		\dots	(Other type)
$I ::= x t X Y T$	(Any identifier)		

Figure 3: Syntax of the module language

and types of the core language. Lists are written with an overhead bar: \overline{D} is a list of D . In order to simplify the treatment of scoping and shadowing, we use *self-references*, ranged over by letter A , in both structures and signatures to refer to the current object; their binding occurrence appears as a subscript to the structure or signature they belong to, so that self-references can freely be renamed. *Abstract types* are specified as types pointing to themselves, e.g., $\text{type } t = A.t$ where A is the self-reference of the current structure. In typing contexts Γ , identifiers, denoted by I , are prefixed by the self-reference A of the structure or signature they are meant to belong to. As an implicit convention, in entries $A.I : D$, the identifier extracted from D is I . We introduce prefixes, written with the letter Q , to range over either a path P or a self reference A . We use a distinct class of variables, written Y , for functor parameters, which can be freely renamed. By contrast, and as usual with modules, neither identifiers X and T for module expressions and signatures, nor the identifiers x and t for core language expressions and types can be renamed, as they also play the role of an external name.

Projections and accesses Several restrictions are built into the grammar: field accesses inside a module type are forbidden, since prefixes Q may not originate from a module type identifier T ; and paths cannot contain functor applications, which makes the system fully *generative*. We allow projection on any module expression, but we restrict functor application to paths. Current OCAML does the opposite, mainly to prevent cases prone to trigger signature avoidance. Our choice is more general, as the OCAML one can be encoded, while the converse requires an explicit signature annotation on the functor argument.

Omitted constructs In addition to restricting the system to be generative, we omit some constructs for the sake of simplicity: the `include` operator, explicit constraints S with $\text{type } t = \tau$

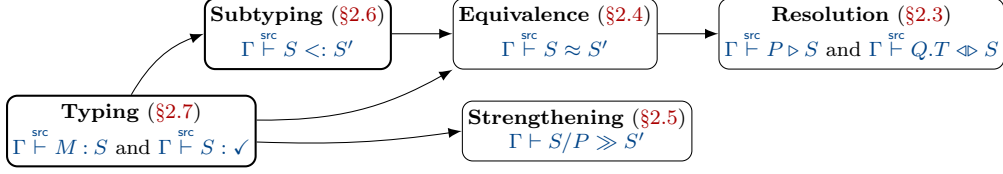


Figure 4: Structure of judgments for the source system

(resp. `module X = P`) and deleting constraints S with type $t := \tau$ (resp. `module X := P`). We believe that they do not impact the overall structure of the system, only adding more cases in the set of rules. We did not include first-class modules.

2.2 System structure and judgments

The typing judgment uses several auxiliary judgments whose dependencies are presented in [Figure 4](#). The system we present revolves around two main components: *typing* (for module expressions and signatures) and *subtyping*. The *path-based* representation of type sharing in OCAML modules requires us to define two more judgments: *equivalence* and *strengthening*. Finally, as retrieving signatures in the environment is non trivial, we define a pair of *resolution* helper judgments. We present and explain these judgments in reversed order of dependency in the subsequent subsections.

2.3 Resolution

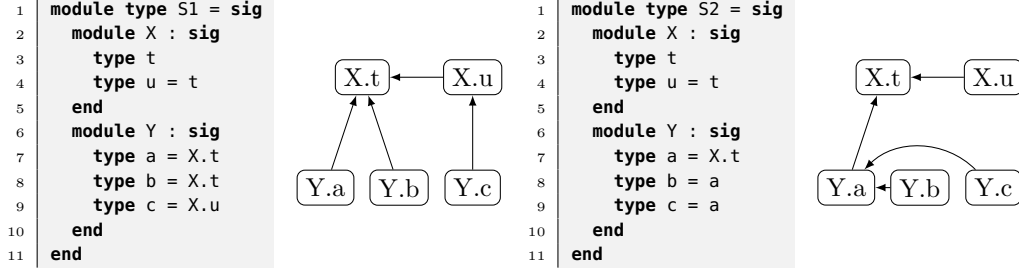
In the OCAML module system, retrieving the signature of a given path (of a module stored in the environment) is not as simple as a lookup. First, objects are stored with multiple levels of indirection (like $X.X'.X''$) which requires to inspect intermediary signatures. Secondly, every signature can be a module type variables, which must also be resolved to its definition before accessing a field. For this purpose, we recursively define a *path resolution* judgment $\Gamma \vdash^{\text{src}} P \triangleright S$ which gives the signature S of a path P and a *signature resolution* judgment $\Gamma \vdash^{\text{src}} Q.T \diamond S$, which retrieves the definition S of a module type $Q.T$ (both in an environment Γ).

We show below two examples of resolution rules. Rule **S-RES-MODTYPE** links path resolution and signature resolution: if the path resolution of P returns a module type $Q.T$, the definition of $Q.T$ should be resolved to a signature S . Rule **S-RES-PROJ-MOD** accesses a submodule X of a module path P . This rule illustrates the transformation of *local* links in the signature S (that might refer to other components of the module at P via its self-reference A) into *absolute* ones by substitution of the self-reference A for the path P . The full set of rules is given in [§C.1](#).

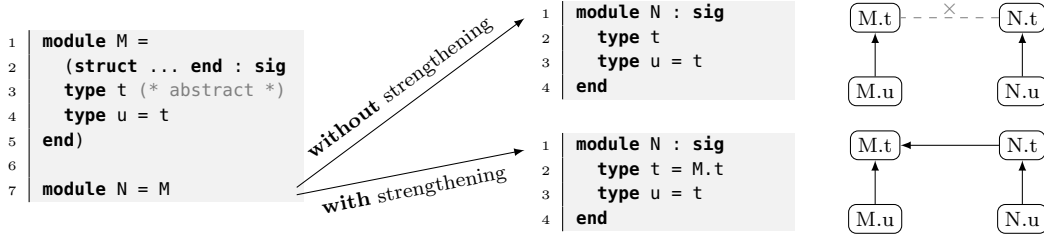
$$\begin{array}{c}
 \text{S-RES-MODTYPE} \\
 \frac{\Gamma \vdash^{\text{src}} P \triangleright Q.T \quad \Gamma \vdash^{\text{src}} Q.T \diamond S}{\Gamma \vdash^{\text{src}} P \triangleright S}
 \end{array}
 \qquad
 \begin{array}{c}
 \text{S-RES-PROJ-MOD} \\
 \frac{\Gamma \vdash^{\text{src}} P \triangleright \text{sig}_A \overline{D} \text{ end} \quad (\text{module } X : S) \in \overline{D}}{\Gamma \vdash^{\text{src}} P.X \triangleright S[A \mapsto P]}
 \end{array}$$

2.4 Equivalence

In the path-based approach of OCaml, several paths can resolve to the same type, as illustrated in [§1.1](#). A type definition can have several *equivalent* expressions, called aliases. This leads us to consider a notion of *type equivalence* based on the existence of a common ancestor when browsing the type sharing tree of aliases. This can be seen in the rules **S-EQV-TYPE-RES** and



Example 2: Two equivalent signatures S_1 and S_2 : they define two *different* type sharing trees that yet have the same connected components. Specifically, two types are aliases in S_1 if and only if they are also aliases in S_2 .



Example 3: Strengthening and its graphical interpretation

S-EQV-TYPE-LOCAL: a type is equivalent to its definition (and only to itself for abstract types). With transitivity, symmetry, and reflexivity, we get an equivalence relationship on types.

$$\begin{array}{c}
 \text{S-EQV-TYPE-RES} \\
 \frac{\Gamma \vdash^{\text{src}} P \triangleright \text{sig}_A \overline{D} \text{ end} \quad (\text{type } t = \tau) \in \overline{D}}{\Gamma \vdash^{\text{src}} P.t \approx \tau[A \mapsto P]} \\
 \\
 \text{S-EQV-TYPE-LOCAL} \\
 \frac{(A.t : \text{type } t = \tau) \in \Gamma}{\Gamma \vdash^{\text{src}} A.t \approx \tau}
 \end{array}$$

Building on this type equivalence, we define *signature equivalence*: two signatures are equivalent if we can go from the first one to the second one by only substituting type aliases. This is illustrated by the two signatures in [Example 2](#). The full set of rules is given in [§C.2](#).

2.5 Strengthening

The need for strengthening is illustrated in [Example 3](#). When aliasing the module M on line 7, the inferred signature for N cannot be straightforwardly obtained by copying the signature of M without loss of type-sharing. Indeed, M defines an abstract type which is a root of a connected component of the underlying type-sharing tree. Duplicating the signature duplicates the root, effectively creating a separated connected component and thus, a new abstract type. The known solution to this problem is to use a *strengthening* operation $\Gamma \vdash P/S \gg S'$ of a signature S by a path P , producing S' , a *strengthened* version of S where each abstract type field has been rewritten into a concrete type field pointing to its original definition in P .

Rule **S-STR-SIG-SIG** shows the case for declarations inside a signature. Our technical choice of having abstract types represented as pointers to themselves (`type t = A.t`) makes the definition of strengthening easy, as substituting the self-reference for the path suffices to transform local links into absolute ones (`type t = P.t`). Thus, strengthening does not change declarations, except

for submodules (rule **S-STR-DECL-MOD**) where the signature is strengthened with the extended path $P.X$. Functor signatures are not strengthened, as they do not introduce new abstract types. The full set of rules is given in §C.3.

$$\frac{\text{S-STR-SIG-SIG} \quad \Gamma \vdash \overline{D}[A \mapsto P] / P \gg \overline{D'}}{\Gamma \vdash \text{sig}_A \overline{D} \text{ end} / P \gg \text{sig}_A \overline{D'} \text{ end}} \quad \frac{\text{S-STR-DECL-MOD} \quad \Gamma \vdash S / (P.X) \gg S'}{\Gamma \vdash (\text{module } X : S) / P \gg \text{module } X : S'}$$

2.6 Subtyping

The subtyping judgment has a fundamental role in the OCAML module system, as it allows the user to define an abstract (polymorphic) interface and assign it to a module that has a richer signature than this interface. In OCaml, some (but not all) subtyping operations have computational content: the removal and reordering of fields implies a runtime copy of the memory representation of the module. We thus distinguish two operations: *Subtyping by abstraction*, written $\Gamma \vdash^{\text{src}} S_1 \prec: S_2$, which has no computational content and *Subtyping*, written $\Gamma \vdash^{\text{src}} S_1 \prec: S_2$, which includes the former and also allows for removal and reordering of fields and therefore has some computational content. Both judgments have the same rules, except for the comparison of structural signatures.

Subtyping on signatures enforces a structural match (up to name resolution): functor signature against functor signature and structural signature against structural signature. The latter case is where the only distinction between the two modes of subtyping appears: in subtyping by abstraction, the two lists of declarations \overline{D} and $\overline{D'}$ are compared directly as shown in rule **S-SUB-SIG-SIG**. In the general form of subtyping, a subset \overline{D}_0 of the declaration list \overline{D} is compared with $\overline{D'}$, allowing for reordering and deletion, as shown in Rule **S-SUB-SIG**. Using a subset instead of a subsequence allows for reordering.

$$\frac{\text{S-SUBEQ-SIG-SIG} \quad \Gamma, \overline{A.D} \vdash^{\text{src}} \overline{D} \prec: \overline{D'}}{\Gamma \vdash^{\text{src}} \text{sig}_A \overline{D} \text{ end} \prec: \text{sig}_A \overline{D'} \text{ end}} \quad \frac{\text{S-SUB-SIG-SIG} \quad \overline{D}_0 \subseteq \overline{D} \quad \Gamma, \overline{A.D} \vdash^{\text{src}} \overline{D}_0 \prec: \overline{D'}}{\Gamma \vdash^{\text{src}} \text{sig}_A \overline{D} \text{ end} \prec: \text{sig}_A \overline{D'} \text{ end}}$$

In both rules, the subtyping is done *declaration by declaration* (independently), in an environment that contains all the declarations of the richer (left-hand-side) signature. This approach matches well with our representation choice of abstract types, allowing us to have only one rule for comparing both abstract types and concrete types declarations (rule **S-SUB-DECL-TYPE**). The other subtyping rules for declarations are straightforward, and depends on signature subtyping. As a module type field can be used in both covariant and contravariant positions (as functor arguments for example), the rule for subtyping a module type declaration checks subtyping in both directions. In §C.4 and in the following, the rules that are similar between abstraction and general subtyping are given with the symbol $\prec:$ being either $\prec:$ or \prec :

$$\frac{\text{S-SUB-DECL-TYPE} \quad \Gamma \vdash^{\text{src}} \tau \prec: \tau'}{\Gamma \vdash^{\text{src}} (\text{type } t = \tau) \prec: (\text{type } t = \tau')} \quad \frac{\text{S-SUB-DECL-MODTYPE} \quad \Gamma \vdash^{\text{src}} S \prec: S' \quad \Gamma \vdash^{\text{src}} S' \prec: S}{\Gamma \vdash^{\text{src}} (\text{module type } T = S) \prec: (\text{module type } T = S')}$$

2.7 Typing

Once all the auxiliary judgments have been laid out, the typing judgment is mostly straightforward. All rules are given in §C.5. The judgments for signature typing $\Gamma \vdash^{\text{src}} S : \checkmark$ and declarations

typing $\Gamma \vdash_A^{\text{src}} D : \checkmark$ ensure that no shadowing occurs and no free variables are used. We omit the rules here. The judgment for bindings $\Gamma \vdash^{\text{src}} B : D$ uses a core-language typing judgment for expressions and typing of signatures, and is mutually recursive with the typing of modules (for submodules). The rules are straightforward and omitted.

Module typing The judgment for modules contains both syntax-directed and free-floating rules, making the presentation purposefully logical rather than algorithmic. There is an equivalent algorithmic presentation, but it is more involved and hides the key insights of the separation of operations between resolution, equivalence, strengthening, subtyping, and typing. Besides, the equivalent, canonical model that we propose in the next section is better suited for deriving algorithmic presentations for the source system. Free-floating typing rules must have no computational content. Therefore, general subtyping (which has some computation content), is not free floating: it should only be used at specific program points, namely functor application and signature ascription, where the target signature of the runtime coercion is explicit in the source. While having no computational content, subtyping by abstraction is still not made free-floating to prevent unnecessary abstraction of types (which would lead to loss of type-sharing). The syntax-directed rules are as follows:

$$\begin{array}{c}
\text{S-TYP-MOD-RES} \quad \text{S-TYP-MOD-SEALING} \quad \text{S-TYP-MOD-STRUCT} \\
\frac{\Gamma \vdash^{\text{src}} P \triangleright S}{\Gamma \vdash^{\text{src}} P : S} \quad \frac{\Gamma \vdash^{\text{src}} S : \checkmark} \quad \frac{\Gamma \vdash^{\text{src}} P : S' \quad \Gamma \vdash^{\text{src}} S' <: S}{\Gamma \vdash^{\text{src}} (P : S) : S} \quad \frac{\Gamma \vdash_A^{\text{src}} \overline{B} : \overline{D} \quad A \notin \Gamma}{\Gamma \vdash^{\text{src}} \text{struct}_A \overline{B} \text{ end} : \text{sig}_A \overline{D} \text{ end}} \\
\\
\text{S-TYP-MOD-FUNCTOR} \quad \text{S-TYP-MOD-APP} \\
\frac{\Gamma \vdash^{\text{src}} S_a : \checkmark \quad \Gamma; (Y : S_a) \vdash^{\text{src}} M : S \quad Y \notin \Gamma}{\Gamma \vdash^{\text{src}} (Y : S_a) \rightarrow M : (Y : S_a) \rightarrow S} \quad \frac{\Gamma \vdash^{\text{src}} P : (Y : S_a) \rightarrow S \quad \Gamma \vdash^{\text{src}} P' : S'_a \quad \Gamma \vdash^{\text{src}} S'_a <: S_a}{\Gamma \vdash^{\text{src}} P(P') : S[Y \mapsto P']} \\
\\
\text{S-TYP-MOD-PROJ} \\
\frac{\Gamma \vdash^{\text{src}} M : \text{sig}_A (\overline{D}_1, \text{module } X : S, \overline{D}_2) \text{ end} \quad \Gamma, \overline{D}_1 \vdash^{\text{src}} S <: S' \quad \Gamma \vdash^{\text{src}} S' : \checkmark}{\Gamma \vdash^{\text{src}} M.X : S'}
\end{array}$$

Functor application (Rule **S-TYP-MOD-APP**) is normally a place where *signature-avoidance* may occur. In our presentation however, the restriction of functor application to paths prevents this issue. Indeed, as all the abstract types of the argument are defined in the context, no type field is hidden by the application, and thus, no escaping of scope can occur. In contrast, as the grammar allows projection on any module expression (which can hide type fields and could trigger signature avoidance), the projection rule **S-TYP-MOD-PROJ** is designed to tackle the issue. The abstraction subtyping $\Gamma, \overline{D}_1 \vdash^{\text{src}} S <: S'$ allows for cutting the links to types in S that will become inaccessible *outside* of the signature of M , i.e., without the fields declared in \overline{D}_1 , so that the resulting signature S' is wellformed in Γ . This rule is however too permissive, as we allow to *implicitly abstract more than necessary* and lose some type sharing. Additional conditions to prevent *unnecessary* loss of type sharing could be added to this rule, but they would be complex to write down in this setting. Fortunately, they will be easy to express using the canonical system.

Free-Floating rules Before doing a projection $M.X$, we must allow to *rewire* the type sharing tree so that types used in X point to other aliases (either external ones, i.e., accessible outside

of M , or local ones, accessible inside of X). Similarly, we must allow strengthening to keep sharing information between aliases. This leads to two additional *free-floating* rules:

$$\frac{\text{S-TYP-STRENGTHEN} \quad \Gamma \overset{\text{src}}{\vdash} P : S \quad \Gamma \vdash S/P \gg S'}{\Gamma \overset{\text{src}}{\vdash} P : S'} \quad \frac{\text{S-TYP-EQUIV} \quad \Gamma \overset{\text{src}}{\vdash} M : S \quad \Gamma \overset{\text{src}}{\vdash} S \approx S'}{\Gamma \overset{\text{src}}{\vdash} M : S'}$$

Remarks on the source system While this declarative presentation of the source system may appear relatively simple, it actually suffers from three main issues. First, it relies on the equivalence judgment to *guess* correct rewritings in signatures. Second, type sharing can be lost when projecting on a submodule via over-abstraction. Thirdly, the strengthening judgment may seem ad-hoc, while being necessary to prevent loss of type-sharing. In current OCaml, these problems are partially solved via *heuristics* which are sufficient in simple cases, but fail for more advanced uses of modules. We now study a canonical presentation of the module system that solves all three issues.

3 Canonical signatures and existential types

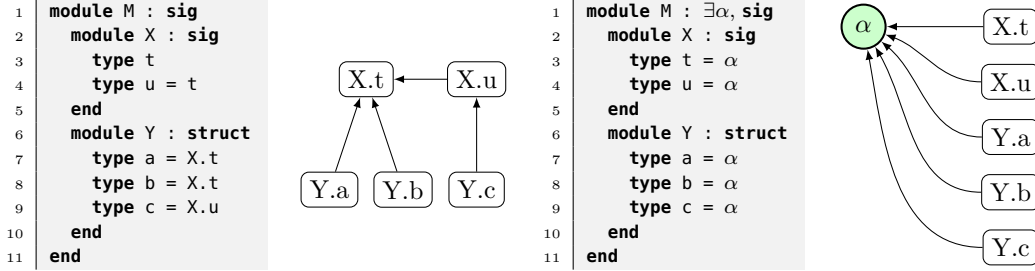
Some issues of the source presentation come from the ambiguity and weaknesses of the path-based type-sharing. In this section, we present an alternative set of typing rules with an extended syntax for signatures inspired by previous work on representation of modules in F^ω . This system, called *canonical*, has simpler typing judgments and does not suffer from the signature avoidance problem. We show that the canonical system is more expressive than the source one, as every module expression that can be typed in the source system can also be typed in the canonical system with a signature that has at least the same amount of type sharing.

3.1 The need for existential types

In the source presentation, maintaining the connected components of the type sharing tree throughout projection —where fields can be deleted— sometimes require rewriting type equalities (through equivalence) or abstracting type fields (through abstraction subtyping). The core issue is that source signatures store type equalities via a type sharing tree that is cumbersome to handle. However, the information we are actually interested in is not the type sharing tree itself, only the connected components. To handle those directly, we introduce an *existential type* that acts as a canonical representative of the equivalence class. In the type sharing tree, it can be seen as an additional point with a special status, and the whole tree is flattened to one level of depth, as illustrated in [Example 4](#). Rooted with an existential type that cannot be deleted, the connected components become insensible to the loss of some type fields. Building on this intuition, we present a new system, called *canonical*, where signatures use quantifiers to handle abstract types.

3.2 System structure and judgments

The grammar of the language, given in [Figure 5](#), extends the OCAML grammar of [Figure 3](#). Crucially, the source syntax (for modules and signatures) remains the same, including source signatures, while new syntactical categories are introduced to represent canonical signatures (and canonical types). By convention, we use curvy capitals (\mathcal{R} , \mathcal{S} , \mathcal{D} , ...) for canonical objects.



Example 4: Comparison between the source and canonical signatures and their associated type-sharing trees. Canonical signatures are extended with an existential binder that acts as a special anchor point for the tree.

<p>Canonical Types</p> $\tau ::= \alpha \quad (\text{Existential identifier})$ $ \dots \quad (\text{Other types})$ <p>Environments</p> $\Gamma ::= \emptyset \quad (\text{Empty})$ $ \Gamma, \bar{\alpha} \quad (\text{Abstract types})$ $ \Gamma, (Y : \mathcal{R}) \quad (\text{Functor Argument})$ $ \Gamma, (A.I : \mathcal{D}) \quad (\text{Declaration})$	<p>Canonical abstract signatures</p> $\mathcal{S} ::= \exists \bar{\alpha}. \mathcal{R} \quad (\text{Abstract signature})$ <p>Canonical manifest signatures</p> $\mathcal{R} ::= \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S} \quad (\text{Functor})$ $ \text{sig}_A \bar{\mathcal{D}} \text{ end} \quad (\text{Signature})$ <p>Canonical declarations</p> $\mathcal{D} ::= \text{val } x : \tau \quad (\text{Values})$ $ \text{type } t = \tau \quad (\text{Types})$ $ \text{module } X : \mathcal{R} \quad (\text{Modules})$ $ \text{module type } T = \lambda \bar{\alpha}. \mathcal{R} \quad (\text{Module types})$
---	---

Figure 5: Syntax extensions of the *canonical* module language.

We distinguish between *manifest* canonical signatures \mathcal{R} that only refer to abstract types bound in the context and *abstract* canonical signatures $\exists \bar{\alpha}. \mathcal{R}$ (written \mathcal{S}) that also specify the existential types $\bar{\alpha}$ created by the module. Notice that canonical manifest signatures cannot refer to other signatures (e.g., using paths), as these are always inlined. Signatures of functors are polymorphic in the abstract types provided by the argument: indeed, these should be treated abstractly, while they can also be returned and thus shared in the result. Signatures stored in module type declarations are just parameterized by their abstract types: these will later be existentially or universally quantified, depending on context. We use an F^ω style λ -binding for these. As we will see in the typing rules, the grammar restricts the positions where abstract types are bounded (existentially or universally). For instance, inside a structural signature, all submodules have manifest signatures, while functor bodies may bound new abstract types.

Canonical signatures remove the need for resolution, equivalence, and strengthening, which significantly simplifies the overall presentation. The judgments are thus reduced to typing of modules (and bindings), typing of source signatures, which we enrich to produce a canonical signature, and subtyping.

3.3 Signature typing

We start with signature typing $\Gamma \vdash^{\text{can}} S : \lambda \bar{\alpha}. \mathcal{R}$ and declaration typing $\Gamma \vdash_A^{\text{can}} D : \lambda \bar{\alpha}. \mathcal{D}$, as they illustrate the key mechanism of *abstract type lifting*. The positions where abstract types are

bound are crucial to ensure the correct sharing of types. First, abstract types are introduced only by an abstract type declaration (Rule **C-TYP-DECL-TYPEABS**). Then, they are gathered and lifted from abstract type definitions and submodules to their parent module to ensure sharing between declarations in the same structure (rules **C-TYP-DECL-MOD** and **C-TYP-DECL-SEQ**). The lifting does not go through a functor definition (Rule **C-TYP-SIG-FUNCTOR**): indeed, in the generative case, each application of the functor should create new unrelated abstract types. The full set of rules are in §D.2.

$$\begin{array}{c}
\text{C-TYP-DECL-TYPEABS} \\
\frac{A.t \notin \Gamma}{\Gamma \vdash_A^{\text{can}} \text{type } t = A.t : \lambda\alpha. \text{type } t = \alpha} \\
\\
\text{C-TYP-DECL-SEQ} \\
\frac{\Gamma \vdash_A^{\text{can}} D_1 : \lambda\bar{\alpha}_1. \mathcal{D}_1 \quad \Gamma, \bar{\alpha}_1, A.I_1 : D_1 \vdash_A^{\text{can}} \bar{D} : \lambda\bar{\alpha}. \bar{D}}{\Gamma \vdash_A^{\text{can}} (D_1, \bar{D}) : \lambda\bar{\alpha}_1 \bar{\alpha}. (\mathcal{D}_1, \bar{D})} \\
\\
\text{C-TYP-DECL-MOD} \\
\frac{\Gamma \vdash^{\text{can}} S : \lambda\bar{\alpha}. \mathcal{R} \quad A.X \notin \Gamma}{\Gamma \vdash_A^{\text{can}} (\text{module } X : S) : \lambda\bar{\alpha}. (\text{module } X : \mathcal{R})} \\
\\
\text{C-TYP-SIG-FUNCTOR} \\
\frac{\Gamma \vdash^{\text{can}} S_a : \lambda\bar{\alpha}. \mathcal{R}_a \quad \Gamma, \bar{\alpha}, Y : \mathcal{R}_a \vdash^{\text{can}} S : \lambda\bar{\beta}. \mathcal{R} \quad Y \notin \Gamma}{\Gamma \vdash^{\text{can}} (Y : S_a) \rightarrow S : \forall\bar{\alpha}. (Y : \mathcal{R}_a) \rightarrow \exists\bar{\beta}. \mathcal{R}}
\end{array}$$

3.4 Subtyping

The subtyping between two canonical signatures $\exists\bar{\alpha}. \mathcal{R}$ and $\exists\bar{\alpha}'. \mathcal{R}'$ is similar to the source subtyping, with the addition of the mechanism to deal with quantifiers.

Rule **C-SUB-SIG-MATCH** allows us to compare abstract signatures, while the other subtyping rules deal with concrete signatures. The types $\bar{\alpha}$ created by the left-hand-side signature are made available (i.e., pushed in the context), while the right-hand-side ones $\bar{\alpha}'$ are instantiated with some types $\bar{\tau}$. If both signatures define the same abstract types (up to α -conversion), a simple renaming instantiation $\bar{\alpha} \mapsto \bar{\alpha}'$ is sufficient. Otherwise, the left-hand-side signature \mathcal{R} can be *less abstract* (i.e., binding fewer abstract types) than the right-hand-side one \mathcal{R}' , in which case some right-hand side abstract types are instantiated with concrete types.

For functor signatures (Rule **C-SUB-SIG-FUNCTOR**), the parameter is in a contravariant position, hence we check subtyping of the argument signatures in reverse order. As with the source presentation, we distinguish between two kinds of subtyping: abstraction-only subtyping $\Gamma \vdash^{\text{can}} \mathcal{S}_1 \triangleleft: \mathcal{S}_2$ and general subtyping $\Gamma \vdash^{\text{can}} \mathcal{S}_1 <: \mathcal{S}_2$, which allows both abstraction and deletion of fields. They differ only by the rules **C-SUB-SIG-SIG** vs. **C-SUBEQ-SIG-SIG**. In §D.1 and in the following, the rules that are similar between abstraction and general subtyping are given with the symbol $\triangleleft:$ being either $<:$ or $\triangleleft:$.

$$\begin{array}{c}
\text{C-SUB-SIG-MATCH} \\
\frac{\Gamma, \bar{\alpha} \vdash^{\text{can}} \mathcal{R} \triangleleft: \mathcal{R}' [\bar{\alpha}' \mapsto \bar{\tau}]}{\Gamma \vdash^{\text{can}} \exists\bar{\alpha}. \mathcal{R} \triangleleft: \exists\bar{\alpha}'. \mathcal{R}'} \\
\\
\text{C-SUB-SIG-FUNCTOR} \\
\frac{\Gamma, \bar{\alpha}' \vdash^{\text{can}} \mathcal{R}' \triangleleft: \mathcal{R} [\bar{\alpha} \mapsto \bar{\tau}] \quad \Gamma, \bar{\alpha}', (Y : \mathcal{R}') \vdash^{\text{can}} \mathcal{S} [\bar{\alpha} \mapsto \bar{\tau}] \triangleleft: \mathcal{S}' \quad Y \notin \Gamma}{\Gamma \vdash^{\text{can}} \forall\bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S} \triangleleft: \forall\bar{\alpha}'. (Y : \mathcal{R}') \rightarrow \mathcal{S}'} \\
\\
\text{C-SUB-SIG-SIG} \\
\frac{\bar{\mathcal{D}}_0 \subseteq \bar{\mathcal{D}} \quad \Gamma \vdash^{\text{can}} \bar{\mathcal{D}}_0 \triangleleft: \bar{\mathcal{D}}}{\Gamma \vdash^{\text{can}} \text{sig}_A \bar{\mathcal{D}} \text{ end} \triangleleft: \text{sig}_A \bar{\mathcal{D}}' \text{ end}} \\
\\
\text{C-SUBEQ-SIG-SIG} \\
\frac{\Gamma \vdash^{\text{can}} \bar{\mathcal{D}} \triangleleft: \bar{\mathcal{D}}'}{\Gamma \vdash^{\text{can}} \text{sig}_A \bar{\mathcal{D}} \text{ end} \triangleleft: \text{sig}_A \bar{\mathcal{D}}' \text{ end}}
\end{array}$$

3.5 Typing

As with subtyping, the use of canonical signatures and existential types makes the typing simpler and syntax directed. The key technical point is the lifting of the existential quantification that

happens during typing, as already explained for the signature typing in §3.3.

Bindings typing The rules presented below for typing a binder give a list of canonical declarations *alongside* a list of (created) existential types. The signature typing judgment is used to convert source types and signatures (as in Rule **C-TYP-DECL-MODTYPE** for instance) into canonical types and signatures.

$$\begin{array}{c}
\text{C-TYP-DECL-MOD} \\
\frac{\Gamma \vdash^{\text{can}} M : \exists \bar{\alpha}. \mathcal{R} \quad A.X \notin \Gamma}{\Gamma \vdash_A^{\text{can}} (\text{module } X = M) : \exists \bar{\alpha}. (\text{module } X : \mathcal{R})}
\end{array}
\qquad
\begin{array}{c}
\text{C-TYP-DECL-MODTYPE} \\
\frac{\Gamma \vdash^{\text{can}} S : \lambda \bar{\alpha}. \mathcal{R} \quad A.T \notin \Gamma}{\Gamma \vdash_A^{\text{can}} (\text{module type } T = S) : (\text{module type } T = \lambda \bar{\alpha}. \mathcal{R})}
\end{array}$$

$$\begin{array}{c}
\text{C-TYP-DECL-SEQ} \\
\frac{\Gamma \vdash_A^{\text{can}} B_1 : \exists \bar{\alpha}_1. \mathcal{D}_1 \quad \Gamma, \bar{\alpha}_1, A.I_1 : \mathcal{D}_1 \vdash_A^{\text{can}} \bar{B} : \exists \bar{\alpha}. \bar{\mathcal{D}}}{\Gamma \vdash_A^{\text{can}} B_1, \bar{B} : \exists \bar{\alpha}_1 \bar{\alpha}. \mathcal{D}_1, \bar{\mathcal{D}}}
\end{array}$$

The existential types—which are lifted from submodules with Rule **C-TYP-DECL-MOD**—are merged by Rule **C-TYP-DECL-SEQ**, to extend their scope to the local enclosing module. For example, if a module M has a submodule X which defines an abstract type α , the existential quantification is lifted from the submodule X to the whole module M .

Module expressions typing Existential types are introduced only by explicit sealing (Rule **C-TYP-MOD-SEALING**). Applying a functor to its argument (Rule **C-TYP-MOD-APP**) can share the types of the argument via the substitution $[\bar{\alpha} \mapsto \bar{\tau}]$.

$$\begin{array}{c}
\text{C-TYP-STRUCT} \\
\frac{\Gamma \vdash_A^{\text{can}} B : \exists \bar{\alpha}. \bar{\mathcal{D}} \quad A \notin \Gamma}{\Gamma \vdash^{\text{can}} \text{struct}_A B \text{ end} : \exists \bar{\alpha}. \text{sig}_A \bar{\mathcal{D}} \text{ end}}
\end{array}
\qquad
\begin{array}{c}
\text{C-TYP-MOD-SEALING} \\
\frac{\Gamma \vdash^{\text{can}} S : \lambda \bar{\alpha}. \mathcal{R} \quad \Gamma \vdash^{\text{can}} P : \mathcal{R}' \quad \Gamma \vdash^{\text{can}} \mathcal{R}' <: \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}]}{\Gamma \vdash^{\text{can}} (P : S) : \exists \bar{\alpha}. \mathcal{R}}
\end{array}$$

$$\begin{array}{c}
\text{C-TYP-MOD-FUNCTOR} \\
\frac{\Gamma \vdash^{\text{can}} S : \lambda \bar{\alpha}. \mathcal{R} \quad \Gamma, \bar{\alpha}, (Y : \mathcal{R}) \vdash^{\text{can}} M : \mathcal{S}}{\Gamma \vdash^{\text{can}} (Y : S) \rightarrow M : \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S}}
\end{array}
\qquad
\begin{array}{c}
\text{C-TYP-MOD-APP} \\
\frac{\Gamma \vdash^{\text{can}} P : \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S} \quad \Gamma \vdash^{\text{can}} P' : \mathcal{R}' \quad \Gamma \vdash^{\text{can}} \mathcal{R}' <: \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}]}{\Gamma \vdash^{\text{can}} P(P') : \mathcal{S}[\bar{\alpha} \mapsto \bar{\tau}]}
\end{array}$$

$$\begin{array}{c}
\text{C-TYP-MOD-PROJ} \\
\frac{\Gamma \vdash^{\text{can}} M : \exists \bar{\alpha}. \text{sig}_A \bar{\mathcal{D}}_1, \text{module } X : \mathcal{R}, \bar{\mathcal{D}}_2 \text{ end}}{\Gamma \vdash^{\text{can}} M.X : \exists \bar{\alpha}. \mathcal{R}}
\end{array}$$

In the source presentation, the projection rule **S-TYP-MOD-PROJ** is the only place where signature avoidance can arise, by cutting local links in unsolvable ways. We needed to allow for abstraction subtyping. In the canonical presentation (Rule **C-TYP-MOD-PROJ**), all the local links go through existentially quantified abstract types in front of the signature. By keeping all the existentially quantified types, projecting a component cannot cause signature avoidance: all the abstract types (that could go out of scope and cause signature avoidance issues) that $M.X$ could use are in the list $\bar{\alpha}$ and remain bound in the signature of $M.X$. The full set of typing rules is given in §D.2.

3.6 From the source to the canonical system

Both presentations share the same grammar for module expressions and signatures, and thus have the same input for typing. By extending the typing predicate to environments (to check wellformedness), we can translate source judgments (resolution, equivalence, strengthening) into the canonical presentation. This translation—called canonification—gives insight on how the source mechanisms can be understood in the canonical framework.

Typing of source environments We translate environments with source signatures into their canonical counterparts, using signature typing, declaration typing, and the following rules:

$$\begin{array}{c}
\text{C-CF-EMPTY} \\
\frac{}{\emptyset \vdash^{\text{can}} \emptyset}
\end{array}
\qquad
\begin{array}{c}
\text{C-CF-FCTARG} \\
\frac{\Gamma \vdash^{\text{can}} \Gamma_s \quad \Gamma \vdash^{\text{can}} S : \lambda\bar{\alpha}.\mathcal{R}}{\Gamma, \bar{\alpha}, (Y : \mathcal{R}) \vdash^{\text{can}} \Gamma_s, (Y : S)}
\end{array}
\qquad
\begin{array}{c}
\text{C-CF-DECL} \\
\frac{\Gamma \vdash^{\text{can}} \Gamma_s \quad \Gamma \vdash_A^{\text{can}} D : \lambda\bar{\alpha}.\mathcal{D}}{\Gamma, \bar{\alpha}, (A.I : \mathcal{D}) \vdash^{\text{can}} \Gamma_s, (A.I : \mathcal{D})}
\end{array}$$

Canonification of resolution We now state how the resolution behaves regarding canonification. The main difference between the two systems comes from the fact that abstract types are quantified outside of the signature in the canonical system (the signature is always concrete), while they are introduced anywhere inside source signatures (which may contain abstract type definitions). Thus, the source signature obtained via resolution of P and the canonical signature of P only differ by the presence of quantifiers.

$$\Gamma_s^{\text{src}} \vdash P \triangleright S \wedge \Gamma \vdash^{\text{can}} \Gamma_s \wedge \Gamma \vdash^{\text{can}} S : \lambda\bar{\alpha}.\mathcal{R} \implies \Gamma \vdash^{\text{can}} P : \mathcal{R} \quad (1)$$

$$\Gamma_s^{\text{src}} \vdash Q.T \triangleleft S \wedge \Gamma \vdash^{\text{can}} \Gamma_s \wedge \Gamma \vdash^{\text{can}} S : \lambda\bar{\alpha}.\mathcal{R} \implies \Gamma \vdash^{\text{can}} Q.T : \lambda\bar{\alpha}.\mathcal{R} \quad (2)$$

Canonification of strengthening Via resolution, the introduction of abstract types is re-exposed by the signature. To prevent an actual duplication of abstract types, the source presentation relies on strengthening. While the resolution of a path P corresponds to a canonical signature with existential types bound in front, a strengthened signature can be made fully concrete (no quantified types): all types refer to their original definition through the path P . In the canonical presentation, this is a *no-op*, as all type definitions already go through the proper quantified existential types in the context:

$$\Gamma_s^{\text{src}} \vdash P \triangleright S \wedge \Gamma_s \vdash S/P \gg S' \wedge \Gamma \vdash^{\text{can}} \Gamma_s \wedge \Gamma \vdash^{\text{can}} S : \lambda\bar{\alpha}.\mathcal{R} \implies \Gamma \vdash^{\text{can}} S' : \mathcal{R} \quad (3)$$

Canonification of equivalence As we described in the previous section, two equivalent signatures have the same connected components of their type sharing tree. In the canonical model, all connected components are flattened into a tree of depth 1, with an existentially quantified variable at the root. This means that two equivalent signatures actually have the same canonification, which matches the intuition (and naming) that canonical signatures are indeed *canonical*. We get the following result:

$$\Gamma_s^{\text{src}} \vdash S_1 \approx S_2 \wedge \Gamma \vdash^{\text{can}} \Gamma_s \wedge \Gamma \vdash^{\text{can}} S_1 : \mathcal{S}_1 \wedge \Gamma \vdash^{\text{can}} S_2 : \mathcal{S}_2 \implies \mathcal{S}_1 = \mathcal{S}_2 \quad (4)$$

Canonification of subtyping As the rules of subtyping in the source and canonical systems are very similar, the two judgments are also very similar. The only difference is that source subtyping can lose type equalities by over-abstractation. However, in the canonical system, with

Rule **C-SUB-SIG-MATCH**, the correspondence between the abstract types is dealt with at the signature level, with no abstraction done at the declaration level. We get the following result:

$$\Gamma_s \vdash^{\text{src}} S_1 <: S_2 \wedge \Gamma \vdash^{\text{can}} \Gamma_s \wedge \Gamma \vdash^{\text{can}} S_1 : S_1 \wedge \Gamma \vdash^{\text{can}} S_2 : S_2 \implies \Gamma \vdash^{\text{can}} S_1 <: S_2 \quad (5)$$

Canonification of typing Once all canonification results have been laid out, the typing result becomes easy to write. All typing derivations can be translated into canonical ones, with the only difference being that some over-abstraction during projections and resolutions can lead to a source signature having lost some type sharing. This type-sharing loss, can be delayed to the last step of the derivation (combining the losses of all the intermediary signatures), which gives the following result:

Theorem 1 (Canonification of typing). Considering any module expression M that admits a signature S in the source presentation, it also admits a corresponding canonical signature S with more type equalities:

$$\Gamma_s \vdash^{\text{src}} M : S \wedge \Gamma \vdash^{\text{can}} \Gamma_s \wedge \Gamma \vdash^{\text{can}} S : S \implies \Gamma \vdash^{\text{can}} M : S' \wedge \Gamma \vdash^{\text{can}} S' <: S$$

This link between the source and canonical system shows the expressiveness of the latter one, but is only half of the story. Exploring at which conditions the canonical derivations can be translated back into the source will uncover a new concept: anchorability.

4 Anchoring: back to the source system

In this section we study the process of translating typing derivations from the canonical system back into the source system, which we call *anchoring*. As the canonical system is more expressive than the source one, it is not always possible to translate signatures back into the source syntax. Still, we can precisely exhibit the conditions under which anchoring is possible. This gives a new specification of the source system, which could also be used to improve the way signature avoidance is actually resolved in OCAML.

The canonical system thus provides the intuition, formal guarantees, and algorithmic insights for the design of OCAML's modules mechanisms.

4.1 Quantification vs structural information

The key insight is the semantic difference in the source syntax between the declaration of a *concrete* type (`type t = τ`) and that of an *abstract* type (`type t`). An abstract type declaration states the *introduction* of both a *new type* and of a *new field* that may later be used as a handle to refer to this abstract type, directly or indirectly via a path. For a signature used in a covariant position, an abstract type declaration effectively *creates* a new type (existential quantification) and *adds* a type field (structural information) to the signature. By contrast, a concrete type definition only introduces structural information—adding a field to refer to an existing type but without introducing a new type.

Canonical signatures *separate* the quantification information (existential, universal, or lambda binders) from the structural information (fields). Thus, it makes the scope of abstract types explicit, using the logical notion of binder, which is perfectly suited for that purpose. More importantly, it allows to refer to types that do not yet have a handle, while the source system cannot. This is illustrated in the following code :

```

1  module X = (struct
2      module X0 = (... : sig type t end)
3      module X1 = struct
4          type u = X0.t * int
5          type v = X0.t * bool
6      end
7  end).X1

```

```

10 (* Canonical signature of X *)
11 module X : ∃α. sig
12     type u = α * int
13     type v = α * string
14 end

```

Here, a type definition (t) is lost when projecting only on the submodule `X1`. The source syntax *cannot* express the existence of a type that is the first component of both `u` and `v` without already having a handle to that type¹: being forced to merge quantification and structural information limits the expressiveness. However, the module `X` can be given a canonical signature, where the two kinds of information are clearly separated.

This allows us to better describe the solvable and unsolvable cases of signature avoidance for the source presentation. In the *solvable* cases of signature avoidance, each use of an abstract type has an *in-scope* alias (that can serve as a handle) defined before the type is used. In such cases, the quantification and structural information coincide in a way expressible by the source syntax. In other cases, signature avoidance is unsolvable.

4.2 Anchoring of canonical signatures

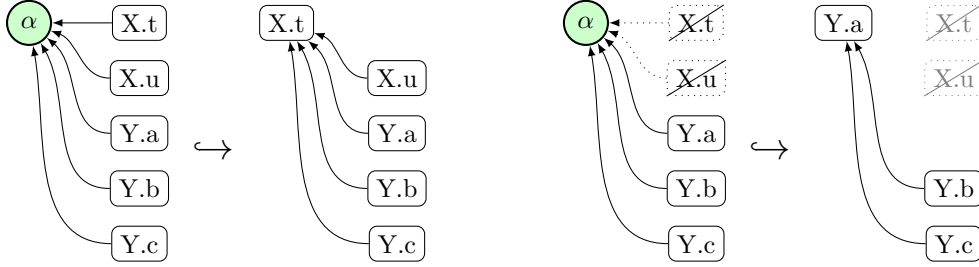
Building on the insights of the previous subsection, we can describe the *anchoring* process that translates a canonical signature back into the source system when the quantification and structural information coincide. The main mechanism boils down to identifying the first place where an abstract type α is used. If it is at the top level of a type declaration `type t = α` , we say that the existential type α is *anchorable*. In the corresponding source signature, we can make `t` an abstract type definition, creating a handle to `t` that can then be used instead of α in the canonical signature. The type `t` must be the *first* reference to the abstract type α , regardless of whether or not it is the original definition point. If the type α is used earlier, we raise a signature avoidance failure.

Example 5 illustrates the process of finding the first available position to rewire the type-sharing tree of **Example 1**. We see the change of the defining instance of α , from `X.t` to `Y.a` after a projection that makes `X.t` and `X.u` unreachable.

The anchoring process consists of associating abstract types with their first available aliases, which we gather using an *anchoring map* $\theta : \alpha \mapsto Q.t$. We define a signature anchoring judgment $\Gamma \vdash^{\text{anc}} \mathcal{S} \leftrightarrow \Gamma_{\mathcal{S}}; \theta_{\Gamma} \vdash S : \theta$ that, given an environment Γ and a canonical signature \mathcal{S} , produces a source environment $\Gamma_{\mathcal{S}}$ associated with an anchoring map θ_{Γ} and a source signature S associated with an anchoring map θ . In a nutshell, the judgment produces the source counterpart of the canonical signature \mathcal{S} by gathering the handles of the abstract types, and replacing the uses of abstract types by their handles. The full set of rules is given in **§D.3**.

Using the anchoring judgment, we define *anchorable typing*, written $\Gamma \vdash^{\text{anc}} M : \mathcal{S}$. The judgment is defined by a copy of the typing rules for the judgment $\Gamma \vdash^{\text{can}} M : \mathcal{S}$ presented in **§3.5**, except for the projection rule which takes an additional premise checking for anchorability of the resulting signature. All anchorable typing derivations can be translated into source typing derivations.

¹Adding a type field for `t` would not work, as it would not be an equivalent signature.



Canonical signature \mathcal{S}_1 Source signature S_1 Canonical signature \mathcal{S}_2 Source signature S_2

Example 5: Graphical representation of the anchoring process on the type sharing trees. In the two canonical signatures \mathcal{S}_1 and \mathcal{S}_2 , the existential type α is anchorable. For \mathcal{S}_1 , we can use $X.t$ for the handle to α , as done in S_1 . For \mathcal{S}_2 , a projection removed $X.t$ and $X.u$. The handle becomes $Y.a$ as shown in S_2 .

Theorem 2 (Anchorable typing). Anchorable typing produces a signature that is valid regarding the source typing rules:

$$\Gamma \vdash^{\text{anc}} M : \mathcal{S} \quad \wedge \quad \Gamma \vdash^{\text{anc}} \mathcal{S} \hookrightarrow \Gamma_s; \theta \vdash S : \theta \quad \Longrightarrow \quad \Gamma_s \vdash^{\text{src}} M : S \quad (6)$$

More explanations and the formal definition of the anchoring process are given in §A. This completes the link between the source and canonical systems: canonical derivations can be translated back into the source when we restrict their expressiveness. The last link, namely the actual encoding in F^ω that inspired the design of the canonical system, is detailed in §B.

Conclusion

ML-Module systems are known for being a well-studied but complex topic. The path-based, OCAML approach has proven to be successful in practice, but has some structural issues. While being restricted to a generative subset, our study of the signature avoidance problem in the source presentation exposes the limitations of the current signature syntax. These limitations are at the heart of the need for *ad-hoc* and complex fixes (strengthening, equivalence). Building on previous works, we introduced the canonical system as a more expressive yet simpler language, equipped with the right construction (existential types) to separate quantification and structural information, and solve the main issues of the source system. Using the canonical system, we shined a new light on the ad-hoc techniques of the source presentation and provided a detailed description of the solvable and unsolvable cases of signature avoidance. While still being close to the OCAML source, the canonical presentation can easily be elaborated into F^ω , which provides formal guarantees. As a middle-point between usability and formalism, the canonical system is both a comprehensive description and a framework for building new features and improving the algorithms (specifically for the *solvable* cases of signature avoidance) of the current OCAML typechecker.

References

- K. Crary. A focused solution to the avoidance problem. *Journal of Functional Programming*, 30:e24, 2020. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796820000222. URL https://www.cambridge.org/core/product/identifier/S0956796820000222/type/journal_article.

- D. DREYER. Recursive type generativity. *Journal of Functional Programming*, 17(4-5): 433–471, 2007. doi: 10.1017/S0956796807006429.
- R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 123–137, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916360. doi: 10.1145/174675.176927. URL <https://doi.org/10.1145/174675.176927>.
- R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 341–354, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913434. doi: 10.1145/96709.96744. URL <https://doi.org/10.1145/96709.96744>.
- X. Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 109–122, New York, NY, USA, 1994. Association for Computing Machinery. ISBN 0897916360. doi: 10.1145/174675.176926. URL <https://doi.org/10.1145/174675.176926>.
- X. Leroy. Applicative functors and fully transparent higher-order modules. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95*, pages 142–153, San Francisco, California, United States, 1995. ACM Press. ISBN 978-0-89791-692-9. doi: 10.1145/199448.199476. URL <http://portal.acm.org/citation.cfm?doid=199448.199476>.
- X. Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000. URL <http://journals.cambridge.org/action/displayAbstract?aid=54525>.
- D. B. MacQueen. *Using Dependent Types to Express Modular Structure*, page 277–286. Association for Computing Machinery, New York, NY, USA, 1986. ISBN 9781450373470. URL <https://doi.org/10.1145/512644.512670>.
- A. Madhavapeddy, R. Mortier, C. Rotsos, D. J. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. In V. Sarkar and R. Bodík, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*, pages 461–472. ACM, 2013. doi: 10.1145/2451116.2451167. URL <https://doi.org/10.1145/2451116.2451167>.
- J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, page 37–51, New York, NY, USA, 1985. Association for Computing Machinery. ISBN 0897911474. doi: 10.1145/318593.318606. URL <https://doi.org/10.1145/318593.318606>.
- B. Montagu and D. Rémy. Modeling Abstract Types in Modules with Open Existential Types. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL '09)*, pages 354–365, Savannah, GA, USA, Jan. 2009. ISBN 978-1-60558-379-2. doi: <http://doi.acm.org/10.1145/1480881.1480926>.
- G. Radanne, T. Gazagnaire, A. Madhavapeddy, J. Yallop, R. Mortier, H. Mehnert, M. Perston, and D. Scott. Programming unikernels in the large via functor driven development, 2019.

- A. Rossberg. 1ML - Core and modules united. *J. Funct. Program.*, 28:e22, 2018. doi: 10.1017/S0956796818000205. URL <https://doi.org/10.1017/S0956796818000205>.
- A. Rossberg, C. Russo, and D. Dreyer. F-ing modules. *Journal of Functional Programming*, 24(5):529–607, Sept. 2014. ISSN 0956-7968, 1469-7653. doi: 10.1017/S0956796814000264. URL https://www.cambridge.org/core/product/identifier/S0956796814000264/type/journal_article.
- C. V. Russo. Types for modules. *Electronic Notes in Theoretical Computer Science*, 60:3–421, 2004. ISSN 1571-0661. doi: [https://doi.org/10.1016/S1571-0661\(05\)82621-0](https://doi.org/10.1016/S1571-0661(05)82621-0). URL <https://www.sciencedirect.com/science/article/pii/S1571066105826210>.
- L. White, F. Bour, and J. Yallop. Modular implicits. In *Proceedings ML Family/OCaml Users and Developers workshops, ML/OCaml 2014, Gothenburg, Sweden, September 4-5, 2014.*, pages 22–63, 2014. doi: 10.4204/EPTCS.198.2. URL <https://doi.org/10.4204/EPTCS.198.2>.

A Anchoring

A.1 Anchoring of signatures

In this section we present in more details the mechanism that, under certain conditions, allows to build one of the source signatures corresponding to a given canonical signature. As mentioned in §4.2, the anchoring judgment, written

$$\Gamma \vdash^{\text{anc}} \mathcal{S} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S : \theta$$

states that, given a canonical signature \mathcal{S} and environment Γ , we can build a corresponding source signature S and source environment Γ_s . Both are returned with an *anchoring map* (θ for the signature and θ_Γ for the environment) that gives a path $Q.t$ for the handle of every abstract type α .

The anchoring map is extended when a type declaration with an *un-mapped* existential is encountered, which can be seen in rule **C-ACH-DECL-ANCHORPOINT**: the anchoring of the type declaration returns a non-empty map $\alpha \mapsto t$. In contrast, if the type field is concrete, an empty anchoring map is produced, as in Rule **C-ACH-DECL-TYPE**. Maps are merged together with the sequence rule **C-ACH-DECL-SEQ**, via the merging operator \uplus . The map associated with the left-hand-side declaration is *prefixed* with the self-reference. More generally, we write $Q.\theta$ as a shortcut for the map $\alpha \mapsto Q.\theta(\alpha)$.

$$\begin{array}{c} \text{C-ACH-DECL-ANCHORPOINT} \\ \frac{\Gamma \hookrightarrow \Gamma_s : \theta_\Gamma \quad \alpha \notin \text{dom}(\theta_\Gamma)}{\Gamma \vdash_A^{\text{anc}} \text{type } t = \alpha \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \text{type } t = A.t : (\alpha \mapsto t)} \end{array} \qquad \begin{array}{c} \text{C-ACH-DECL-TYPE} \\ \frac{\Gamma \vdash^{\text{anc}} \tau \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \tau' : \emptyset}{\Gamma \vdash_A^{\text{anc}} \text{type } t = \tau \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \text{type } t = \tau' : \emptyset} \end{array}$$

$$\begin{array}{c} \text{C-ACH-DECL-SEQ} \\ \frac{\Gamma \vdash^{\text{anc}} \mathcal{D} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash D : \theta \quad \Gamma \vdash^{\text{anc}} \overline{\mathcal{D}} \hookrightarrow (\Gamma_s, A.I : D); (\theta_\Gamma \uplus A.\theta) \vdash \overline{D} : \theta'}{\Gamma \vdash_A^{\text{anc}} \mathcal{D}, \overline{\mathcal{D}} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash D, \overline{D} : \theta_\Gamma \uplus \theta'} \end{array}$$

The judgment is easily extended to signatures and environments. Rule **C-ACH-ENV-DECL** shows how the anchoring map associated with a single declaration is merged with the environment map.

$$\begin{array}{c} \text{C-ACH-ENV-DECL} \\ \frac{\Gamma \hookrightarrow \Gamma_s : \theta_\Gamma \quad \Gamma, \overline{\alpha} \vdash_A^{\text{anc}} \mathcal{D} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash D : \theta \quad \text{dom}(\theta) = \overline{\alpha}}{\Gamma, \overline{\alpha}, A.I : \mathcal{D} \hookrightarrow (\Gamma_s, A.I : D) : \theta_\Gamma \uplus A.\theta} \end{array}$$

A.2 Anchorable typing

As mentioned in §4.2, we can use the anchoring judgment to restrict the canonical typing to anchorable signatures. In our approach, the only place where a *non-anchorable* signature can be introduced is during projection on a submodule. Thus, we reuse all the rules of §3.5, except for the projection rule, which is modified to force the resulting signature to be anchorable:

$$\begin{array}{c} \text{C-TYPA-MOD-PROJ} \\ \frac{\Gamma \vdash^{\text{anc}} M : \exists \overline{\alpha}. \text{sig}_A \overline{\mathcal{D}}_1, \text{module } X : \mathcal{R}, \overline{\mathcal{D}}_2 \text{ end} \quad \Gamma \vdash^{\text{anc}} \exists \overline{\alpha}. \mathcal{R} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S : \theta}{\Gamma \vdash^{\text{anc}} M.X : \exists \overline{\alpha}. \mathcal{R}} \end{array}$$

As expected, the anchorable typing, as a restriction of the canonical typing to *source-only* mechanisms, respects the source typing. More formally:

Theorem 3 (Anchorable typing). Given any environment Γ , module M and (canonical) signature \mathcal{S} , the anchored signature S is a valid signature for the source typing:

$$\left. \begin{array}{l} \Gamma \vdash^{\text{anc}} M : \mathcal{S} \\ \Gamma \vdash^{\text{anc}} \mathcal{S} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S : \theta \end{array} \right\} \Longrightarrow \Gamma_s \vdash^{\text{src}} M : S \quad (7)$$

The proof of this theorem uses the following lemma :

Lemma 1 (Anchorability of subtyping). Given Γ , Γ_s , σ_Γ , two canonical signatures \mathcal{S}_1 and \mathcal{S}_2 , two source signatures S_1 and S_2 , and two substitutions σ_1 and σ_2 , we have:

$$\left. \begin{array}{l} \Gamma \vdash^{\text{anc}} \mathcal{S} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S : \theta \\ \Gamma \vdash^{\text{anc}} \mathcal{S}' \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S' : \theta' \\ \Gamma \vdash^{\text{can}} \mathcal{S} <: \mathcal{S}' \end{array} \right\} \Longrightarrow \Gamma_s \vdash^{\text{src}} S <: S' \quad (8)$$

This completes the presentation of the link between the source and the canonical systems. As mentioned, the canonical system was designed as a simpler presentation of the mechanisms found in the F^ω encoding of modules. To complete this presentation, the next section gives in more details the encoding and the main techniques used, especially regarding the sharing of abstract types.

B F^ω and elaboration

To properly finish the transition from source to F^ω via the canonical system, we are missing one final link from the canonical system to F^ω . In this section we give formal foundations and guarantees for the canonical system through *elaboration* into F^ω : the elaboration is defined by *hybrid* judgments that relate module expressions and canonical signatures to terms and types of F^ω . This gives a precise correspondence between the key mechanisms of the canonical system, in particular existential types, and standard mechanisms of F^ω . The canonical system can actually be understood as a *particular mode of use* of F^ω . Thanks to F^ω , elaborated terms enjoy the properties of soundness, progress, and type preservation. In fact, the canonical system has been designed with the elaboration in mind, mainly by erasing proof terms and only retaining F^ω types. Hence, the elaboration is mostly straightforward.

Although, we use F^ω as the target language, we do not actually need the power of F^ω to model generative functors. The reason to pursue the development in F^ω is to allow parameterized types in the source language and to be able to extend the encoding to the case of applicative functors in the future.

We start by giving the syntax and rules of the version of F^ω (extended with existential types and records) that we use for the elaboration and present the F^ω terms and types that encode modules and signatures. Then, we introduce the hybrid judgments. Finally, we state the correctness theorem for the elaboration and the link between the canonical system and F^ω .

This section is largely inspired by [Rossberg et al. \[2014\]](#). We only made minor changes to the encoding of module expressions and signatures: the encoding of module types use λ -abstraction rather than existential quantification; declarations are encoded differently. The main difference is the extension of the judgments to produce *both* the F^ω type and the canonical signature, whereas in [Rossberg et al. \[2014\]](#), typing and subtyping are only defined by elaboration.

$\kappa := \Omega \mid \kappa \rightarrow \kappa$	(kinds)
$\rho := \alpha \mid \rho \rightarrow \rho \mid \{\overline{\ell} : \rho\} \mid \forall \alpha : \kappa. \rho \mid \exists \alpha : \kappa. \rho \mid \lambda \alpha : \kappa. \rho \mid \rho \rho$	(types)
$E := x \mid \lambda x : \rho. E \mid E E \mid \{\overline{\ell} = \overline{E}\} \mid E. \ell \mid \lambda \alpha : \kappa. E \mid E \rho$ $\mid \text{pack } \langle \rho, E \rangle_\rho \mid \text{unpack } \langle \alpha, x \rangle = E \text{ in } E$	(terms)
$W := \lambda x : \rho. E \mid \{\overline{\ell} = \overline{W}\} \mid \lambda \alpha : \kappa. E \mid \text{pack } \langle \rho, W \rangle_\rho$	(values)
$\Gamma := \cdot \mid \Gamma, \alpha : \kappa \mid \Gamma, x : \rho$	(environments)

Figure 6: Syntax of F^ω **Abstract signature** $\Pi := \exists \overline{\alpha}. \Sigma$ **Concrete signature**
 $\Sigma := \forall \overline{\alpha}. \Sigma \rightarrow \Pi$ (Functor)
 $\mid \{\overline{\zeta}\}$ (Signature)
Declaration (with syntactic sugar)
 $\zeta := \text{val } x : \rho \triangleq \ell_x : \rho$ (Value)
 $\mid \text{type } t = \rho \triangleq \ell_t : \langle \rho \rangle$ (Type)
 $\mid \text{module } X : \Sigma \triangleq \ell_X : \Sigma$ (Module)
 $\mid \text{module type } T = \lambda \overline{\alpha}. \Sigma \triangleq \ell_T : \langle \lambda \overline{\alpha}. \Sigma \rangle$ (Module type)

Figure 7: Syntactic categories and encoding

B.1 F-omega and encoded signatures

We use a standard variant of explicitly typed F^ω with primitive records and existential types; We only give its syntax in [Figure 6](#); its typing rules are available in [§E](#). We use letters ρ and E to range over types and expressions so as to distinguish them from the core language types and expressions, τ and e , although these should be seen as a subset of ρ and E . While kinds are part of terms, we usually omit them in F^ω terms for the sake of conciseness and readability.

To help with the elaboration, we assume a collection ℓ_I of record labels indexed by identifiers of the canonical (and source) system. The encoding of the canonical system into F^ω is described in [Figure 7](#). Functors are encoded as functions and structural signatures as records. Declarations are encoded as record entries: we use the category of the identifier to distinguish between the encoding of values, types, modules and module types. We also introduce some syntactic sugar to have the encoding of declarations look similar to the usual canonical and source syntax. We use the following (overloaded) operator to shorten the encoding of type and module-type fields, with the type encoding on the left and the corresponding term on the right:

$$\langle \langle \rho \rangle \rangle := \forall (\beta : \kappa \rightarrow \star). \beta \rho \rightarrow \beta \rho \quad (\text{Type}) \qquad \langle \langle \rho \rangle \rangle := \Lambda (\beta : \kappa \rightarrow \star). \lambda (x : \beta \rho). x \quad (\text{Term})$$

This allows to represent *type* fields as F^ω *terms*, specifically record fields: the term is always the identity and only its type encoding in the annotation matters, namely to enforce typing constraints. Indeed, erasing type fields during elaboration would have been possible, but this would prevent the elaboration from distinguishing canonical terms that would only differ from their type fields. This will ensure that two modules that differ only from their type definitions (i.e., their type fields) will also differ in their signatures. The type is generalized with a type operator β to allow the encoding of higher-kinded types.

A representative example of the encoding of a simple module is given in [Example 6](#). The encoded signature is a type of F^ω that can be read as a canonical signature. Using the syntactic sugar makes the similarity with the source type even more striking. In the evidence term, we can see that the sealing of `int` corresponds to a `pack(int, ...)` term. To share the abstract type between the two fields (`X` and `u`), the abstract type is unpacked and then repacked, using a

pattern of the form $\text{unpack}\langle\bar{\alpha}, \cdot\rangle = \cdot$ in $\text{pack}\langle\bar{\alpha}, \cdot\rangle$ called a *repacking* pattern.

<pre> 1 module M = struct 2 module X = (struct 3 type t = int 4 let v = 42 5 end : sig 6 type t 7 val v : t 8 end) 9 type u = X.t*bool 10 end </pre>	<p>Encoded signature</p> $\Pi = \exists\alpha \left\{ \begin{array}{l} \ell_X : \left\{ \begin{array}{l} \ell_t : \langle\langle\alpha\rangle\rangle \\ \ell_v : \alpha \end{array} \right\} \\ \ell_u : \langle\langle\alpha \times \text{bool}\rangle\rangle \end{array} \right\} \triangleq \exists\alpha \left\{ \begin{array}{l} \text{module } X : \left\{ \begin{array}{l} \text{type } t = \alpha \\ \text{val } v : \alpha \end{array} \right\} \\ \text{type } u = \alpha \times \text{bool} \end{array} \right\}$ <p>Encoded module</p> $E = \text{unpack}\langle\alpha_1, y_1\rangle = \text{pack}\langle\text{int}, \{\ell_X = \{\ell_t = \langle\langle\text{int}\rangle\rangle, \ell_v = 42\}\}\rangle \text{ in} \\ \text{unpack}\langle\emptyset, y_2\rangle = \{\ell_u = \langle\langle(\alpha_1 \times \text{bool})\rangle\rangle\} \text{ in} \\ \text{pack}\langle\alpha_1, \{\ell_X = (y_1.\ell_X), \ell_u = (y_2.\ell_u)\}\rangle$
--	---

Example 6: A simple module with two components (a submodule X and a type definition u). Its signature is encoded into a type Π of F^ω . A term E (called an *evidence term*) represents the module and has type Π .

B.2 An extension of the canonical system

The *hybrid* system can be seen as an *extension* of the canonical one: the judgments are extended to actually present both the canonical and F^ω parts. We define all judgments over an extended, hybrid environment Δ that contains both canonical and F^ω terms, defined as follows:

$$\Delta ::= \emptyset \mid \Delta, \bar{\alpha} \mid \Delta, (Y : (\mathcal{R}, \Sigma)) \mid \Delta, (A.I : (\mathcal{D}, \zeta))$$

The elaborated judgments rely on a core-language translation into F^ω for terms and types, that is omitted here.

B.2.1 Subtyping

We define the subtyping judgment $\Delta \vdash^{\text{elab}} \mathcal{S}_1 <: \mathcal{S}_2 \rightsquigarrow \Pi_1 <: \Pi_2 \Rightarrow f$, which follows closely the canonical one, as the handling of canonical signatures and F^ω types are similar. We use a version of F^ω with explicit subtyping, so the judgment is extended to produce a *subtyping function* $f : \Pi_1 \rightarrow \Pi_2$. The nature of this subtyping function gives interesting insights on the compilation of modules. For abstraction-only subtyping, the subtyping function is $\beta\eta$ -convertible to the identity as the memory representation of modules (the terms) are the same when stripped of types. For general subtyping (with reordering or deletion of fields), the subtyping function is not *code-free*; correspondingly, it requires a specific action from the compiler (copy, access table, etc.).

The rule for functor subtyping **E-SUB-SIG-FUNCTOR** illustrates the main mechanisms: building on the subtyping functions (and type matching $\bar{\tau}$) for the parameter and the body, we craft a subtyping function from the left-hand-side functor signature to the right-hand-side one. All

the rules are given in §F.1.

E-SUB-SIG-FUNCTOR

$$\frac{\Delta, \bar{\alpha}' \vdash^{\text{elab}} \mathcal{R}'_a <: \mathcal{R}_a[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow \Sigma'_a <: \Sigma_a[\bar{\tau} \mapsto \bar{\alpha}] \Rightarrow f_a}{\Delta, \bar{\alpha}', (Y : (\mathcal{R}'_a, \Sigma'_a)) \vdash^{\text{elab}} \mathcal{S}[\bar{\alpha} \mapsto \bar{\tau}] <: \mathcal{S}' \rightsquigarrow \Pi[\bar{\alpha} \mapsto \bar{\tau}] <: \Pi' \Rightarrow f \quad Y \notin \Delta} \Delta \vdash^{\text{elab}} \forall \bar{\alpha}. (Y : \mathcal{R}_a) \rightarrow \mathcal{S} <: \forall \bar{\alpha}'. (Y : \mathcal{R}'_a) \rightarrow \mathcal{S}' \rightsquigarrow \forall \bar{\alpha}. \Sigma_a \rightarrow \Pi <: \forall \bar{\alpha}'. \Sigma'_a \rightarrow \Pi' \\ [\lambda (g : (\forall \bar{\alpha}. \Sigma_a \rightarrow \Pi)). \Lambda \bar{\alpha}'. \lambda (x : \Sigma'_a). f (g \bar{\tau} (f_a x))]$$

B.2.2 Typing

As for subtyping, the typing rules extend the canonical ones and produce an F^ω type (and term for module and bindings). When handling signatures and types, the lifting of existentials is just a rewriting. Producing the actual proof terms shows how the extension of scope is made using the repacking pattern. All the rules are given in §F.2.

Binding typing rules A binder is encoded with a single-field declaration record. The extrusion of existential types is illustrated in Rule **E-TYP-DECL-MOD**. In the sequence rule **E-TYP-DECL-SEQ**, two bindings are merged using a let-binding for renaming. The extension of scope needed to make the abstract types of the first declaration $\bar{\alpha}_1$ available to the rest also uses the repacking pattern.

E-TYP-DECL-MOD

$$\frac{\Delta \vdash^{\text{elab}} M : \exists \bar{\alpha}. \mathcal{R} \rightsquigarrow e : \exists \bar{\alpha}. \Sigma \quad A.X \notin \Delta}{\Delta \vdash^{\text{elab}}_A (\text{module } X = M) : (\exists \bar{\alpha}. \text{module } X : \mathcal{R}) \rightsquigarrow \text{unpack} \langle \bar{\alpha}, y \rangle = e \text{ in } \text{pack} \langle \bar{\alpha}, \{\ell_X = y\} \rangle : (\exists \bar{\alpha}. \text{module } X : \Sigma)}$$

E-TYP-DECL-SEQ

$$\frac{\Delta \vdash^{\text{elab}}_A B_1 : \exists \bar{\alpha}_1. \mathcal{D}_1 \rightsquigarrow e_1 : \exists \bar{\alpha}_1. \{\zeta_1\} \quad \Delta, \bar{\alpha}_1, A.I_1 : (\mathcal{D}_1, \zeta_1) \vdash^{\text{elab}}_A \bar{B} : \exists \bar{\alpha}. \bar{\mathcal{D}} \rightsquigarrow e : \exists \bar{\alpha}. \{\bar{\zeta}\}}{\Delta \vdash^{\text{elab}}_A B_1, \bar{B} : \exists \bar{\alpha}_1 \bar{\alpha}. \bar{\mathcal{D}}_1, \bar{\mathcal{D}} \rightsquigarrow \text{unpack} \langle \bar{\alpha}_1, x_1 \rangle = e_1 \text{ in } \text{unpack} \langle \bar{\alpha}, x \rangle = (\text{let } A.I_1 = x_1.\ell_{I_1} \text{ in } e) \text{ in } \text{pack} \langle \bar{\alpha}_1 \bar{\alpha}, \{\ell_I = x_1.\ell_I, \bar{\ell}_I = x.\ell_I\} \rangle : \exists \bar{\alpha}_1 \bar{\alpha}. \{\zeta_1, \bar{\zeta}\}}$$

Module expressions typing We only present the key typing rules for module expressions. The introduction of abstract types is done by sealing (**E-TYP-MOD-SEALING**), and features an explicit isolated *packing* (as opposed to a repacking). The application of functor (**E-TYP-MOD-APP**) shows how the polymorphic interface (universally quantified) is instantiated with types (obtained from the matching of the argument's and parameter's signatures). Finally, the projection rule (**E-TYP-MOD-PROJ**) illustrates another usage of the repacking pattern to collect

abstract types, project on the X submodule, and put back the abstract types.

$$\begin{array}{c}
\text{E-TYP-MOD-SEALING} \\
\frac{\Delta \vdash^{\text{elab}} S : \lambda \bar{\alpha}. \mathcal{R} \rightsquigarrow \lambda \bar{\alpha}. \Sigma \quad \Delta \vdash^{\text{elab}} P : \mathcal{R}' \rightsquigarrow e : \Sigma'}{\Delta \vdash^{\text{elab}} \mathcal{R}' <: \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow \Sigma' <: \Sigma[\bar{\alpha} \mapsto \bar{\tau}] \Rightarrow f} \\
\Delta \vdash^{\text{elab}} (P : S) : \exists \bar{\alpha}. \mathcal{R} \rightsquigarrow \text{pack}(\bar{\tau}, (f e)) : \lambda \bar{\alpha}. \Sigma
\end{array}
\qquad
\begin{array}{c}
\text{E-TYP-MOD-APP} \\
\frac{\Delta \vdash^{\text{elab}} P' : \mathcal{R}' \rightsquigarrow e' : \Sigma' \quad \Delta \vdash^{\text{elab}} P : \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S} \rightsquigarrow e : \forall \bar{\alpha}. \Sigma \rightarrow \mathcal{S}}{\Delta \vdash^{\text{elab}} \mathcal{R}' <: \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow \Sigma' <: \Sigma[\bar{\alpha} \mapsto \bar{\tau}] \Rightarrow f} \\
\Delta \vdash^{\text{elab}} P(P') : \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow e \bar{\tau} (f e') : \Pi[\bar{\alpha} \mapsto \bar{\tau}]
\end{array}$$

$$\begin{array}{c}
\text{E-TYP-MOD-PROJ} \\
\frac{\Delta \vdash^{\text{elab}} M : \exists \bar{\alpha}. \text{sig}_A \bar{D}_1, \text{module } X : \mathcal{R}, \bar{D}_2 \text{ end} \rightsquigarrow e : \{\bar{\zeta}_1, \ell_X : \Sigma, \bar{\zeta}_2\}}{\Delta \vdash^{\text{elab}} M.X : \exists \bar{\alpha}. \mathcal{R} \rightsquigarrow \text{unpack}(\bar{\alpha}, y) = e \text{ in } \text{pack}(\bar{\alpha}, e.\ell_X) : \exists \bar{\alpha}. \Sigma}
\end{array}$$

B.3 Correctness and link with the canonical system

We write Δ^{can} and Δ^{F} for the left and right projections of the hybrid context Δ , defined in the obvious way. We present implications as rules.

Theorem 4 (Correctness of elaboration). The elaboration judgment ensures well-typedness on both sides. For subtyping, we have the following result:

$$\Delta \vdash^{\text{elab}} \mathcal{S} <: \mathcal{S}' \rightsquigarrow \Pi <: \Pi' \Rightarrow f \implies \begin{cases} \Delta^{\text{F}} \vdash^{\text{can}} \mathcal{S} <: \mathcal{S}' \\ \Delta^{\text{F}} \vdash f : \Pi \rightarrow \Pi' \end{cases} \quad (9)$$

For typing, we have the following result:

$$\Delta \vdash^{\text{elab}} M : \mathcal{S} \rightsquigarrow e : \Pi \implies \begin{cases} \Delta^{\text{can}} \vdash^{\text{can}} M : \mathcal{S} \\ \Delta^{\text{F}} \vdash e : \Pi \end{cases} \quad (10)$$

Conversely, we show that well-typedness in the canonical system ensures that elaboration is well-defined. We first observe that there is an isomorphism between canonical and F^ω -encoded signatures. This allows us, for a given signature \mathcal{S} (resp. \mathcal{R}) to consider its encoding in F^ω , noted $\Pi_{\mathcal{S}}$ (resp. $\Sigma_{\mathcal{R}}$). Similarly, we may consider the extension of an environment Γ into a hybrid enriched environment Δ_{Γ} (we omit the definition.)

Theorem 5 (Enrichment of canonical judgments). The F^ω terms ($e, f, \Pi_{\mathcal{S}}$) corresponding to canonical derivation always exist. More formally, we have:

$$\Gamma \vdash^{\text{can}} \mathcal{S} : \mathcal{S} \implies \Delta_{\Gamma} \vdash^{\text{elab}} \mathcal{S} : \mathcal{S} \rightsquigarrow \Pi_{\mathcal{S}} \quad (11)$$

$$\Gamma \vdash^{\text{can}} \mathcal{S}_1 <: \mathcal{S}_2 \implies \exists f, \Delta_{\Gamma} \vdash^{\text{elab}} \mathcal{S}_1 <: \mathcal{S}_2 \rightsquigarrow \Pi_{\mathcal{S}_1} <: \Pi_{\mathcal{S}_2} \Rightarrow f \quad (12)$$

$$\Gamma \vdash^{\text{can}} M : \mathcal{S} \implies \exists e, \Delta_{\Gamma} \vdash^{\text{elab}} M : \mathcal{S} \rightsquigarrow e : \Pi_{\mathcal{S}} \quad (13)$$

C Source system – Complete rules

C.1 Resolution

C.1.1 $\Gamma \vdash^{\text{src}} P \triangleright S$ – Path resolution

$\frac{\text{S-RES-LOCALMOD} \quad (A.X : S) \in \Gamma}{\Gamma \stackrel{\text{src}}{\vdash} A.X \triangleright S}$	$\frac{\text{S-RES-FCTARG} \quad (Y : S) \in \Gamma}{\Gamma \stackrel{\text{src}}{\vdash} Y \triangleright S}$	$\frac{\text{S-RES-PROJ-MOD} \quad \Gamma \stackrel{\text{src}}{\vdash} P \triangleright \text{sig}_A \overline{D} \text{ end} \quad (\text{module } X : S) \in \overline{D}}{\Gamma \stackrel{\text{src}}{\vdash} P.X \triangleright S[A \mapsto P]}$
$\frac{\text{S-RES-MODTYPE} \quad \Gamma \stackrel{\text{src}}{\vdash} P \triangleright Q.T \quad \Gamma \stackrel{\text{src}}{\vdash} Q.T \triangleleft S}{\Gamma \stackrel{\text{src}}{\vdash} P \triangleright S}$		

C.1.2 $\Gamma \stackrel{\text{src}}{\vdash} S \triangleleft S'$ – Signature resolution

$\frac{\text{S-RES-LOCALSIG} \quad (A.T : S) \in \Gamma}{\Gamma \stackrel{\text{src}}{\vdash} A.T \triangleleft S}$	$\frac{\text{S-RES-PROJ-MODTYPE} \quad \Gamma \stackrel{\text{src}}{\vdash} P \triangleright \text{sig}_A \overline{D} \text{ end} \quad (\text{module type } T = S) \in \overline{D}}{\Gamma \stackrel{\text{src}}{\vdash} P.T \triangleleft S[A \mapsto P]}$
$\frac{\text{S-RES-MODTYPE-REC} \quad \Gamma \stackrel{\text{src}}{\vdash} Q.T \triangleleft Q'.T' \quad \Gamma \stackrel{\text{src}}{\vdash} Q'.T' \triangleleft S}{\Gamma \stackrel{\text{src}}{\vdash} Q.T \triangleleft S}$	

C.2 Equivalence

C.2.1 $\Gamma \stackrel{\text{src}}{\vdash} \tau \approx \tau'$ – Type equivalence

$\frac{\text{S-EQV-TYPE-LOCAL} \quad (A.t : \text{type } t = \tau) \in \Gamma}{\Gamma \stackrel{\text{src}}{\vdash} A.t \approx \tau}$	$\frac{\text{S-EQV-TYPE-RES} \quad \Gamma \stackrel{\text{src}}{\vdash} P \triangleright \text{sig}_A \overline{D} \text{ end} \quad (\text{type } t = \tau) \in \overline{D}}{\Gamma \stackrel{\text{src}}{\vdash} P.t \approx \tau[A \mapsto P]}$
$\frac{\text{S-EQV-TYPE-TRANS} \quad \Gamma \stackrel{\text{src}}{\vdash} \tau_1 \approx \tau_2 \quad \Gamma \stackrel{\text{src}}{\vdash} \tau_2 \approx \tau_3}{\Gamma \stackrel{\text{src}}{\vdash} \tau_1 \approx \tau_3}$	$\frac{\text{S-EQV-TYPE-SYM} \quad \Gamma \stackrel{\text{src}}{\vdash} \tau' \approx \tau}{\Gamma \stackrel{\text{src}}{\vdash} \tau \approx \tau'}$
$\text{S-EQV-TYPE-REFL} \quad \Gamma \stackrel{\text{src}}{\vdash} \tau \approx \tau$	

C.2.2 $\Gamma \stackrel{\text{src}}{\vdash}_A D \approx D'$ – Declaration equivalence

$\frac{\text{S-EQV-DECL-VAL} \quad \Gamma \vdash^{\text{src}} \tau \approx \tau'}{\Gamma \vdash_A^{\text{src}} (\text{val } x : \tau) \approx (\text{val } x : \tau')}$	$\frac{\text{S-EQV-DECL-TYPE} \quad \Gamma \vdash^{\text{src}} \tau \approx \tau'}{\Gamma \vdash_A^{\text{src}} (\text{type } t = \tau) \approx (\text{type } t = \tau')}$
$\frac{\text{S-EQV-DECL-MOD} \quad \Gamma \vdash^{\text{src}} S \approx S'}{\Gamma \vdash_A^{\text{src}} (\text{module } X : S) \approx (\text{module } X : S')}$	
$\frac{\text{S-EQV-DECL-MODTYPE} \quad \Gamma \vdash^{\text{src}} S \approx S'}{\Gamma \vdash_A^{\text{src}} (\text{module type } T = S) \approx (\text{module type } T = S')}$	$\frac{\text{S-EQV-DECL-EMPTY} \quad \Gamma \vdash^{\text{src}} \emptyset \approx \emptyset}{\Gamma \vdash^{\text{src}} \emptyset \approx \emptyset}$
$\frac{\text{S-EQV-DECL-SEQ} \quad \Gamma \vdash_A^{\text{src}} D_1 \approx D'_1 \quad \Gamma, A.I_1 : D_1 \vdash_A^{\text{src}} \bar{D} \approx \bar{D}'}{\Gamma \vdash_A^{\text{src}} D_1, \bar{D} \approx D'_1, \bar{D}'}$	

C.2.3 $\Gamma \vdash^{\text{src}} S \approx S'$ – Signature equivalence

$\frac{\text{S-EQV-SIG-MODTYPE} \quad \Gamma \vdash^{\text{src}} Q.T \triangleleft S}{\Gamma \vdash^{\text{src}} Q.T \approx S}$	$\frac{\text{S-EQV-SIG-FUNCTOR} \quad \Gamma \vdash^{\text{src}} S_a \approx S'_a \quad \Gamma, Y : S_a \vdash^{\text{src}} S \approx S'}{\Gamma \vdash^{\text{src}} (Y : S_a) \rightarrow S \approx (Y : S'_a) \rightarrow S'}$
$\frac{\text{S-EQV-SIG-SIG} \quad \Gamma \vdash_A^{\text{src}} \bar{D} \approx \bar{D}'}{\Gamma \vdash^{\text{src}} \text{sig}_A \bar{D} \text{ end} \approx \text{sig}_A \bar{D}' \text{ end}}$	$\frac{\text{S-EQV-SIG-TRANS} \quad \Gamma \vdash^{\text{src}} S \approx S' \quad \Gamma \vdash^{\text{src}} S' \approx S''}{\Gamma \vdash^{\text{src}} S \approx S''}$
	$\frac{\text{S-EQV-SIG-REFL} \quad \Gamma \vdash^{\text{src}} S \approx S}{\Gamma \vdash^{\text{src}} S \approx S}$
	$\frac{\text{S-EQV-SIG-SYM} \quad \Gamma \vdash^{\text{src}} S' \approx S}{\Gamma \vdash^{\text{src}} S \approx S'}$

C.3 Strengthening

C.3.1 $\Gamma \vdash S/P \gg S'$ – Signature strengthening

$\frac{\text{S-STR-SIG-FUNCTOR} \quad \Gamma \vdash (Y : S_a) \rightarrow S/P \gg (Y : S_a) \rightarrow S}{\Gamma \vdash (Y : S_a) \rightarrow S/P \gg (Y : S_a) \rightarrow S}$	$\frac{\text{S-STR-SIG-SIG} \quad \Gamma \vdash \bar{D}[A \mapsto P]/P \gg \bar{D}'}{\Gamma \vdash \text{sig}_A \bar{D} \text{ end}/P \gg \text{sig}_A \bar{D}' \text{ end}}$
--	---

C.3.2 $\Gamma \vdash D/P \gg D'$ – Declaration strengthening

S-STR-DECL-VAL $\Gamma \vdash \text{val } x : \tau / P \gg \text{val } x : \tau$	S-STR-DECL-TYPE $\Gamma \vdash \text{type } t = \tau / P \gg \text{type } t = \tau$
S-STR-DECL-MOD $\frac{\Gamma \vdash S / (P.X) \gg S'}{\Gamma \vdash (\text{module } X : S) / P \gg \text{module } X : S'}$	
$\text{S-STR-DECL-MODTYPE}$ $\Gamma \vdash \text{module type } T = S / P \gg \text{module type } T = S$	

C.4 Subtyping

Rules for both general and abstraction subtyping are shown, \prec : being either $<$: or \triangleleft :

C.4.1 $\Gamma \vdash^{\text{sfc}} S \prec: S'$ – Signature subtyping

S-SUB-SIG-EQUIV $\frac{\Gamma \vdash^{\text{sfc}} S \approx S'}{\Gamma \vdash^{\text{sfc}} S \prec: S'}$	S-SUB-SIG-TRANS $\frac{\Gamma \vdash^{\text{sfc}} S \prec: S' \quad \Gamma \vdash^{\text{sfc}} S' \prec: S''}{\Gamma \vdash^{\text{sfc}} S \prec: S''}$
S-SUB-SIG-FUNCTOR $\frac{\Gamma \vdash^{\text{sfc}} S'_a \prec: S_a \quad \Gamma, Y : S'_a \vdash^{\text{sfc}} S \prec: S' \quad Y \notin \Gamma}{\Gamma \vdash^{\text{sfc}} (Y : S_a) \rightarrow S \prec: (Y : S'_a) \rightarrow S'}$	S-SUB-SIG-SIG $\frac{\overline{D}_0 \subseteq \overline{D} \quad \Gamma, \overline{A}.\overline{D} \vdash_A^{\text{sfc}} \overline{D}_0 \prec: \overline{D}'}{\Gamma \vdash^{\text{sfc}} \text{sig}_A \overline{D} \text{ end} \prec: \text{sig}_A \overline{D}' \text{ end}}$
S-SUBEQ-SIG-SIG $\frac{\Gamma, \overline{A}.\overline{D} \vdash^{\text{sfc}} \overline{D} \prec: \overline{D}'}{\Gamma \vdash^{\text{sfc}} \text{sig}_A \overline{D} \text{ end} \triangleleft: \text{sig}_A \overline{D}' \text{ end}}$	

C.4.2 $\Gamma \vdash_A^{\text{sfc}} D \prec: D'$ – Declaration subtyping

S-SUB-DECL-VAL $\frac{\Gamma \vdash^{\text{sfc}} \tau \approx \tau'}{\Gamma \vdash_A^{\text{sfc}} (\text{val } x : \tau) \prec: (\text{val } x : \tau')}$	S-SUB-DECL-TYPE $\frac{\Gamma \vdash^{\text{sfc}} \tau \approx \tau'}{\Gamma \vdash_A^{\text{sfc}} (\text{type } t = \tau) \prec: (\text{type } t = \tau')}$
S-SUB-DECL-MOD $\frac{\Gamma \vdash^{\text{sfc}} S \prec: S'}{\Gamma \vdash_A^{\text{sfc}} (\text{module } X : S) \prec: (\text{module } X : S')}$	
$\text{S-SUB-DECL-MODTYPE}$ $\frac{\Gamma \vdash^{\text{sfc}} S \prec: S' \quad \Gamma \vdash^{\text{sfc}} S' \prec: S}{\Gamma \vdash_A^{\text{sfc}} (\text{module type } T = S) \prec: (\text{module type } T = S')}$	

C.5 Typing

C.5.1 $\Gamma \vdash^{\text{src}} S : \checkmark$ – Signature typing

$\frac{\text{S-TYP-SIG-LOCALMODTYPE} \quad \Gamma \vdash^{\text{src}} P.T : \text{sig}_A \overline{D} \text{ end} \quad \text{module type } T = S \in \overline{D}}{\Gamma \vdash^{\text{src}} P.T : \checkmark}$	$\frac{\text{S-TYP-SIG-MODTYPE} \quad A.T : \text{module type } T = S \in \Gamma}{\Gamma \vdash^{\text{src}} A.T : \checkmark}$
$\frac{\text{S-TYP-SIG-FUNCTOR} \quad \Gamma \vdash^{\text{src}} S_a : \checkmark \quad \Gamma, Y : S_a \vdash^{\text{src}} S : \checkmark \quad Y \notin \Gamma}{\Gamma \vdash^{\text{src}} (Y : S_a) \rightarrow S : \checkmark}$	$\frac{\text{S-TYP-SIG-SIG} \quad \Gamma \vdash^{\text{src}} \overline{D} : \checkmark \quad A \notin \Gamma}{\Gamma \vdash^{\text{src}} \text{sig}_A \overline{D} \text{ end} : \checkmark}$

C.5.2 $\Gamma \vdash_A^{\text{src}} D : \checkmark$ – Declarations typing

$\frac{\text{S-TYP-DECL-VAL} \quad \Gamma \vdash^{\text{src}} \tau : \checkmark \quad A.x \notin \Gamma}{\Gamma \vdash_A^{\text{src}} (\text{val } x : \tau) : \checkmark}$	$\frac{\text{S-TYP-DECL-TYPE} \quad \Gamma \vdash^{\text{src}} \tau : \checkmark \quad A.t \notin \Gamma}{\Gamma \vdash_A^{\text{src}} (\text{type } t = \tau) : \checkmark}$	$\frac{\text{S-TYP-DECL-TYPEABS} \quad A.t \notin \Gamma}{\Gamma \vdash_A^{\text{src}} (\text{type } t = A.t) : \checkmark}$
$\frac{\text{S-TYP-DECL-MOD} \quad \Gamma \vdash^{\text{src}} S : \checkmark \quad A.X \notin \Gamma}{\Gamma \vdash_A^{\text{src}} (\text{module } X : S) : \checkmark}$	$\frac{\text{S-TYP-DECL-MODTYPE} \quad \Gamma \vdash^{\text{src}} S : \checkmark \quad A.T \notin \Gamma}{\Gamma \vdash_A^{\text{src}} (\text{module type } T = S) : \checkmark}$	$\frac{\text{S-TYP-DECL-EMPTY}}{\Gamma \vdash_A^{\text{src}} \emptyset : \checkmark}$
$\frac{\text{S-TYP-DECL-SEQ} \quad \Gamma \vdash_A^{\text{src}} D_1 : \checkmark \quad \Gamma, A.I_1 : D_1 \vdash_A^{\text{src}} \overline{D} : \checkmark}{\Gamma \vdash_A^{\text{src}} (D_1, \overline{D}) : \checkmark}$		

C.5.3 $\Gamma \vdash^{\text{src}} \tau : \checkmark$ – Type checking

$\frac{\text{S-TYP-TYPE-LOCALTYPE} \quad A.t : \text{type } t = \tau \in \Gamma}{\Gamma \vdash^{\text{src}} A.t : \checkmark}$	$\frac{\text{S-TYP-TYPE-QUALIFIEDPATHTYPE} \quad \Gamma \vdash^{\text{src}} P : \text{sig}_A \overline{D} \text{ end} \quad \text{type } t = \tau \in \overline{D}}{\Gamma \vdash^{\text{src}} P.t : \checkmark}$
--	---

C.5.4 $\Gamma \vdash^{\text{src}} M : S$ – Module typing

$\frac{\text{S-TYP-MOD-RES}}{\Gamma \overset{\text{src}}{\vdash} P \triangleright S} \quad \Gamma \overset{\text{src}}{\vdash} P : S$	$\frac{\text{S-TYP-MOD-STRENGTHEN}}{\Gamma \overset{\text{src}}{\vdash} P : S' \quad \Gamma \vdash S/P \gg S'} \quad \Gamma \overset{\text{src}}{\vdash} P : S$	$\frac{\text{S-TYP-MOD-EQUIV}}{\Gamma \overset{\text{src}}{\vdash} M : S' \quad \Gamma \overset{\text{src}}{\vdash} S \approx S'} \quad \Gamma \overset{\text{src}}{\vdash} M : S$
$\frac{\text{S-TYP-MOD-SEALING}}{\Gamma \overset{\text{src}}{\vdash} (P : S) : S} \quad \Gamma \overset{\text{src}}{\vdash} S : \checkmark \quad \Gamma \overset{\text{src}}{\vdash} P : S' \quad \Gamma \overset{\text{src}}{\vdash} S' <: S$	$\frac{\text{S-TYP-MOD-FUNCTOR}}{\Gamma \overset{\text{src}}{\vdash} (Y : S_a) \rightarrow M : (Y : S_a) \rightarrow S'} \quad \Gamma \overset{\text{src}}{\vdash} S_a : \checkmark \quad \Gamma; (Y : S_a) \overset{\text{src}}{\vdash} M : S \quad Y \notin \Gamma$	
$\frac{\text{S-TYP-MOD-FCTAPP}}{\Gamma \overset{\text{src}}{\vdash} P(P') : S[Y \mapsto P']} \quad \Gamma \overset{\text{src}}{\vdash} P : (Y : S_a) \rightarrow S \quad \Gamma \overset{\text{src}}{\vdash} P' : S'_a \quad \Gamma \overset{\text{src}}{\vdash} S'_a <: S_a$	$\frac{\text{S-TYP-MOD-STRUCT}}{\Gamma \overset{\text{src}}{\vdash} \text{struct}_A \bar{B} \text{ end} : \text{sig}_A \bar{D} \text{ end}} \quad \Gamma \overset{\text{src}}{\vdash}_A \bar{B} : \bar{D} \quad A \notin \Gamma$	
$\frac{\text{S-TYP-PROJ}}{\Gamma \overset{\text{src}}{\vdash} M.X : S'} \quad \Gamma \overset{\text{src}}{\vdash} M : \text{sig}_A (\bar{D}_1, \text{module } X : S, \bar{D}_2) \text{ end} \quad \Gamma, \bar{D}_1 \overset{\text{src}}{\vdash} S <: S' \quad \Gamma \overset{\text{src}}{\vdash} S' : \checkmark$		

C.5.5 $\Gamma \overset{\text{src}}{\vdash}_A B : D$ – Bindings typing

$\frac{\text{S-TYP-BIND-LET}}{\Gamma \overset{\text{src}}{\vdash}_A (\text{let } x = e) : (\text{val } x : \tau)} \quad \Gamma \overset{\text{src}}{\vdash} e : \tau \quad A.x \notin \Gamma$	$\frac{\text{S-TYP-BIND-TYPE}}{\Gamma \overset{\text{src}}{\vdash}_A (\text{type } t = \tau) : (\text{type } t = \tau)} \quad \Gamma \overset{\text{src}}{\vdash} \tau : \checkmark \quad A.t \notin \Gamma$
$\frac{\text{S-TYP-BIND-ABSType}}{\Gamma \overset{\text{src}}{\vdash}_A (\text{type } t = A.t) : (\text{type } t = A.t)} \quad A.t \notin \Gamma$	$\frac{\text{S-TYP-BIND-MOD}}{\Gamma \overset{\text{src}}{\vdash}_A (\text{module } X = M) : (\text{module } X : S)} \quad \Gamma \overset{\text{src}}{\vdash} M : S \quad A.X \notin \Gamma$
$\frac{\text{S-TYP-BIND-MODType}}{\Gamma \overset{\text{src}}{\vdash}_A (\text{module type } T = S) : (\text{module type } T = S)} \quad \Gamma \overset{\text{src}}{\vdash} S : \checkmark \quad A.T \notin \Gamma$	$\frac{\text{S-TYP-BIND-EMPTY}}{\Gamma \overset{\text{src}}{\vdash}_A \emptyset : \emptyset}$
$\frac{\text{S-TYP-BIND-SEQ}}{\Gamma \overset{\text{src}}{\vdash}_A B_1, B_2 : D_1 \uparrow D_2} \quad \Gamma \overset{\text{src}}{\vdash}_A B_1 : D_1 \quad \Gamma, A.I_1 : D_1 \overset{\text{src}}{\vdash}_A B_2 : D_2$	

D Canonical system – Complete rules

D.1 Subtyping

Rules for both general and abstraction subtyping are shown, \prec : being either $<$: or \prec :.

D.1.1 $\Gamma \vdash^{\text{can}} \mathcal{S} \prec: \mathcal{S}'$ – Signatures subtyping

<p>C-SUB-SIG-FUNCTOR</p> $\frac{\Gamma, \bar{\alpha}' \vdash^{\text{can}} \mathcal{R}' \prec: \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}] \quad \Gamma, \bar{\alpha}', (Y : \mathcal{R}') \vdash^{\text{can}} \mathcal{S}[\bar{\alpha} \mapsto \bar{\tau}] \prec: \mathcal{S}' \quad Y \notin \Gamma}{\Gamma \vdash^{\text{can}} \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S} \prec: \forall \bar{\alpha}'. (Y : \mathcal{R}') \rightarrow \mathcal{S}'}$	
<p>C-SUB-SIG-MATCH</p> $\frac{\Gamma, \bar{\alpha} \vdash^{\text{can}} \mathcal{R} \prec: \mathcal{R}'[\bar{\alpha}' \mapsto \bar{\tau}]}{\Gamma \vdash^{\text{can}} \exists \bar{\alpha}. \mathcal{R} \prec: \exists \bar{\alpha}'. \mathcal{R}'}$	<p>C-SUB-SIG-SIG</p> $\frac{\bar{\mathcal{D}}_0 \subseteq \bar{\mathcal{D}} \quad \Gamma \vdash^{\text{can}} \bar{\mathcal{D}}_0 \prec: \bar{\mathcal{D}}}{\Gamma \vdash^{\text{can}} \text{sig}_A \bar{\mathcal{D}} \text{ end} \prec: \text{sig}_A \bar{\mathcal{D}}' \text{ end}}$
<p>C-SUBEQ-SIG-SIG</p> $\frac{\Gamma \vdash^{\text{can}} \bar{\mathcal{D}} \prec: \bar{\mathcal{D}}'}{\Gamma \vdash^{\text{can}} \text{sig}_A \bar{\mathcal{D}} \text{ end} \prec: \text{sig}_A \bar{\mathcal{D}}' \text{ end}}$	

D.1.2 $\Gamma \vdash^{\text{can}} \mathcal{D} \prec: \mathcal{D}'$ – Declarations subtyping

<p>C-SUB-DECL-VAL</p> $\frac{\Gamma \vdash^{\text{can}} \tau \prec: \tau'}{\Gamma \vdash^{\text{can}} (\text{val } x : \tau) \prec: (\text{val } x : \tau')}$	<p>C-SUB-DECL-TYPE</p> $\frac{\Gamma \vdash^{\text{can}} \tau \prec: \tau'}{\Gamma \vdash^{\text{can}} (\text{type } t = \tau) \prec: (\text{type } t = \tau')}$
<p>C-SUB-DECL-MOD</p> $\frac{\Gamma \vdash^{\text{can}} \mathcal{R} \prec: \mathcal{R}'}{\Gamma \vdash^{\text{can}} (\text{module } X : \mathcal{R}) \prec: (\text{module } X : \mathcal{R}')}$	
<p>C-SUB-DECL-MODTYPE</p> $\frac{\Gamma \vdash^{\text{can}} \mathcal{R} \prec: \mathcal{R}' \quad \Gamma \vdash^{\text{can}} \mathcal{R}' \prec: \mathcal{R}}{\Gamma \vdash^{\text{can}} (\text{module type } T = \lambda \bar{\alpha}. \mathcal{R}) \prec: (\text{module type } T = \lambda \bar{\alpha}. \mathcal{R}')}$	

D.2 Typing

D.2.1 $\Gamma \vdash^{\text{can}} Q.t : \tau$ – Type checking

<p>C-TYP-TYPE-LOCALTYPE</p> $\frac{A.t : \text{type } t = \tau \in \Gamma}{\Gamma \vdash^{\text{can}} A.t : \tau}$	<p>C-TYP-TYPE-PATHTYPE</p> $\frac{\Gamma \vdash^{\text{can}} P : \text{sig}_A \bar{\mathcal{D}} \text{ end} \quad (\text{type } t = \tau) \in \bar{\mathcal{D}}}{\Gamma \vdash^{\text{can}} P.t : \tau}$
--	--

D.2.2 $\Gamma \vdash_A^{\text{can}} D : \lambda\bar{\alpha}.\mathcal{D}$ – Declaration typing

$\frac{\text{C-TYP-DECL-VAL} \quad \Gamma \vdash^{\text{can}} \tau : \tau' \quad A.x \notin \Gamma}{\Gamma \vdash_A^{\text{can}} \text{val } x : \tau : \text{val } x : \tau'}$	$\frac{\text{C-TYP-DECL-TYPE} \quad \Gamma \vdash^{\text{can}} \tau : \tau' \quad A.t \notin \Gamma}{\Gamma \vdash_A^{\text{can}} \text{type } t = \tau : \text{type } t = \tau'}$	$\frac{\text{C-TYP-DECL-TYPEABS} \quad A.t \notin \Gamma}{\Gamma \vdash_A^{\text{can}} \text{type } t = A.t : \lambda\alpha. \text{type } t = \alpha}$
$\frac{\text{C-TYP-DECL-MOD} \quad \Gamma \vdash^{\text{can}} S : \lambda\bar{\alpha}.\mathcal{R} \quad A.X \notin \Gamma}{\Gamma \vdash_A^{\text{can}} (\text{module } X : S) : \lambda\bar{\alpha}. (\text{module } X : \mathcal{R})}$		
$\frac{\text{C-TYP-DECL-MODTYPE} \quad \Gamma \vdash^{\text{can}} S : \lambda\bar{\alpha}.\mathcal{R} \quad A.T \notin \Gamma}{\Gamma \vdash_A^{\text{can}} (\text{module type } T = S) : (\text{module type } T = \lambda\bar{\alpha}.\mathcal{R})}$		$\frac{\text{C-TYP-DECL-EMPTY}}{\Gamma \vdash_A^{\text{can}} \emptyset : \emptyset}$
$\frac{\text{C-TYP-DECL-SEQ} \quad \Gamma \vdash_A^{\text{can}} D_1 : \lambda\bar{\alpha}_1.\mathcal{D}_1 \quad \Gamma, \bar{\alpha}_1, A.I_1 : \mathcal{D}_1 \vdash_A^{\text{can}} \bar{D} : \lambda\bar{\alpha}.\bar{D}}{\Gamma \vdash_A^{\text{can}} (D_1, \bar{D}) : \lambda\bar{\alpha}_1 \bar{\alpha}. (D_1, \bar{D})}$		

D.2.3 $\Gamma \vdash^{\text{can}} S : \lambda\bar{\alpha}.\mathcal{R}$ – Signature typing

$\frac{\text{C-TYP-SIG-SIG} \quad \Gamma \vdash_A^{\text{can}} \bar{D} : \lambda\bar{\alpha}.\bar{D} \quad A \notin \Gamma}{\Gamma \vdash^{\text{can}} \text{sig}_A \bar{D} \text{ end} : \lambda\bar{\alpha}.\text{sig}_A \bar{D} \text{ end}}$	$\frac{\text{C-CF-MODTYPE} \quad \Gamma \vdash^{\text{can}} P : \text{sig}_A \bar{D} \text{ end} \quad \text{module type } T = \lambda\bar{\alpha}.\mathcal{R} \in \bar{D}}{\Gamma \vdash^{\text{can}} A.T : \lambda\bar{\alpha}.\mathcal{R}}$
$\frac{\text{C-TYP-SIG-LOCALMODTYPE} \quad (A.T : \text{module type } T = \lambda\bar{\alpha}.\mathcal{R}) \in \Gamma}{\Gamma \vdash^{\text{can}} A.T : \lambda\bar{\alpha}.\mathcal{R}}$	
$\frac{\text{C-TYP-SIG-FUNCTOR} \quad \Gamma \vdash^{\text{can}} S_a : \lambda\bar{\alpha}.\mathcal{R}_a \quad \Gamma, \bar{\alpha}, Y : \mathcal{R}_a \vdash^{\text{can}} S : \lambda\bar{\beta}.\mathcal{R} \quad Y \notin \Gamma}{\Gamma \vdash^{\text{can}} (Y : S_a) \rightarrow S : \forall\bar{\alpha}. (Y : \mathcal{R}_a) \rightarrow \exists\bar{\beta}.\mathcal{R}}$	

D.2.4 $\Gamma \vdash^{\text{can}} M : S$ – Module typing

$\frac{\text{C-TYP-MOD-FCTARG} \quad (Y : \mathcal{R}) \in \Gamma}{\Gamma \vdash^{\text{can}} Y : \mathcal{R}}$	$\frac{\text{C-TYP-MOD-LOCAL} \quad (A.X : \mathcal{R}) \in \Gamma}{\Gamma \vdash^{\text{can}} A.X : \mathcal{R}}$	$\frac{\text{C-TYP-STRUCT} \quad \Gamma \vdash_A^{\text{can}} B : \exists \bar{\alpha}. \bar{\mathcal{D}} \quad A \notin \Gamma}{\Gamma \vdash^{\text{can}} \text{struct}_A B \text{ end} : \exists \bar{\alpha}. \text{sig}_A \bar{\mathcal{D}} \text{ end}}$
$\frac{\text{C-TYP-MOD-SEALING} \quad \Gamma \vdash^{\text{can}} S : \lambda \bar{\alpha}. \mathcal{R} \quad \Gamma \vdash^{\text{can}} P : \mathcal{R}' \quad \Gamma \vdash^{\text{can}} \mathcal{R}' <: \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}]}{\Gamma \vdash^{\text{can}} (P : S) : \exists \bar{\alpha}. \mathcal{R}}$		
$\frac{\text{C-TYP-MOD-FUNCTOR} \quad \Gamma \vdash^{\text{can}} S : \lambda \bar{\alpha}. \mathcal{R} \quad \Gamma, \bar{\alpha}, (Y : \mathcal{R}) \vdash^{\text{can}} M : \mathcal{S}}{\Gamma \vdash^{\text{can}} (Y : S) \rightarrow M : \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S}}$		
$\frac{\text{C-TYP-MOD-APP} \quad \Gamma \vdash^{\text{can}} P : \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S} \quad \Gamma \vdash^{\text{can}} P' : \mathcal{R}' \quad \Gamma \vdash^{\text{can}} \mathcal{R}' <: \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}]}{\Gamma \vdash^{\text{can}} P(P') : \mathcal{S}[\bar{\alpha} \mapsto \bar{\tau}]}$		
$\frac{\text{C-TYP-MOD-PROJ} \quad \Gamma \vdash^{\text{can}} M : \exists \bar{\alpha}. \text{sig}_A \bar{\mathcal{D}}_1, \text{module } X : \mathcal{R}, \bar{\mathcal{D}}_2 \text{ end}}{\Gamma \vdash^{\text{can}} M.X : \exists \bar{\alpha}. \mathcal{R}}$		

D.2.5 $\Gamma \vdash_A^{\text{can}} B : \exists \bar{\alpha}. \bar{\mathcal{D}}$ – Bindings typing

$\frac{\text{C-TYP-DECL-LET} \quad \Gamma \vdash^{\text{can}} e : \tau \quad \Gamma \vdash^{\text{can}} \tau : \tau' \quad A.x \notin \Gamma}{\Gamma \vdash_A^{\text{can}} (\text{let } x = e) : (\text{val } x : \tau')}$	$\frac{\text{C-TYP-DECL-TYPE} \quad \Gamma \vdash^{\text{can}} \tau : \tau' \quad A.t \notin \Gamma}{\Gamma \vdash_A^{\text{can}} (\text{type } t = \tau) : (\text{type } t = \tau')}$
$\frac{\text{C-TYP-DECL-MOD} \quad \Gamma \vdash^{\text{can}} M : \exists \bar{\alpha}. \mathcal{R} \quad A.X \notin \Gamma}{\Gamma \vdash_A^{\text{can}} (\text{module } X = M) : (\exists \bar{\alpha}. \text{module } X : \mathcal{R})}$	
$\frac{\text{C-TYP-DECL-MODTYPE} \quad \Gamma \vdash^{\text{can}} S : \lambda \bar{\alpha}. \mathcal{R} \quad A.T \notin \Gamma}{\Gamma \vdash_A^{\text{can}} (\text{module type } T = S) : (\text{module type } T = \lambda \bar{\alpha}. \mathcal{R})}$	$\text{C-TYP-DECL-EMPTY} \quad \Gamma \vdash_A^{\text{can}} \emptyset : \emptyset$
$\frac{\text{C-TYP-DECL-SEQ} \quad \Gamma \vdash_A^{\text{can}} B_1 : \exists \bar{\alpha}_1. \bar{\mathcal{D}}_1 \quad \Gamma, \bar{\alpha}_1, A.I_1 : \bar{\mathcal{D}}_1 \vdash_A^{\text{can}} \bar{B} : \exists \bar{\alpha}. \bar{\mathcal{D}}}{\Gamma \vdash_A^{\text{can}} B_1, \bar{B} : \exists \bar{\alpha}_1 \bar{\alpha}. \bar{\mathcal{D}}_1, \bar{\mathcal{D}}}$	

D.3 Anchoring

D.3.1 $\Gamma \vdash^{\text{anc}} \tau \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \tau'$ – Type anchoring

$\frac{\text{C-ACH-TYPE-ANCHOR} \quad \Gamma \hookrightarrow \Gamma_s; \theta_\Gamma \quad \theta_\Gamma(\alpha) = Q.t}{\Gamma \vdash^{\text{anc}} \alpha \hookrightarrow \Gamma_s; \theta_\Gamma \vdash Q.t}$	$\frac{\text{C-ACH-DECL-TYPE} \quad \Gamma \vdash^{\text{anc}} \tau \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \tau' : \emptyset}{\Gamma \vdash_A^{\text{anc}} \text{type } t = \tau \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \text{type } t = \tau' : \emptyset}$
--	--

D.3.2 $\Gamma \vdash_A^{\text{anc}} D \hookrightarrow \Gamma_s; \theta_\Gamma \vdash D : \theta$ – Declaration anchoring

$\frac{\text{C-ACH-DECL-VAL} \quad \Gamma \vdash^{\text{anc}} \tau \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \tau' : \emptyset}{\Gamma \vdash_A^{\text{anc}} \text{val } x : \tau \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \text{val } x : \tau' : \emptyset}$	$\frac{\text{C-ACH-DECL-ANCHORPOINT} \quad \Gamma \hookrightarrow \Gamma_s; \theta_\Gamma \quad \alpha \notin \text{dom}(\theta_\Gamma)}{\Gamma \vdash_A^{\text{anc}} \text{type } t = \alpha \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \text{type } t = A.t : (\alpha \mapsto t)}$
$\frac{\text{C-ACH-DECL-TYPE} \quad \Gamma \vdash^{\text{anc}} \tau \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \tau' : \emptyset}{\Gamma \vdash_A^{\text{anc}} \text{type } t = \tau \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \text{type } t = \tau' : \emptyset}$	
$\frac{\text{C-ACH-DECL-MOD} \quad \Gamma \vdash^{\text{anc}} \mathcal{R} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S : \theta}{\Gamma \vdash_A^{\text{anc}} \text{module } X : \mathcal{R} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \text{module } X : S : X.\theta}$	
$\frac{\text{C-ACH-DECL-SIG} \quad \Gamma, \bar{\alpha} \vdash^{\text{anc}} \mathcal{R} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S : \theta \quad \text{dom}(\theta) = \bar{\alpha}}{\Gamma \vdash_A^{\text{anc}} \text{module } T : \lambda \bar{\alpha}. \mathcal{R} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \text{module } T : S : \emptyset}$	
$\frac{\text{C-ACH-DECL-SEQ} \quad \Gamma \vdash^{\text{anc}} D \hookrightarrow \Gamma_s; \theta_\Gamma \vdash D : \theta \quad \Gamma \vdash^{\text{anc}} \bar{D} \hookrightarrow (\Gamma_s, A.I : D); (\theta_\Gamma \uplus A.\theta) \vdash \bar{D} : \theta'}{\Gamma \vdash_A^{\text{anc}} D, \bar{D} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash D, \bar{D} : \theta_\Gamma \uplus \theta'}$	

D.3.3 $\Gamma \vdash^{\text{can}} S : \lambda \bar{\alpha}. \mathcal{R}$ – Signature anchoring

$\frac{\text{C-ACH-SIG-FUNCTOR} \quad \Gamma, \bar{\alpha} \vdash^{\text{anc}} \mathcal{R} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S_a : \theta_a \quad \text{dom}(\theta_a) = \bar{\alpha} \quad \Gamma, \bar{\alpha}, Y : \mathcal{R} \vdash^{\text{anc}} S \hookrightarrow (\Gamma_s, Y : S_a); (\theta_\Gamma \uplus Y.\theta) \vdash S : \theta}{\Gamma \vdash^{\text{anc}} \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow S \hookrightarrow \Gamma_s; \theta_\Gamma \vdash (Y : S_a) \rightarrow S : \emptyset}$
$\frac{\text{C-ACH-SIG-SIG} \quad \Gamma \vdash_A^{\text{anc}} \bar{D} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \bar{D} : \theta}{\Gamma \vdash^{\text{anc}} \text{sig}_A \bar{D} \text{ end} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash \text{sig}_A \bar{D} \text{ end} : \theta}$

D.3.4 $\Gamma \hookrightarrow \Gamma_s : \theta_\Gamma$ – Environment anchoring

<p>C-ACH-ENV-EMPTY $\emptyset \hookrightarrow \emptyset : \emptyset$</p>	<p>C-ACH-ENV-FCTARG</p> $\frac{\Gamma \hookrightarrow \Gamma_s : \theta_\Gamma \quad \Gamma, \bar{\alpha} \vdash^{\text{anc}} \mathcal{R} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S : \theta \quad \text{dom}(\theta) = \bar{\alpha}}{\Gamma, \bar{\alpha}, Y : \mathcal{R} \hookrightarrow (\Gamma_s, Y : S) : \theta_\Gamma \uplus Y.\theta}$
<p>C-ACH-ENV-DECL</p>	$\frac{\Gamma \hookrightarrow \Gamma_s : \theta_\Gamma \quad \Gamma, \bar{\alpha} \vdash^{\text{anc}} \mathcal{D} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash D : \theta \quad \text{dom}(\theta) = \bar{\alpha}}{\Gamma, \bar{\alpha}, A.I : \mathcal{D} \hookrightarrow (\Gamma_s, A.I : D) : \theta_\Gamma \uplus A.\theta}$

D.3.5 $\Gamma \vdash^{\text{anc}} M : S$ – Anchorable typing

<p>C-TYPA-MOD-PROJ</p>	$\frac{\Gamma \vdash^{\text{anc}} M : \exists \bar{\alpha}. \text{sig}_A \bar{\mathcal{D}}_1, \text{module } X : \mathcal{R}, \bar{\mathcal{D}}_2 \text{ end} \quad \Gamma \vdash^{\text{anc}} \exists \bar{\alpha}. \mathcal{R} \hookrightarrow \Gamma_s; \theta_\Gamma \vdash S : \theta}{\Gamma \vdash^{\text{anc}} M.X : \exists \bar{\alpha}. \mathcal{R}}$
------------------------	--

E F^ω – Complete rules

E.1 $\Gamma : \mathbf{wf}$ – Environment checking

$$\begin{array}{c} \cdot : \mathbf{wf} \\ \frac{\Gamma : \mathbf{wf} \quad \alpha \notin \Gamma}{\Gamma, \alpha : \kappa : \mathbf{wf}} \quad \frac{\Gamma \vdash \tau : \star \quad x \notin \Gamma}{\Gamma, (x : \tau) : \mathbf{wf}} \end{array}$$

E.2 $\Gamma \vdash \tau : \kappa$ – Type checking

$$\begin{array}{c} \frac{\Gamma \vdash \tau_1 : \star \quad \Gamma \vdash \tau_2 : \star}{\Gamma \vdash \tau_1 \rightarrow \tau_2 : \star} \quad \frac{\overline{\Gamma \vdash \tau : \star} \quad \Gamma : \mathbf{wf}}{\Gamma \vdash \{\overline{\ell}_i : \tau\} : \star} \quad \frac{\Gamma : \mathbf{wf}}{\Gamma \vdash \alpha : \Gamma(\alpha)} \quad \frac{\Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \forall \alpha : \kappa. \tau : \star} \\ \frac{\Gamma, \alpha : \kappa \vdash \tau : \star}{\Gamma \vdash \exists \alpha : \kappa. \tau : \star} \quad \frac{\Gamma, \alpha : \kappa \vdash \tau : \kappa'}{\Gamma \vdash \lambda \alpha : \kappa. \tau : \kappa \rightarrow \kappa'} \quad \frac{\Gamma \vdash \tau_1 : \kappa' \rightarrow \kappa \quad \Gamma \vdash \tau_2 : \kappa'}{\Gamma \vdash \tau_1 \tau_2 : \kappa} \end{array}$$

E.3 $\Gamma \vdash e : \tau$ – Term typing

$$\begin{array}{c} \frac{\Gamma : \mathbf{wf}}{\Gamma \vdash x : \Gamma(x)} \quad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \quad \frac{\Gamma \vdash e_1 : \tau' \rightarrow \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \\ \frac{\overline{\Gamma \vdash e : \tau} \quad \Gamma : \mathbf{wf}}{\Gamma \vdash \{\overline{l} = e\} : \{\overline{l} : \tau\}} \quad \frac{\Gamma \vdash e : \{\overline{l} : \tau, \overline{l}' : \tau'\}}{\Gamma \vdash e.l : \tau} \quad \frac{\Gamma, \alpha : \kappa \vdash e : \tau}{\Gamma \vdash \Lambda(\alpha : \kappa). e : \forall(\alpha : \kappa). \tau} \\ \frac{\Gamma \vdash e : \forall \alpha : \kappa. \tau' \quad \Gamma \vdash \tau : \kappa}{\Gamma \vdash e \tau : \tau'[\tau \mapsto \alpha]} \quad \frac{\Gamma \vdash \tau : \kappa \quad \Gamma \vdash e : \tau'[\tau \mapsto \alpha] \quad \Gamma \vdash \exists \alpha : \kappa. \tau' : \star}{\Gamma \vdash \text{pack } \langle \tau, e \rangle \text{ as } \exists \alpha : \kappa. \tau' : \exists \alpha : \kappa. \tau'} \\ \frac{\Gamma \vdash e_1 : \exists \alpha : \kappa. \tau' \quad \Gamma, \alpha : \kappa, x : \tau' \vdash e_2 : \tau \quad \Gamma \vdash \tau : \star}{\Gamma \vdash \text{unpack } \langle \alpha, x \rangle = e_1 \text{ in } e_2 : \tau} \end{array}$$

F Elaborated system – Complete rules

F.1 Subtyping

F.1.1 $\Delta \vdash^{\text{elab}} \mathcal{S} <: \mathcal{S}' \rightsquigarrow \Pi <: \Pi' \Rightarrow f$ – Signature subtyping

<p style="margin: 0;">E-SUB-SIG-FUNCTOR</p> $\frac{\Delta, \bar{\alpha}' \vdash^{\text{elab}} \mathcal{R}'_a <: \mathcal{R}_a[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow \Sigma'_a <: \Sigma_a[\bar{\alpha} \mapsto \bar{\tau}] \Rightarrow f_a}{\Delta, \bar{\alpha}', (Y : (\mathcal{R}'_a, \Sigma'_a)) \vdash^{\text{elab}} \mathcal{S}[\bar{\alpha} \mapsto \bar{\tau}] <: \mathcal{S}' \rightsquigarrow \Pi[\bar{\alpha} \mapsto \bar{\tau}] <: \Pi' \Rightarrow f \quad Y \notin \Delta}$ $\Delta \vdash^{\text{elab}} \forall \bar{\alpha}. (Y : \mathcal{R}_a) \rightarrow \mathcal{S} <: \forall \bar{\alpha}'. (Y : \mathcal{R}'_a) \rightarrow \mathcal{S}' \rightsquigarrow \forall \bar{\alpha}. \Sigma_a \rightarrow \Pi <: \forall \bar{\alpha}'. \Sigma'_a \rightarrow \Pi'$ $\left[\lambda (g : (\forall \bar{\alpha}. \Sigma_a \rightarrow \Pi)) . \Lambda \bar{\alpha}'. \lambda (x : \Sigma'_a) . f (g \bar{\tau} (f_a x)) \right]$ <p style="margin: 0;">E-SUB-SIG-SIG</p> $\frac{\bar{\mathcal{D}}_0 \subseteq \bar{\mathcal{D}} \quad \Delta \vdash_A^{\text{elab}} \bar{\mathcal{D}}_0 <: \bar{\mathcal{D}}' \rightsquigarrow \bar{\zeta}_0 <: \bar{\zeta} \Rightarrow \bar{f}}{\Delta \vdash^{\text{elab}} \text{sig}_A \bar{\mathcal{D}} \text{ end} <: \text{sig}_A \bar{\mathcal{D}}' \text{ end} \rightsquigarrow \{\bar{\ell}_I : \bar{\zeta}\} <: \{\bar{\ell}_I : \bar{\zeta}'\} \Rightarrow \lambda x. \{\bar{\ell}_{I_0} : (f \bar{\zeta})\}}$ <p style="margin: 0;">E-SUBEQ-SIG-SIG</p> $\frac{\Delta \vdash_A^{\text{elab}} \bar{\mathcal{D}} <: \bar{\mathcal{D}}' \rightsquigarrow \bar{\zeta} <: \bar{\zeta}' \Rightarrow \bar{f}}{\Delta \vdash^{\text{elab}} \text{sig}_A \bar{\mathcal{D}} \text{ end} <: \text{sig}_A \bar{\mathcal{D}}' \text{ end} \rightsquigarrow \{\bar{\ell}_I : \bar{\zeta}\} <: \{\bar{\ell}_I : \bar{\zeta}'\} \Rightarrow \lambda x. \{\bar{\ell}_I : (f \bar{\zeta})\}}$
--

F.1.2 $\Delta \vdash_A^{\text{elab}} \mathcal{D} <: \mathcal{D}' \rightsquigarrow \zeta <: \zeta' \Rightarrow f$ – Declaration subtyping

<p style="margin: 0;">E-SUB-VAL</p> $\frac{\Delta \vdash^{\text{elab}} \tau <: \tau' \uparrow f}{\Delta \vdash_A^{\text{elab}} (\text{val } x : \tau) <: (\text{val } x : \tau') \rightsquigarrow \llbracket \tau \rrbracket <: \llbracket \tau' \rrbracket \Rightarrow \lambda x. \llbracket f(x.\text{val}) \rrbracket}$ <p style="margin: 0;">E-SUB-TYPE</p> $\frac{\Delta \vdash^{\text{elab}} \tau <: \tau' \uparrow f}{\Delta \vdash_A^{\text{elab}} (\text{type } t = \tau) <: (\text{type } t = \tau') \rightsquigarrow \llbracket \tau : \kappa \rrbracket <: \llbracket \tau' : \kappa \rrbracket \Rightarrow \lambda x. \llbracket \tau' : \kappa \rrbracket}$ <p style="margin: 0;">E-SUB-MOD</p> $\frac{\Delta \vdash^{\text{elab}} \mathcal{R} <: \mathcal{R}' \rightsquigarrow \Sigma <: \Sigma' \Rightarrow f}{\Delta \vdash_A^{\text{elab}} (\text{module } X : \mathcal{R}) <: (\text{module } X : \mathcal{R}') \rightsquigarrow \llbracket \Sigma \rrbracket <: \llbracket \Sigma' \rrbracket \Rightarrow \lambda x. \llbracket f(x.\text{mod}) \rrbracket}$ <p style="margin: 0;">E-SUB-MODTYPE</p> $\frac{\Delta, \bar{\alpha} \vdash^{\text{elab}} \mathcal{R} <: \mathcal{R}' \rightsquigarrow \Sigma <: \Sigma' \Rightarrow f \quad \Delta, \bar{\alpha} \vdash^{\text{elab}} \mathcal{R}' <: \mathcal{R} \rightsquigarrow \Sigma' <: \Sigma \Rightarrow f'}{\Delta \vdash_A^{\text{elab}} (\text{module type } T = \lambda \bar{\alpha}. \mathcal{R}) <: (\text{module type } T = \lambda \bar{\alpha}. \mathcal{R}') \rightsquigarrow \llbracket \lambda \bar{\alpha}. \Sigma \rrbracket <: \llbracket \lambda \bar{\alpha}. \Sigma' \rrbracket \Rightarrow \lambda x. \llbracket \lambda \bar{\alpha}. \Sigma' \rrbracket}$

F.2 Typing

F.2.1 $\Delta \vdash^{\text{elab}} Q.t : \tau$ – Type checking

$$\begin{array}{c}
\text{E-TYP-TYPE-LOCALTYPE} \\
\frac{A.t : (\text{type } t = \tau, \text{type } t = \tau) \in \Delta}{\Delta \vdash^{\text{elab}} A.t : \tau} \\
\\
\text{E-TYP-TYPE-PATHTYPE} \\
\frac{\Delta \vdash^{\text{elab}} P : \text{sig}_A \bar{D} \text{ end} \rightsquigarrow _ : \Sigma \quad (\text{type } t = \tau) \in \bar{D} \quad \Sigma.l_t : \langle\langle \tau \rangle\rangle}{\Delta \vdash^{\text{elab}} P.t : \tau}
\end{array}$$

F.2.2 $\Delta \vdash_A^{\text{elab}} D : \lambda \bar{\alpha}. \mathcal{D} \rightsquigarrow \lambda \bar{\alpha}. \zeta$ – Declaration typing

$$\begin{array}{c}
\text{E-TYP-DECL-VAL} \qquad \qquad \qquad \text{E-TYP-DECL-TYPE} \\
\frac{\Delta \vdash^{\text{elab}} \tau : \tau' \quad A.x \notin \Delta}{\Delta \vdash_A^{\text{elab}} \text{val } x : \tau : \text{val } x : \tau' \rightsquigarrow \text{val } x : \tau'} \qquad \frac{\Delta \vdash^{\text{elab}} \tau : \tau' \quad A.t \notin \Delta}{\Delta \vdash_A^{\text{elab}} \text{type } t = \tau : \text{type } t = \tau' \rightsquigarrow \text{type } t = \tau'} \\
\\
\text{E-TYP-DECL-TYPEABS} \\
\frac{A.t \notin \Delta}{\Delta \vdash_A^{\text{elab}} \text{type } t = A.t : \lambda \alpha. \text{type } t = \alpha \rightsquigarrow \lambda \alpha. \text{type } t = \alpha} \\
\\
\text{E-TYP-DECL-MOD} \\
\frac{\Delta \vdash^{\text{elab}} S : \lambda \bar{\alpha}. \mathcal{R} \rightsquigarrow \lambda \bar{\alpha}. \Sigma \quad A.X \notin \Delta}{\Delta \vdash_A^{\text{elab}} (\text{module } X : S) : \lambda \bar{\alpha}. (\text{module } X : \mathcal{R}) \rightsquigarrow \lambda \bar{\alpha}. \text{module } X : \Sigma} \\
\\
\text{E-TYP-DECL-MODTYPE} \\
\frac{\Delta \vdash^{\text{elab}} S : \lambda \bar{\alpha}. \mathcal{R} \rightsquigarrow \lambda \bar{\alpha}. \Sigma \quad A.T \notin \Delta}{\Delta \vdash_A^{\text{elab}} (\text{module type } T = S) : (\text{module type } T = \lambda \bar{\alpha}. \mathcal{R}) \rightsquigarrow (\text{module type } T = \lambda \bar{\alpha}. \Sigma)} \\
\\
\text{E-TYP-DECL-EMPTY} \\
\Delta \vdash_A^{\text{elab}} \emptyset : \emptyset \rightsquigarrow \emptyset \\
\\
\text{E-TYP-DECL-SEQ} \\
\frac{\Delta \vdash_A^{\text{elab}} D_1 : \lambda \bar{\alpha}_1. \mathcal{D}_1 \rightsquigarrow \lambda \bar{\alpha}_1. \zeta_1 \quad \Delta, \bar{\alpha}_1, A.I_1 : (D_1, \zeta_1) \vdash_A^{\text{elab}} \bar{D} : \lambda \bar{\alpha}. \bar{D} \rightsquigarrow \lambda \bar{\alpha}. \bar{\zeta}}{\Delta \vdash_A^{\text{elab}} (D_1, \bar{D}) : \lambda \bar{\alpha}_1 \bar{\alpha}. (D_1, \bar{D}) \rightsquigarrow \lambda \bar{\alpha}_1 \bar{\alpha}. (\zeta_1, \bar{\zeta})}
\end{array}$$

F.2.3 $\Delta \vdash^{\text{elab}} S : \lambda \bar{\alpha}. \mathcal{R} \rightsquigarrow \lambda \bar{\alpha}. \Sigma$ – Signature typing

$$\begin{array}{c}
\text{E-TYP-SIG-SIG} \\
\frac{\Delta \vdash_A^{\text{elab}} \bar{D} : \lambda\bar{\alpha}.\bar{D} \rightsquigarrow \lambda\bar{\alpha}.\bar{\zeta} \quad A \notin \Delta}{\Delta \vdash^{\text{elab}} \text{sig}_A \bar{D} \text{ end} : \lambda\bar{\alpha}.\text{sig}_A \bar{D} \text{ end} \rightsquigarrow \lambda\bar{\alpha}.\{\bar{\zeta}\}} \\
\text{E-CF-MODTYPE} \\
\frac{\Delta \vdash^{\text{elab}} P : \text{sig}_A \bar{D} \text{ end} \rightsquigarrow _ : \Sigma' \quad \text{module type } T = \lambda\bar{\alpha}.\mathcal{R} \in \bar{D} \quad \Sigma'.\ell_T : \langle\langle \lambda\bar{\alpha}.\Sigma \rangle\rangle}{\Delta \vdash^{\text{elab}} A.T : \lambda\bar{\alpha}.\mathcal{R} \rightsquigarrow \lambda\bar{\alpha}.\Sigma} \\
\text{E-TYP-SIG-LOCALMODTYPE} \\
\frac{A.I : (\text{module type } T = \lambda\bar{\alpha}.\mathcal{R}, \text{module type } T = \lambda\bar{\alpha}.\Sigma) \in \Delta}{\Delta \vdash^{\text{elab}} A.T : \lambda\bar{\alpha}.\mathcal{R} \rightsquigarrow \lambda\bar{\alpha}.\Sigma} \\
\text{E-TYP-SIG-FUNCTOR} \\
\frac{\Delta \vdash^{\text{elab}} S_a : \lambda\bar{\alpha}.\mathcal{R}_a \rightsquigarrow \lambda\bar{\alpha}.\Sigma_a \quad \Delta, \bar{\alpha}, Y : (\mathcal{R}_a, \Sigma_a) \vdash^{\text{elab}} S : \lambda\bar{\beta}.\mathcal{R} \rightsquigarrow \lambda\bar{\beta}.\Sigma \quad Y \notin \Delta}{\Delta \vdash^{\text{elab}} (Y : S_a) \rightarrow S : \forall\bar{\alpha}.(Y : \mathcal{R}_a) \rightarrow \exists\bar{\beta}.\mathcal{R} \rightsquigarrow \forall\bar{\alpha}.\Sigma_a \rightarrow \exists\bar{\beta}.\Sigma}
\end{array}$$

F.2.4 $\Delta \vdash^{\text{elab}} M : \mathcal{S} \rightsquigarrow e : \Pi$ – Module typing

$\frac{\text{E-TYP-MOD-FCTARG} \quad Y : (\mathcal{R}, \Sigma) \in \Delta}{\Delta \vdash^{\text{elab}} Y : \mathcal{R} \rightsquigarrow Y : \Sigma}$	$\frac{\text{E-TYP-MOD-LOCAL} \quad A.X : (\mathcal{R}, \Sigma) \in \Delta}{\Delta \vdash^{\text{elab}} A.X : \mathcal{R} \rightsquigarrow A.X : \Sigma}$
$\frac{\text{E-TYP-STRUCT} \quad \Delta \vdash_A^{\text{elab}} B : \exists \bar{\alpha}. \bar{\mathcal{D}} \rightsquigarrow e : \exists \bar{\alpha}. \{\bar{\zeta}\} \quad A \notin \Delta}{\Delta \vdash^{\text{elab}} \text{struct}_A B \text{ end} : \exists \bar{\alpha}. \text{sig}_A \bar{\mathcal{D}} \text{ end} \rightsquigarrow e : \exists \bar{\alpha}. \{\bar{\zeta}\}}$	
$\text{E-TYP-MOD-SEALING} \quad \Delta \vdash^{\text{elab}} S : \lambda \bar{\alpha}. \mathcal{R} \rightsquigarrow \lambda \bar{\alpha}. \Sigma$ $\frac{\Delta \vdash^{\text{elab}} P : \mathcal{R}' \rightsquigarrow e : \Sigma' \quad \Delta \vdash^{\text{elab}} \mathcal{R}' <: \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow \Sigma' <: \Sigma[\bar{\alpha} \mapsto \bar{\tau}] \Rightarrow f}{\Delta \vdash^{\text{elab}} (P : S) : \exists \bar{\alpha}. \mathcal{R} \rightsquigarrow \text{pack}(\bar{\tau}, (f e)) : \lambda \bar{\alpha}. \Sigma}$	
$\text{E-TYP-MOD-FUNCTOR} \quad \Delta \vdash^{\text{elab}} S : \lambda \bar{\alpha}. \mathcal{R} \rightsquigarrow \lambda \bar{\alpha}. \Sigma \quad \Delta, \bar{\alpha}, Y : (\mathcal{R}, \Sigma) \vdash^{\text{elab}} M : \mathcal{S} \rightsquigarrow e : \Pi$ $\Delta \vdash^{\text{elab}} (Y : \mathcal{S}) \rightarrow M : \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S} \rightsquigarrow \forall \bar{\alpha}. \lambda(Y : \mathcal{R}). e : \forall \bar{\alpha}. \Sigma \rightarrow \Pi$	
$\text{E-TYP-MOD-APP} \quad \Delta \vdash^{\text{elab}} P : \forall \bar{\alpha}. (Y : \mathcal{R}) \rightarrow \mathcal{S} \rightsquigarrow e_P : \forall \bar{\alpha}. \Sigma \rightarrow \mathcal{S}$ $\frac{\Delta \vdash^{\text{elab}} P' : \mathcal{R}' \rightsquigarrow e : \Sigma' \quad \Delta \vdash^{\text{elab}} \mathcal{R}' <: \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow \Sigma' <: \Sigma[\bar{\alpha} \mapsto \bar{\tau}] \Rightarrow f}{\Delta \vdash^{\text{elab}} P(P') : \mathcal{R}[\bar{\alpha} \mapsto \bar{\tau}] \rightsquigarrow e_P \bar{\tau} (f e) : \Pi[\bar{\alpha} \mapsto \bar{\tau}]}$	
$\text{E-TYP-MOD-PROJ} \quad \Delta \vdash^{\text{elab}} M : \exists \bar{\alpha}. \text{sig}_A \bar{\mathcal{D}}_1, \text{module } X : \mathcal{R}, \bar{\mathcal{D}}_2 \text{ end} \rightsquigarrow e : \{\bar{\zeta}_1, \ell_X : \Sigma, \bar{\zeta}_2\}$ $\Delta \vdash^{\text{elab}} M.X : \exists \bar{\alpha}. \mathcal{R} \rightsquigarrow \text{unpack}(\bar{\alpha}, y) = e \text{ in } \text{pack}(\bar{\alpha}, e.\ell_X) : \exists \bar{\alpha}. \Sigma$	

F.2.5 $\Delta \vdash_A^{\text{elab}} B : \exists \bar{\alpha}. \bar{\mathcal{D}} \rightsquigarrow e : \exists \bar{\alpha}. \{\bar{\ell}_I : \bar{\zeta}\}$ – Bindings typing

<p>E-TYP-DECL-LET</p> $\frac{\Delta \vdash^{elab} e : \tau \rightsquigarrow e' : \tau' \quad \Delta \vdash^{elab} \tau : \tau' \quad A.x \notin \Delta}{\Delta \vdash_A^{elab} (\text{let } x = e) : (\text{val } x : \tau') \rightsquigarrow \{\ell_x = e'\} : (\text{val } x : \tau')}$
<p>E-TYP-DECL-TYPE</p> $\frac{\Delta \vdash^{elab} \tau : \tau' \quad A.t \notin \Delta}{\Delta \vdash_A^{elab} (\text{type } t = \tau) : (\text{type } t = \tau') \rightsquigarrow \{\ell_t = \langle\langle \tau' \rangle\rangle\} : (\text{type } t = \tau')}$
<p>E-TYP-DECL-MOD</p> $\frac{\Delta \vdash^{elab} M : \exists \bar{\alpha}. \mathcal{R} \rightsquigarrow e : \exists \bar{\alpha}. \Sigma \quad A.X \notin \Delta}{\Delta \vdash_A^{elab} (\text{module } X = M) : (\exists \bar{\alpha}. \text{module } X : \mathcal{R}) \rightsquigarrow \text{unpack}\langle \bar{\alpha}, y \rangle = e \text{ in } \text{pack}\langle \bar{\alpha}, \{\ell_X = y\} \rangle : \exists \bar{\alpha}. \text{module } X : \Sigma}$
<p>E-TYP-DECL-MODTYPE</p> $\frac{\Delta \vdash^{elab} S : \lambda \bar{\alpha}. \mathcal{R} \rightsquigarrow \lambda \bar{\alpha}. \Sigma \quad A.T \notin \Delta}{\Delta \vdash_A^{elab} (\text{module type } T = S) : (\text{module type } T = \lambda \bar{\alpha}. \mathcal{R}) \rightsquigarrow \{\ell_T = \langle\langle \lambda \bar{\alpha}. \mathcal{R} \rangle\rangle\} : (\text{module type } T = \lambda \bar{\alpha}. \Sigma)}$
<p>E-TYP-DECL-SEQ</p> $\frac{\Delta \vdash_A^{elab} B_1 : \exists \bar{\alpha}_1. \mathcal{D}_1 \rightsquigarrow e_1 : \exists \bar{\alpha}_1. \{\zeta_1\} \quad \Delta, \bar{\alpha}_1, A.I_1 : (\mathcal{D}_1, \zeta_1) \vdash_A^{elab} \bar{B} : \exists \bar{\alpha}. \bar{\mathcal{D}} \rightsquigarrow e : \exists \bar{\alpha}. \{\bar{\zeta}\}}{\Delta \vdash_A^{elab} B_1, \bar{B} : \exists \bar{\alpha}_1 \bar{\alpha}. \bar{\mathcal{D}}_1, \bar{\mathcal{D}} \rightsquigarrow \text{unpack}\langle \bar{\alpha}_1, x_1 \rangle = e_1 \text{ in } \bar{B} : \exists \bar{\alpha}_1 \bar{\alpha}. \{\zeta_1, \bar{\zeta}\}}$ $\text{unpack}\langle \bar{\alpha}, x \rangle = (\text{let } A.I_1 = x_1.l_{I_1} \text{ in } e) \text{ in}$ $\text{pack}\langle \bar{\alpha}_1 \bar{\alpha}, \{\ell_I = x_1.l_I, \bar{\ell}_I = x.l_I\} \rangle$

Analyse de dépendance vérifiée pour un langage synchrone à flot de données

Timothy Bourke^{1,2}, Basile Pesin^{1,2}, Marc Pouzet^{2,1}

¹ Inria Paris, France

² Département d'informatique de l'École normale supérieure, CNRS, PSL University, Paris, France

Résumé

Vélus est une formalisation d'un langage synchrone à flots de données et de sa compilation dans l'assistant de preuve Coq. Il inclut une définition de la sémantique dynamique du langage, un compilateur produisant du code impératif, et une preuve de bout en bout que le compilateur préserve la sémantique des programmes.

Dans cet article, on spécifie dans Vélus la sémantique de deux structures d'activation présentes dans les compilateurs modernes : `switch` et déclarations locales. Ces nouvelles constructions nécessitent une adaptation de l'analyse statique de dépendance de Vélus, qui produit un graphe acyclique comme témoin de la bonne formation d'un programme. On utilise ce témoin pour construire un schéma d'induction propre aux programmes bien formés. Ce schéma permet de démontrer le déterminisme du modèle sémantique dans Coq.

1 Introduction

Le projet Vélus [3, 4, 5] formalise un langage synchrone à flot de données dans l'assistant de preuve Coq [11]. Ce langage est un sous-ensemble du langage industriel Scade 6 [9], qui est utilisé pour la spécification de systèmes de contrôle embarqués critiques. Scade 6 s'appuie lui-même sur des langages de recherche tels que Lustre [6] et Lucid Synchrone [16]. Vélus intègre un compilateur produisant du code impératif Clight, langage d'entrée du compilateur vérifié CompCert [13], ainsi qu'une sémantique à flots de données pour son langage d'entrée, et une preuve de bout en bout mettant en relation cette sémantique et celle du code assembleur généré.

Dans cet article, on ajoute à Vélus deux nouvelles structures d'activation présentes dans Lucid Synchrone [7] et Scade 6 [9] : le `switch`, et le bloc de déclarations locales. La sémantique à flots de données du langage source de Vélus est étendue pour traiter ces constructions. De précédents travaux se sont concentrés sur la preuve de correction du schéma de compilation de Vélus. On a étendu ces preuves pour traiter le `switch` et les déclarations locales. Dans cet article, on s'intéresse plutôt à la preuve de propriétés dynamiques du modèle sémantique, comme l'adéquation des flots calculés aux définitions statiques d'un programme (typage des valeurs et échantillonnage), ou son déterminisme : pour une entrée donnée, une seule sortie est possible. Ces propriétés dépendent de la bonne formation du programme et en particulier de l'absence de cycle de dépendance dans ses définitions. L'encodage de cette notion de dépendance dans Coq est rendue plus complexe par l'ajout de structures d'activation.

Un exemple : contrôle des portes d'une télécabine On présente un programme Lustre contrôlant l'ouverture et la fermeture de la porte d'une télécabine hypothétique. Un moteur pas à pas bipolaire est utilisé pour déplacer les portes le long de leurs rails. Le tableau de la [figure 1](#) indique le signal à appliquer à chacun des pôles du moteur pour le faire tourner dans le sens horaire. Ce comportement peut être implémenté dans Vélus à l'aide de deux équations utilisant chacune l'opérateur de délai initialisé `fbv`. Pour faire tourner le moteur dans le sens anti-horaire, il faut inverser cette séquence d'activation. Le nœud `control_moteur` gère ce contrôle de bas

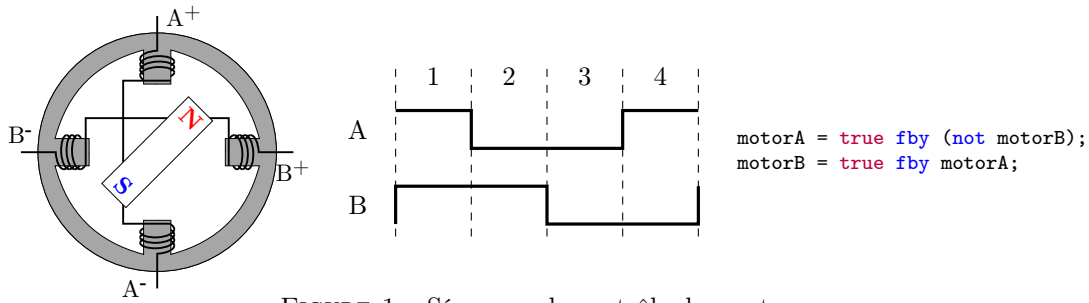


FIGURE 1 – Séquence de contrôle des moteurs

```

type moteurDir = Ouvre | Ferme | Bloque

node control_moteur(dir : moteurDir) returns (moteurA, moteurB : bool)
let
  switch dir
  | Ouvre do
    moteurA = true fby (not moteurB);
    moteurB = true fby moteurA;
  | Ferme do
    moteurA = true fby moteurB;
    moteurB = true fby (not moteurA);
  | Bloque do
    (moteurA, moteurB) = (false, false);
  end
end
tel

```

FIGURE 2 – Contrôle des moteurs

niveau. La structure d'activation `switch` active le bon comportement en fonction du paramètre `dir`, dans le sens horaire pour `Ouvre`, anti-horaire pour `Ferme`, ou stoppé pour `Bloque`.

Ce nœud est instancié par le nœud `control_porte`, présenté en figure 3, qui choisit la direction des moteurs. Quand la cabine est en dehors de la station, les portes sont fermées. La station est équipée de 8 repères fixés au rail. Quand la cabine passe un repère, l'entrée `repere` vaut `true`. Les portes doivent s'ouvrir entre les repères 1 et 3, rester ouvertes jusqu'au repère 6, puis se fermer avant le repère 8. Le compteur `nbRepere` est déclaré localement à la branche `true` du `switch`. Il est incrémenté chaque fois qu'un repère est détecté, jusqu'à 8. Pour le mettre à jour, on utilise la construction `last`, introduite dans [8], qui permet d'accéder à la valeur précédente d'un flot, même calculée dans l'autre branche du `switch` interne. La déclaration `last` doit être accompagnée d'une expression d'initialisation, pour donner une valeur à `last nbRepere` au premier instant.

Un fabricant de capteurs (hypothétique également) propose d'ajouter une redondance permettant de vérifier si les portes peuvent être ouvertes sans danger. Pour cela, il faut que la cabine soit proche du sol et que sa vitesse verticale soit suffisamment faible. Un capteur infrarouge placé sous la cabine permet de mesurer la distance avec le sol. Le manque de visibilité (neige, brouillard) peut parfois empêcher le capteur de fonctionner correctement pendant le trajet de la cabine. Pour pallier à ce problème, un accéléromètre permet de mesurer l'accélération verticale de la cabine. Cette mesure est alors intégrée pour calculer la vitesse courante, puis la position verticale. L'inconvénient de ce deuxième capteur est que de petites erreurs de mesure peuvent faire dériver l'estimation. Le nœud `ouvrir_ok` combine donc les données de ces capteurs pour vérifier que la porte peut être ouverte sans danger. Si le capteur infrarouge n'est pas

```

node control_porte(en_station : bool; repere : bool) returns (moteurA, moteurB : bool)
var dir : moteurDir;
let
  switch en_station
  | true do
    var last nbRepere : int = 0;
    let
      switch repere
      | true do nbRepere = (last nbRepere + 1) mod 8;
      | false do nbRepere = last nbRepere
      end;
      dir = if nbRepere > 0 and nbRepere < 3 then Ouvre
            else if nbRepere > 5 then Ferme
            else Bloque;
    tel
  | false do dir = Bloque;
  end;
  (moteurA, moteurB) = control_moteur(dir);
tel

```

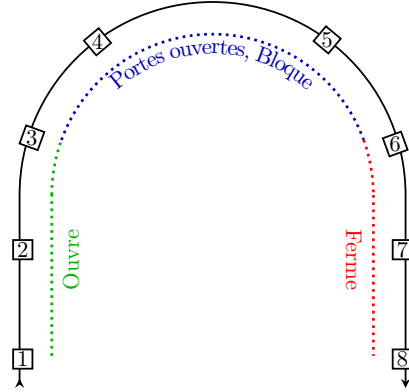


FIGURE 3 – Contrôle des portes

disponible, la valeur de `captdist` est négative et on utilise alors la valeur de l'accéléromètre `accely`. On note que les dépendances de ce programme commutent : dans la branche `true`, `vy` dépend de `y`, alors que c'est le contraire dans la branche `false`.

```

node ouvrir_ok(y_max, vy_max, captdist, accely, dt : double) returns (ok : bool)
var last vy : double = 0.0; last y : double = 0.0;
let
  switch (captdist > 0.0)
  | true do
    y = captdist;
    vy = (y - last y) / dt;
  | false do
    vy = last vy + accely * dt;
    y = last y + vy * dt;
  end;
  ok = y < y_max and vy < vy_max;
tel

```

FIGURE 4 – Vérification de la position et vitesse de la cabine

La suite présente une formalisation de la sémantique du `switch` et des déclarations locales dans Vélus. La section 3 adapte l'analyse de dépendance introduite dans [5, §2.3], et détaille son implémentation en Coq. On s'intéresse particulièrement au traitement des déclarations locales et de la commutation de dépendance. En section 4, on détaille le schéma d'induction introduit dans [5, §2.3], qu'on applique pour prouver que la sémantique de Vélus est déterministe.

La documentation du développement Coq décrit dans cet article, ainsi qu'une démonstration de l'exemple présenté ci-dessus sont accessibles à <https://velus.inria.fr/jfla2023>.

2 Sémantique à flots de données de Vélus

La syntaxe abstraite du langage d'entrée de Vélus est présentée en figure 5. Le langage des expressions étend celui présenté en [5]. Une expression peut faire référence à une constante primitive `c` ou énumérée `C`, un flot nommé avec `x`, et à sa valeur précédente avec `last x`. Les

opérateurs unaires et binaires arithmétiques et logiques habituels sont disponibles. Le **fbv** introduit un délai initialisé par les expressions à sa gauche. La flèche d'initialisation \rightarrow permet de donner une valeur particulière à l'instant initial. Le **when** échantillonne son premier argument selon la valeur d'un flot énuméré, et le **merge** fusionne des flots échantillonnés complémentaires. L'opérateur **case** est une généralisation de **if-then-else** pour une condition énumérée. Pour ces cinq dernières constructions, chaque opérande peut contenir une liste (non-vide) de sous-expressions (notée e^+); cela permet à une expression de définir plusieurs flots. Une expression peut enfin appeler un autre nœud du programme. Le compilateur supporte aussi le **reset** modulaire, traité formellement dans [4].

Dans [5], un bloc peut seulement définir une équation. On ajoute les structures d'activation **switch**, et les déclarations locales. Chaque déclaration est annotée par son type et son horloge statique. On y ajoute, pendant l'élaboration du programme, une étiquette unique pour chaque flot, ou deux pour un flot défini avec **last**. Par ailleurs, chaque branche d'un **switch** est annotée par une fonction de renommage de ces étiquettes σ . On verra plus loin comment ces étiquettes sont utilisées lors de l'analyse de dépendance d'un programme. Chaque nœud contient un bloc et spécifie ses entrées et sorties. Un programme est constitué d'une liste de définitions de types énumérés, suivie d'une liste non vide de nœuds.

$ \begin{array}{l} e ::= c \\ C \\ x \\ \text{last } x \\ \diamond e \quad \quad e \oplus e \\ e^+ \text{fbv } e^+ \\ e^+ \rightarrow e^+ \\ e^+ \text{when } (x = C) \\ \text{merge } x (C \Rightarrow e^+)^+ \\ \text{case } e \text{ of } (C \Rightarrow e^+)^+ \\ f(e^+) \\ (\text{reset } f \text{ every } e)(e^+) \end{array} $	$ \begin{array}{l} blk ::= x^+ = e^+ ; \\ \text{var } loc^* \text{ let } blk^+ \text{ tel} \\ \text{switch } e (C \text{ do}_\sigma blk^+)^+ \text{ end} \\ \\ d ::= x_{ty}^{ck} \\ \\ loc ::= d : \alpha_x \quad \quad \text{last } d : (\alpha_x, \alpha_{\text{last } x}) = e : \\ \\ td ::= \text{type } tx = C^+ \\ \\ n ::= \text{node } f ((d : \alpha_x)^+) \text{ returns } ((d : \alpha_x)^+) blk \\ \\ G ::= td^* n^+ \end{array} $
--	--

FIGURE 5 – Syntaxe abstraite de Lustre

2.1 Noyau du langage

On s'intéresse maintenant au modèle sémantique de Vélus, défini par un ensemble de règles, encodées comme des prédicats inductifs sur la syntaxe abstraite. Ce modèle contraint des flots infinis, représentés en Coq par le type co-inductif $\text{Stream } A := \text{Cons} : A \rightarrow \text{Stream } A \rightarrow \text{Stream } A$. Dans la suite, on note $x \cdot xs$ pour $\text{Cons } x \text{ } xs$. Ces flots étant infinis, on utilise la relation d'équivalence entre flots \equiv de la bibliothèque standard, dont la définition est rappelée ci-dessous.

Définition 1 (Equivalence de flots).

$$x \cdot xs \equiv y \cdot ys \quad \text{ssi} \quad x = y \quad \text{et} \quad xs \equiv ys$$

On rappelle d'abord le modèle sémantique de Vélus déjà présenté dans [5]. Ces règles encodent, sous forme de relation entre flots, la sémantique décrite dans [10]. On présente en **figure 6** les règles centrales de ce modèle. Le jugement $G, H, bs \vdash e \Downarrow vss$ se lit « sous l'environnement global G , l'historique H et l'horloge de base bs , l'expression e produit les flots vss ». Comme dit plus

haut, une expression peut produire plusieurs flots, et vss est donc une liste. Dans les règles suivantes, on note $[vs]$ quand cette liste ne contient qu'un élément. L'historique H associe un flot à chaque entrée, sortie et déclaration locale du nœud. Dans la suite, on note $x \in \text{dom}(H)$ le jugement « il y a un flot associé à x dans H ». Si c'est le cas, $H(x)$ est défini comme le flot associé à x dans H . La première règle de [figure 6](#) indique que, pour donner une sémantique à l'expression x , on lit le flot associé à x dans l'historique.

Un flot peut être échantillonné, c'est-à-dire que ses valeurs ne sont pas toujours présentes. Dans Vélus, on caractérise explicitement les absences (notées $\langle \rangle$) et présences (notées $\langle v \rangle$). Le « rythme » d'un flot échantillonné est caractérisé par un flot booléen, son horloge. L'horloge de base bs est celle des flots les plus rapides du nœud. La fonction `base-of` permet de la calculer : elle est vraie à un instant si et seulement si au moins une entrée du nœud est présente. Intuitivement, le nœud est actif à chaque fois qu'au moins l'une de ses entrées est présente. L'horloge bs est utilisée par la deuxième règle pour donner le rythme d'une constante avec la fonction `const`.

Le jugement $G, H, bs \vdash blk$ indique que la sémantique d'un bloc blk peut être donnée sous le contexte G, H, bs . Un bloc ne produit pas de valeur, mais induit un ensemble de contraintes sur l'historique H . La règle pour l'équation est rappelée en [figure 6](#) : les valeurs associées aux noms à gauche de l'équation doivent être celles calculées par les expressions à droite.

Enfin, l'environnement global G contient l'ensemble des nœuds du programme. Pour donner la sémantique d'un nœud de G , il faut donner un historique H respectant les contraintes des blocs du nœud. Les flots associés aux entrées et sorties du nœud sont lus dans H .

$$\begin{array}{c}
\frac{H(x) \equiv vs}{G, H, bs \vdash x \Downarrow [vs]} \quad \frac{}{G, H, bs \vdash c \Downarrow [\text{const } bs \ c]} \quad \text{const } (\mathbf{T} \cdot bs) \ c \equiv \langle c \rangle \cdot \text{const } bs \ c \\
\text{const } (\mathbf{F} \cdot bs) \ c \equiv \langle \rangle \cdot \text{const } bs \ c \\
\frac{G, H, bs \vdash es \Downarrow (vs_1, \dots, vs_n) \quad \forall i \in 1 \dots n, H(x_i) \equiv vs_i}{G, H, bs \vdash (x_1, \dots, x_n) = es} \\
\frac{\forall i \in 1 \dots n, H(x_i) \equiv xs_i \quad \forall i \in 1 \dots m, H(y_i) \equiv ys_i \quad G, H, (\text{base-of } (xs_1, \dots, xs_n)) \vdash blk}{G \vdash f(xs_1, \dots, xs_n) \Downarrow (ys_1, \dots, ys_m)} \quad G(f) = \text{node } f(x_1, \dots, x_n) \text{ returns } (y_1, \dots, y_m) \text{ blk}
\end{array}$$

FIGURE 6 – Règles sémantiques centrales [5]

La sémantique de chaque autre opérateur du langage d'expressions est exprimée au moyen d'une fonction de flots. On donne ici l'exemple de l'opérateur de délai initialisé `fby`, qui permet d'accéder à la valeur des flots aux instants précédents. Ce fonctionnement est encodé par les opérateurs `fby` et `fby1`, présentés en [figure 7](#). La première valeur présente du flot de gauche est produite par `fby`, tandis que les valeurs du deuxième flot sont conservées dans le premier argument de `fby1` jusqu'à la présence suivante. Pour donner une sémantique à l'expression `es0 fby es1`, on distribue l'opérateur `fby` sur les flots produits par les sous-expressions.

On note que l'opérateur `fby` est insensible aux absences, c'est à dire qu'on peut en insérer ou supprimer sans changer son comportement fondamental. Il prend une valeur du flot de gauche et l'ajoute devant le flot de droite. Cette idée s'étend aux nœuds, qui associent aux absences en entrée des absences en sortie. Les nœuds sont donc eux aussi insensibles aux absences.

$$\begin{array}{lcl}
\text{fby } (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) & \equiv & \langle \rangle \cdot \text{fby } xs \ ys \\
\text{fby } (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) & \equiv & \langle v_1 \rangle \cdot \text{fby1 } v_2 \ xs \ ys \\
\text{fby1 } v_0 (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) & \equiv & \langle \rangle \cdot \text{fby1 } v_0 \ xs \ ys \\
\text{fby1 } v_0 (\langle v_1 \rangle \cdot xs) (\langle v_2 \rangle \cdot ys) & \equiv & \langle v_0 \rangle \cdot \text{fby1 } v_2 \ xs \ ys
\end{array}
\quad
\begin{array}{l}
G, H, bs \vdash es_0 \Downarrow (xs_1, \dots, xs_n) \\
G, H, bs \vdash es_1 \Downarrow (ys_1, \dots, ys_n) \\
\forall i \in 1 \dots n, \text{fby } xs_i \ ys_i \equiv vs_i \\
\hline
G, H, bs \vdash es_0 \text{fby } es_1 \Downarrow (vs_1, \dots, vs_n)
\end{array}$$

FIGURE 7 – Sémantique de l'opérateur de délai initialisé **fby**

$$\begin{array}{lcl}
\frac{G, H, bs \vdash e \Downarrow [cs] \quad \forall i, G, \text{when}^{C_i}(H, bs) \ cs \vdash \text{blks}_i}{G, H, bs \vdash \text{switch } e (C_i \text{ do } \text{blks}_i)_i \text{ end}} & \text{when}^C (\langle \rangle \cdot xs) (\langle \rangle \cdot ys) & \equiv \langle \rangle \cdot \text{when}^C \ xs \ ys \\
& \text{when}^C (\langle v \rangle \cdot xs) (\langle C \rangle \cdot ys) & \equiv \langle v \rangle \cdot \text{when}^C \ xs \ ys \\
& \text{when}^C (\langle v \rangle \cdot xs) (\langle C' \rangle \cdot ys) & \equiv \langle \rangle \cdot \text{when}^C \ xs \ ys
\end{array}$$

FIGURE 8 – Sémantique du switch

2.2 Structure d'activation switch

Pour modéliser la sémantique du **switch**, on s'appuie sur l'idée d'insensibilité aux absences présentée ci-dessus. Une branche doit être insensible, dans le même sens que pour les **fby** et les nœuds, quand l'expression de garde ne correspond pas à l'étiquette de la branche. De plus, le **switch** doit combiner les flots produits par les branches, en choisissant à chaque instant les valeurs de la branche désignée par l'expression de garde, les valeurs de toutes les autres branches étant absentes. Dans les instants où l'expression de garde est absente, toutes les valeurs de toutes les branches sont absentes.

On concrétise ces intuitions par la règle du **switch** définie en **figure 8**, à gauche. L'opérateur **when**, appliqué à l'historique et à l'horloge de base, est au cœur de cette modélisation. Pour chaque branche, il est appliqué avec le flot de l'expression de garde cs et l'étiquette C_i de la branche. Ainsi, il rend la branche insensible quand le flot de garde ne correspond pas à l'étiquette, et ne reporte les valeurs produites par la branche que quand elles sont présentes.

La définition de **when** pour un flot est donnée en **figure 8**, à droite. C'est une fonction partielle : ses deux entrées doivent être présentes en même temps. Pour appliquer **when** à l'horloge de base, un flot booléen sans absence explicite, ce critère est relaxé, et on interprète les absences dans le flot produit par **when** comme le booléen **false**. Les historiques étant représentés comme des fonctions partielles, on peut définir **when** appliqué à un historique par $(\text{when}^C H \ cs)(x) = \text{when}^C (H(x)) \ cs$. Cela signifie que $(\text{when}^C H \ cs)(x)$ est défini si et seulement si le flot associé à x dans H a le même rythme que cs . Ainsi, la sémantique des sous-blocs du **switch** est donnée dans un environnement plus restreint que H . Cela correspond d'ailleurs à un critère de typage statique du langage, que nous ne détaillerons pas ici.

2.3 Déclarations locales

Comme on l'a vu dans les exemples, l'utilisateur peut arbitrairement imbriquer des déclarations locales, y compris dans la branche d'un **switch**. On introduit en **figure 10** une définition pour la sémantique d'un bloc de déclarations locales. La première prémisse de la règle spécifie le domaine d'un historique local H' qui associe un flot à chaque déclaration locale du bloc. Cet historique H' vient compléter l'historique global H pour donner une sémantique aux sous-blocs, comme indiqué dans la troisième prémisse. Cette formalisation est inspirée de celle présentée

dans [7, §3.2]. L'opérateur d'union d'historique $+$ est défini par

$$(H_1 + H_2)(x) = \begin{cases} H_2(x) & \text{si } x \in \text{dom}(H_2) \\ H_1(x) & \text{sinon.} \end{cases}$$

Le cas où $x \in \text{dom}(H_1)$ et $x \in \text{dom}(H_2)$ ne peut en fait pas se produire, puisque Vélus rejette statiquement le masquage d'une déclaration globale par une déclaration locale. Dans le cas contraire, on pourrait écrire le programme de la figure 9, où la variable x dont dépend l'horloge de y est masquée. Pour éviter ce problème, il serait nécessaire d'introduire un niveau d'indirection dans les types d'horloges [8, Fig. 3], ce qu'on n'a pas fait dans Vélus.

```
node f(x : bool; y : int when x) returns (z : int)
var x : int;
let
  x = y + 1;
  z = x;
tel
```

FIGURE 9 – Programme avec masquage, rejeté par Vélus

Variables partagées Les flots locaux peuvent être déclarés comme partagés en utilisant le mot clé `last`. Pour traiter cette fonctionnalité, la définition d'historique est enrichie pour associer un flot différent à x et à `last x`, comme dans [17]; on y accède respectivement avec $H(x)$ et $H(\text{last } x)$. Dans le développement Coq, l'historique est encodé par une paire d'environnements, l'un pour les variables courantes et l'autre pour les `last`. Les règles de sémantique pour `last` sont présentées en figure 10. La deuxième règle indique que l'expression `last x` produit le flot associé à `last x` dans l'historique. Cette association est contrainte au point de définition de x par la première règle. Le flot associé à `last x` correspond à celui associé à x , retardé par un délai initialisé par le flot de l'expression d'initialisation e .

$$\frac{\forall x, x \in \text{dom}(H') \iff x \in \text{locs} \quad \forall x e, (\text{last } x = e) \in \text{locs} \implies G, H + H', bs \vdash \text{last } x = e \quad G, H + H', bs \vdash \text{blks}}{G, H, bs \vdash \text{var locs let blks tel}}$$

$$\frac{H(\text{last } x) \equiv vs}{G, H, bs \vdash \text{last } x \Downarrow [vs]}$$

$$\frac{G, H + H', bs \vdash e \Downarrow [vs_0] \quad H'(x) \equiv vs_1 \quad H'(\text{last } x) \equiv \text{fby } vs_0 \text{ } vs_1}{G, H + H', bs \vdash \text{last } x = e}$$

FIGURE 10 – Sémantique des déclarations locales et `last`

3 Analyse de dépendance

Dans la section précédente, on a présenté la sémantique dynamique de Vélus. Cependant, ce modèle seul ne suffit pas à garantir les propriétés attendues du langage. Considérons l'équation

$x = x$. Un nœud la contenant ne serait pas déterministe, puisqu'elle n'applique aucune contrainte au flot associé à x , qui peut prendre n'importe quelle valeur. À l'inverse, l'équation $x = x + 1$ n'admet pas de sémantique, alors qu'elle est bien typée. De plus, de telles équations ne peuvent pas être compilées, puisque dans le code impératif produit, une valeur doit être calculée avant son utilisation. C'est aussi le cas pour les systèmes d'équations contenant un cycle, comme par exemple, $x = y + 1$; $y = x * 2$.

De nombreux travaux se sont penchés sur l'analyse statique de ces dépendances. Les analyses les plus fines [12, 1, 9] les modélisent sous forme de systèmes de types. Ces formulations permettent de donner à un nœud un type indiquant les dépendances entre entrées et sorties. Elles peuvent par exemple accepter l'équation $(x, y) = f(x)$ si la première sortie de f ne dépend pas instantanément de son entrée. Le schéma de compilation doit alors être adapté pour produire un programme ordonnançable, soit en expansant les appels de nœud [9], soit en les décomposant en plusieurs appels ordonnançables séparément [18, 14]. Le schéma de compilation de Vélus est bien plus simple : chaque nœud est une « boîte noire », et les appels sont atomiques [2]. L'exemple est donc rejeté. L'analyse de dépendance de Vélus introduite dans [5, §2.3] est basée sur une analyse de graphe. Dans cette section, on étend cette analyse pour traiter les nouvelles structures d'activation, et on décrit l'implémentation Coq de l'algorithme d'analyse de graphe.

3.1 Règles de dépendances pour le langage

A chaque flot x du programme est associé une étiquette α_x . Cette indirection permet de donner une étiquette différente à deux déclarations locales utilisant le même nom. La suite définit d'abord la fonction calculant l'ensemble des variables utilisées instantanément dans une expression avant de présenter formellement la relation de dépendance entre deux étiquettes.

3.1.1 Variables utilisées instantanément

Pour analyser les cycles de dépendance dans un programme, il faut d'abord déterminer l'ensemble de variables utilisées par une expression. Considérons l'équation $x = 0 \text{ fby } (x + 1)$. Elle ne doit pas induire de cycle de dépendance puisque x ne dépend que de sa valeur précédente. On ne s'intéressera donc qu'aux variables utilisées *instantanément* dans une expression, c'est-à-dire celles n'apparaissant pas à droite d'un **fby**.

Une fonction pour ce calcul est proposée dans [2, Fig. 3], mais ne s'applique que sur un programme normalisé. Notre analyse doit être un peu plus précise pour traiter le langage général. Par exemple, l'équation $(x, y) = \text{if } c \text{ then } (0, x) \text{ else } (1, 0)$ ne contient pas de cycle : y dépend de x , mais x ne dépend pas de lui-même. Effectivement, cette équation est normalisée [5] en deux équations $x = \text{if } c \text{ then } 0 \text{ else } 1$ et $y = \text{if } c \text{ then } x \text{ else } 0$, dans lesquelles l'absence de cycle est évidente. Pour traiter cette subtilité, on définit la fonction $\text{UsedInst}_\Gamma(e)[k]$ qui collecte les étiquettes des variables utilisées instantanément pour définir le k -ième flot produit par l'expression e . Dans cet exemple, $k = 0$ désigne la définition $x = \text{if } c \text{ then } 0 \text{ else } 1$, et $k = 1$ désigne la définition $y = \text{if } c \text{ then } x \text{ else } 0$. L'environnement Γ associe le nom d'un flot à son étiquette. Il est construit à partir des déclarations encodées dans la syntaxe abstraite.

On donne en [figure 11](#) quelques cas intéressants de la définition de cette fonction, qui est encodée comme un prédicat inductif comme dans le développement Coq. Pour une variable, on récupère l'annotation correspondante dans l'environnement. Pour une expression composée comme le **if-then-else**, on considère le flot de la condition, et les k -ième flots des deux alternatives. Pour traiter une liste de sous expressions, il faut choisir le k -ième élément de la liste aplatie des flots. Pour cela, on calcule le nombre de flots de chaque expression avec

$$\begin{array}{c}
\frac{\Gamma(x) = \alpha_x}{\alpha_x \in \text{UsedInst}_\Gamma(x)[0]} \quad \frac{\alpha \in \text{UsedInst}_\Gamma(e)[0] \quad k < \text{numstreams}(\text{if } e \text{ then } es_0 \text{ else } es_1)}{\alpha \in \text{UsedInst}_\Gamma(\text{if } e \text{ then } es_0 \text{ else } es_1)[k]} \\
\\
\frac{\alpha \in \text{UsedInst}_\Gamma(es_0)[k]}{\alpha \in \text{UsedInst}_\Gamma(\text{if } e \text{ then } es_0 \text{ else } es_1)[k]} \quad \frac{\alpha \in \text{UsedInst}_\Gamma(es_1)[k]}{\alpha \in \text{UsedInst}_\Gamma(\text{if } e \text{ then } es_0 \text{ else } es_1)[k]} \\
\\
\frac{\alpha \in \text{UsedInst}_\Gamma(e)[k]}{\alpha \in \text{UsedInst}_\Gamma(e :: es)[k]} \quad \frac{\alpha \in \text{UsedInst}_\Gamma(es)[k]}{\alpha \in \text{UsedInst}_\Gamma(e :: es)[\text{numstreams}(e) + k]} \\
\\
\frac{\alpha \in \text{UsedInst}_\Gamma(es_0)[k]}{\alpha \in \text{UsedInst}_\Gamma(es_0 \text{ fby } es_1)[k]} \quad \frac{\alpha \in \text{UsedInst}_\Gamma(es)[k'] \quad k < \text{numstreams}(f(es))}{\alpha \in \text{UsedInst}_\Gamma(f(es))[k]}
\end{array}$$

FIGURE 11 – Quelques règles pour `UsedInst`

`numstreams(e)`. Pour un `fbym`, on ne prends pas en compte les variables apparaissant à droite, comme expliqué plus tôt. Enfin, un appel de nœud est atomique : l'expression `f(es)` dépend de toutes les variables utilisées instantanément dans `es`.

3.1.2 Variables définies et dépendances

On souhaite maintenant établir les contraintes de dépendance induites par un bloc. On note $\Gamma \vdash blk \mid \alpha_2 \xleftarrow{dep} \alpha_1$ pour indiquer que sous l'environnement Γ , dans le bloc `blk`, α_2 dépend de α_1 . Les règles qui définissent ces contraintes dépendent aussi d'une fonction $\text{Def}_\Gamma(blk)$ qui calcule les étiquettes définies par un bloc `blk`.

Equations : Les étiquettes définies par une équation sont celles des variables à gauche de l'équation. Une équation induit immédiatement une dépendance entre l'étiquette du k -ième flot défini, et les variables utilisées instantanément dans le k -ième flot des expressions. Par exemple, dans $(x, y) = (0, x \text{ fby } z)$, x ne dépend d'aucune étiquette et y dépend seulement de x .

$$\frac{\Gamma(x_i) = \alpha_{x_i}}{\alpha_{x_i} \in \text{Def}_\Gamma((x_1, \dots, x_n) = es)} \quad \frac{\Gamma(x_i) = \alpha_{x_i} \quad \alpha \in \text{UsedInst}_\Gamma(es)[i]}{\Gamma \vdash (x_1, \dots, x_n) = es \mid \alpha_{x_i} \xleftarrow{dep} \alpha}$$

FIGURE 12 – Étiquettes définies et dépendances pour une équation

Déclarations locales : Les déclarations locales introduisent de nouvelles étiquettes `locs`. Pour traiter les sous-blocs d'une déclaration, il faut les prendre en compte, en étendant l'environnement avec $+$, défini comme : $\forall x, (\Gamma + locs)(x) = \alpha_x \leftrightarrow (\Gamma(x) = \alpha_x \vee locs(x) = \alpha_x)$. Les règles de calcul de `Def` et de dépendance utilisant cet opérateur sont données en [figure 13](#). On note que les étiquettes d'une déclaration locale échappent à leur portée. C'est nécessaire pour construire un graphe de dépendance global au nœud. Un invariant statique, non présenté ici, garantit par ailleurs que toutes les étiquettes du nœud sont distinctes.

$$\begin{array}{c}
\frac{\alpha \in \text{Def}_{(\Gamma+locs)}(blks)}{\alpha \in \text{Def}_{\Gamma}(\text{var } locs \text{ let } blks \text{ tel})} \qquad \frac{(\Gamma+locs) \vdash blks \mid \alpha_y \xleftarrow{dep} \alpha_x}{\Gamma \vdash \text{var } locs \text{ let } blks \text{ tel} \mid \alpha_y \xleftarrow{dep} \alpha_x} \\
\\
\frac{(\text{last } y : (\alpha_y, \alpha_{\text{last}y}) = e) \in locs \quad \alpha_x \in \text{UsedInst}_{(\Gamma+locs)}(e)[0]}{\Gamma \vdash \text{var } locs \text{ let } blks \text{ tel} \mid \alpha_{\text{last}y} \xleftarrow{dep} \alpha_x}
\end{array}$$

FIGURE 13 – Etiquettes définies et dépendances pour une déclaration locale

Last : Si l'utilisateur déclare un flot local x avec une valeur `last` initiale, on ajoute également une étiquette distincte pour `last` x . Cela permet de traiter `last` x comme un flot à part, qui ne dépend que des flots apparaissant dans son expression d'initialisation. Ainsi, les blocs présentés en [figure 14](#) sont valides (mais ne produisent pas le même flot pour x). En revanche, celui présenté en [figure 15](#) ne l'est pas, puisque x dépend immédiatement de `last` x et inversement.

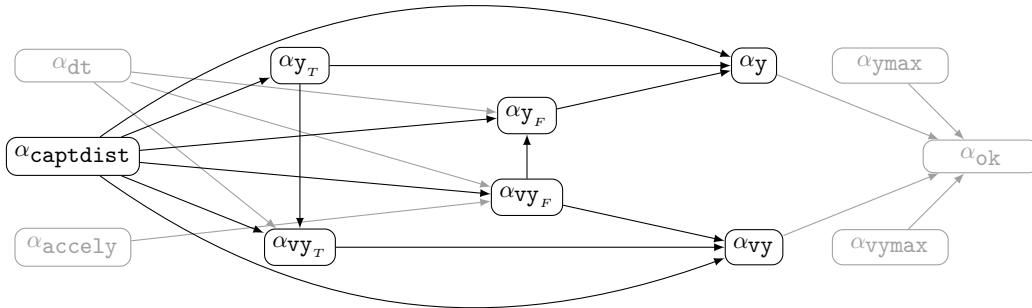
```
var last x = 0;
let x = last x + 1; tel
```

```
var last x = x + 1;
let x = 0; tel
```

```
var last x = x + 1;
let x = last x; tel
```

FIGURE 14 – Déclarations `last` correctesFIGURE 15 – Déclaration `last` cyclique

Switch-Case : Le cas du switch est plus complexe. On rappelle le nœud présenté en [figure 4](#). Il ne serait pas accepté si on associait les étiquettes α_y et α_{vy} à y et vy respectivement : il y aurait un cycle. On peut en revanche associer des étiquettes α_{y_T} et α_{y_F} aux définitions de y dans les branches `true` et `false`, et de même pour vy . Les deux branches étant mutuellement exclusives, il ne peut pas y avoir d'interaction entre α_{y_T} et α_{y_F} . L'étiquette α_y dépend par contre de ces étiquettes locales aux branches. De même, tous ces flots dépendent de la condition qui détermine quelle branche est activée, et donc de α_{captdist} . Le graphe de dépendance pour l'exemple est présenté en [figure 16](#). Il est bien acyclique.

FIGURE 16 – Graphe de dépendance pour la [figure 4](#)

Pour encoder ces changements d'associations, chaque branche d'un `switch` est annotée dans la syntaxe abstraite par une fonction σ qui s'applique à l'environnement pour modifier les étiquettes. On omet cette annotation dans la présentation des règles où elle n'est pas utile. Un invariant statique non présenté ici nous garantit que les étiquettes renommées par σ sont celles

des flots définis par le **switch**. On a

$$\sigma(\Gamma)(x) = \begin{cases} \sigma(x) & \text{si } x \in \sigma \\ \Gamma(x) & \text{sinon.} \end{cases}$$

La [figure 17](#) présente les règles de dépendances pour le **switch**. Les étiquettes définies par un **switch** sont celles des sous-blocs après renommage par σ . La deuxième règle y ajoute les étiquettes correspondantes avant le renommage. La fonction σ est aussi utilisée pour calculer les dépendances des sous-blocs. Toutes les étiquettes des flots définis par le **switch** dépendent de la condition. La dernière règle ajoute une dépendance entre chaque étiquette interne α_x et l'étiquette correspondante externe α_y , pour « connecter » les flots définis.

Cet étiquetage reflète le schéma de compilation présenté dans [8]. Une nouvelle variable est introduite pour chaque variable définie et chaque branche, et un **merge** combine les valeurs de ces nouvelles variables pour définir la variable originale.

$$\frac{\alpha \in \text{Def}_{\sigma_i(\Gamma)}(blks_i)}{\alpha \in \text{Def}_{\Gamma}(\text{switch } e (C_i \text{ do}_{\sigma_i} blks_i)_i \text{ end})} \qquad \frac{x \in \sigma_i \quad \Gamma(x) = \alpha}{\alpha \in \text{Def}_{\Gamma}(\text{switch } e (C_i \text{ do}_{\sigma_i} blks_i)_i \text{ end})}$$

$$\frac{\sigma_i(\Gamma) \vdash blks_i \mid \alpha_1 \xleftarrow{dep} \alpha_2}{\Gamma \vdash \text{switch } e (C_i \text{ do}_{\sigma_i} blks_i)_i \text{ end} \mid \alpha_1 \xleftarrow{dep} \alpha_2}$$

$$\frac{\alpha_x \in \text{UsedInst}_{\Gamma}(e)[0] \quad \alpha_y \in \text{Def}_{\Gamma}(\text{switch } e (C_i \text{ do } blks_i)_i \text{ end})}{\Gamma \vdash \text{switch } e (C_i \text{ do } blks_i)_i \text{ end} \mid \alpha_y \xleftarrow{dep} \alpha_x}$$

$$\frac{x \in \sigma_i \quad \Gamma(x) = \alpha_y \quad \sigma_i(\Gamma)(x) = \alpha_x}{\Gamma \vdash \text{switch } e (C_i \text{ do}_{\sigma_i} blks_i)_i \text{ end} \mid \alpha_y \xleftarrow{dep} \alpha_x}$$

FIGURE 17 – Etiquettes définies et dépendances pour un **switch**

3.2 Analyse du graphe de dépendance

Les relations de dépendance décrites plus haut nous permettent de construire le graphe de dépendance d'un programme. On exprime ce graphe en Coq par une table associant chaque étiquette à la liste de ses prédécesseurs. Cette représentation ne garantit bien sûr pas l'absence de cycles dans le graphe. C'est le prédicat inductif **AcyGraph**, introduit dans [5], qui caractérise les graphes acycliques. Il est défini par les trois règles rappelées dans la [figure 18](#). Ce prédicat est paramétré par un ensemble de sommets et un ensemble d'arcs. Un graphe vide est acyclique. Ajouter un sommet préserve l'acyclicité. Cependant, on ne peut ajouter un arc entre deux sommets α_x et α_y que s'ils sont distincts, et qu'il n'existe pas d'arc retour dans la clôture transitive des arcs existants.

On veut maintenant développer une fonction vérifiant qu'un graphe de dépendance construit selon les règles de la [section 3.1](#) satisfait le prédicat d'acyclicité. On s'appuie sur un algorithme bien connu de recherche en profondeur pour traverser les prédécesseurs de chaque sommet dans le graphe. Pour justifier de sa terminaison et de sa correction, on utilise les types dépendants de Coq pour encoder les invariants de l'algorithme. On utilise l'extension **Program** [19] qui permet

$$\frac{}{\text{AcyGraph } \emptyset \emptyset} \quad \frac{\text{AcyGraph } VA}{\text{AcyGraph } (V \cup \{\alpha\})A} \quad \frac{\text{AcyGraph } VA \quad \alpha_x, \alpha_y \in V \quad \alpha_x \neq \alpha_y \quad \alpha_y \not\rightarrow_A^+ \alpha_x}{\text{AcyGraph } V(A \cup \{\alpha_x \rightarrow \alpha_y\})}$$

FIGURE 18 – Graphe acyclique par construction

de définir la partie calculatoire de l’algorithme en Gallina, le langage de programmation à la ML de Coq, et de traiter séparément les obligations de preuve avec le langage de tactiques Ltac.

On s’intéresse d’abord à la partie calculatoire, présentée [figure 19](#), en ignorant les types dépendants utilisés pour les entrées et sorties. L’appel `dfs' s x v` recherche récursivement les prédécesseurs de `x` dans le `graph`. Elle renvoie l’ensemble des sommets visités à l’issue de son exécution. L’ensemble `v` représente les sommets déjà visités. L’ensemble `s` contient les sommets sur le chemin parcouru depuis le sommet initial. Ces ensembles sont représentés par le type `PS.t`. On utilise la monade `option` pour représenter la réussite ou l’échec de l’algorithme. La fonction échoue (ligne 13) si elle rencontre un sommet `x ∈ s`, ce qui signifie qu’il y a un cycle dans le graphe. Sinon, si `x ∈ v`, c’est qu’on connaît déjà tous ses prédécesseurs, donc on peut renvoyer `v` (ligne 16). Si `x` reste à traiter, on récupère la liste de ses prédécesseurs (ligne 18), et on appelle récursivement `dfs'` sur chacun, après avoir ajouté `x` à `s` (ligne 21). Pour itérer sur la liste des prédécesseurs, on utilise `ofold_left : (B → A → option B) → list A → option B → option B`, qui se comporte comme `fold_left`, mais s’interrompt si la fonction retourne une erreur (ligne 22).

Aux lignes 12 et 15, on utilise la construction `match/with`, plutôt qu’un `if/then/else` pour traiter la condition booléenne. C’est une contrainte liée à l’utilisation de `Program` qui, si on utilise `if/then/else`, ne conserve pas assez d’information dans les obligations de preuves générées.

Terminaison : La fonction `dfs'` ne respecte pas le critère de récursion gardée de Coq où chaque appel récursif doit être effectué sur un sous terme de l’argument initial. On utilise `Program Fixpoint` pour définir une fonction récursive en utilisant une « mesure », c’est-à-dire une fonction des arguments vers un entier naturel. Si cette mesure décroît strictement à chaque appel récursif, la fonction termine, puisque (\mathbb{N}, \leq) est bien fondé. Pour cet algorithme, on sait que l’ensemble `s` grandit à chaque appel récursif, et qu’il est inclus dans l’ensemble des sommets du graphe. La mesure `num_remaining`, définie ligne 3 décroît donc strictement.

Correction : L’algorithme est correct si sa réussite indique qu’un témoin d’`AcyGraph` peut être associé au graphe analysé. Il est difficile de raisonner à posteriori sur un programme écrit avec `Program Fixpoint`, car le programme Coq généré contient des termes complexes justifiant de sa terminaison. À la place, on intègre les invariants de l’algorithme dans le type de la fonction.

L’invariant `visited s v` garantit d’abord que l’ensemble `s` des sommets dans le parcours et l’ensemble `v` des sommets visités sont bien disjoints (ligne 6). Par ailleurs, il indique qu’il existe un ensemble d’arcs `a`, tel que (1) le graphe formé par les sommets `v` et les arcs `a` est acyclique, et (2) pour chaque sommet `x` de `v`, tout arc allant vers `x` dans le `graph` analysé est présent dans `a`, ce qui est caractérisé par `has_arc`. Intuitivement, cela indique que le sous-graphe de `graph` réduit aux sommets de `v` est effectivement acyclique. On note que l’ensemble d’arcs `a` n’a pas besoin d’être calculé explicitement. Un tel existentiel disparaîtra dans le code OCaml extrait et ne coûtera donc pas de temps ni de mémoire à l’exécution. Par ailleurs, le type de `dfs'` garantit que le sommet `x` recherché est bien dans l’ensemble `v'` renvoyé, qui est un sur-ensemble de celui passé en entrée à la fonction.

Itérer `dfs` sur l’ensemble des sommets du graphe permet donc de construire un témoin `AcyGraph` pour l’ensemble des sommets du graphe. A chaque nœud du programme source analysé,

```

1 Variable graph : Env.t (list label).
2
3 Definition num_remaining (s : PS.t) : nat := Env.cardinal graph - PS.cardinal s.
4
5 Definition visited (s : PS.t) (v : PS.t) : Prop :=
6   (∀ x, PS.In x s → ¬PS.In x v)
7   ∧ ∃a, AcyGraph v a ∧ (∀ x, PS.In x v → ∃zs, Env.find x graph = Some zs ∧ (∀ y, In y zs → has_arc a y x)).
8
9 Program Fixpoint dfs' (s : { p | ∀x, PS.In x p → Env.In x graph }) (x : ident)
10   (v : { v | visited s v }) {measure (num_remaining s)}
11   : option { v' | visited s v' & PS.In x v' ∧ PS.Subset v v' } :=
12   match PS.mem x s with
13   | true ⇒ None
14   | false ⇒
15     match PS.mem x v with
16     | true ⇒ Some (exist2 _ _ v _)
17     | false ⇒
18       match Env.find x graph with
19       | None ⇒ None
20       | Some zs ⇒
21         let s' := exist _ (PS.add x s) _ in
22         match ofold_left (fun v w ⇒ obind (dfs' s' w v) (fun v' ⇒ Some (exist _ v' _))) zs (Some v) with
23         | None ⇒ None
24         | Some v' ⇒ Some (exist2 _ _ (PS.add x v') _ _)
25         end
26       end
27     end
28   end.
29
30 Definition dfs (x : ident) (v : { v | visited PS.empty v })
31   : option { v' | visited PS.empty v' & (PS.In x v' ∧ PS.Subset v v') } :=
32   dfs' (exist _ PS.empty _)

```

FIGURE 19 – Algorithme de recherche en profondeur certifiant

on peut donc associer un graphe dans lequel chaque dépendance entre étiquettes du nœud est reflétée par un arc. Si le graphe est acyclique, on dit que le nœud est causal, voir [définition 2](#).

Définition 2 (Nœud « causal »).

$$\text{node_causal } n := \exists VA, \text{AcyGraph } VA \wedge (\forall \alpha_x \alpha_y, \vdash n \mid \alpha_y \xleftarrow{\text{dep}} \alpha_x \implies \alpha_x \rightarrow_A \alpha_y)$$

4 Preuves par induction causale

Des travaux antérieurs [5, §2.3] présentent une approche permettant de construire un schéma d'induction pour les nœuds causaux. Dans la suite de cette section, on rappelle ce schéma avec plus de détails. On propose également un autre schéma d'induction basé sur UsedInst. On applique ensuite ces schémas pour prouver le déterminisme du modèle sémantique de Vélus.

4.1 Principe d'induction sur les étiquettes

Etant donné un graphe acyclique caractérisé par $\text{AcyGraph } VA$, on peut extraire un ordre topologique des étiquettes, sur lequel on peut aisément raisonner par induction. La [figure 20](#) présente les règles de construction de cet ordre. La liste vide représente un ordre valide. Pour

ajouter un sommet à la liste, il faut que (1) ce soit bien un sommet du graphe, (2) qu'il n'apparaisse pas dans la liste, et (3) que tous les prédécesseurs, transitivement, de ce sommet soient déjà dans la liste. Pour ce dernier critère, on a prouvé que ne prendre que les arcs immédiats ($\alpha_y \rightarrow_A \alpha_x$) donne une formulation équivalente. Cela dit, on préfère utiliser la formulation de la [figure 20](#), qui est plus simple à manipuler dans nos preuves car elle correspond mieux à celle d'AcyGraph. En particulier, on établit l'existence d'un ordre topologique sur l'ensemble des sommets de tout graphe acyclique, comme énoncé dans la [figure 20](#).

$$\frac{}{\text{TopoOrder (AcyGraph } VA) \square} \quad \frac{\text{TopoOrder (AcyGraph } VA) \text{ lord} \quad \alpha_x \in V \quad \neg \text{In } \alpha_x \text{ lord} \quad (\forall \alpha_y, \alpha_y \rightarrow_A^+ \alpha_x \implies \text{In } \alpha_y \text{ lord})}{\text{TopoOrder (AcyGraph } VA) (\alpha_x :: \text{lord})}$$

$$\text{AcyGraph } VA \implies \exists \text{lord}, (\forall \alpha_x, \alpha_x \in V \iff \text{In } \alpha_x \text{ lord}) \wedge \text{TopoOrder (AcyGraph } VA) \text{ lord}$$

FIGURE 20 – Définition et existence d'un ordre topologique

Le principe d'induction est présenté en [lemme 1](#). On pose $P_var : \text{étiquette} \rightarrow \text{Prop}$ le prédicat à prouver pour chaque étiquette d'un nœud. La fonction `locals` collecte l'ensemble des étiquettes apparaissant dans les déclarations locales du bloc, y compris celles associées aux `last`. Le nœud doit être causal. Pour toute étiquette α_x , si toutes les étiquettes dont α_x dépend respectent P_var , alors α_x doit respecter P_var . Si ces deux conditions sont respectées, alors on sait que toutes les étiquettes apparaissant dans les entrées, sorties et déclarations locales du nœud respectent P_var .

Lemme 1 (Induction sur les étiquettes d'un nœud causal).

$$\begin{aligned} & \text{si } \text{node_causal } (\text{node } (ins) \text{ returns } (outs) \text{ blk}) \\ & \text{et } (\forall \alpha_x, \text{In } \alpha_x (ins + outs + locals \text{ blk}) \implies \\ & \quad (\forall \alpha_y, (ins + outs) \vdash \text{blk} \mid \alpha_x \xleftarrow{dep} \alpha_y \implies P_var \alpha_y) \implies \\ & \quad P_var \alpha_x) \\ & \text{alors } (\forall \alpha_x, \text{In } \alpha_x (ins + outs + locals \text{ blk}) \implies P_var \alpha_x) \end{aligned}$$

La preuve de ce lemme utilise `node_causal` pour faire apparaître le graphe acyclique g associé au nœud. On utilise le mécanisme d'induction habituel de Coq sur le témoin de `TopoOrder` associé à g pour prouver que P_var est vraie, de proche en proche, pour chaque étiquette de l'ordre topologique. Pour une dépendance $\alpha_x \xleftarrow{dep} \alpha_y$ dans le nœud, il y a un arc de α_y vers α_x dans le graphe, donc α_y apparaît avant α_x dans l'ordre topologique, et donc, par induction, $P_var \alpha_y$ est vrai.

4.2 Principe d'induction sur les expressions

L'élément syntaxique central induisant une dépendance est l'équation. Considérons l'équation $(x, y) = \text{es}$. Le flot associé à x est le premier flot associé à es . Pour établir une propriété P_var sur ce flot, il faut d'abord raisonner par induction sur les expressions es . C'est l'objectif du deuxième schéma d'induction que l'on présente maintenant. On pose $P_exp : \text{exp} \rightarrow \text{nat} \rightarrow \text{Prop}$,

un prédicat sur le k -ième flot produit par une expression, et P_exps , un prédicat équivalent sur une liste d'expressions. Si toutes les variables utilisées instantanément du k -ième flot de l'expression e satisfont P_var , alors $P_exp\ e\ k$ doit être satisfaite. Le schéma d'induction correspondant est présenté en [lemme 2](#). Il suit la définition de `UsedInst` de la [section 3.1.1](#).

Lemme 2 (Induction sur `UsedInst`).

si $P_var\ \alpha_x \implies P_exp\ x\ 0$
et $\forall k, P_exp\ e\ 0 \wedge P_exps\ es_0\ k \wedge P_exps\ es_1\ k \implies P_exp\ (\text{if } e \text{ then } es_0 \text{ else } es_1)\ k$
et $\forall k, P_exps\ es_0\ k \implies P_exp\ (es_0 \text{ fby } es_1)\ k$
et ...
alors $\forall e\ k, (\forall x, x \in \text{UsedInst}_\Gamma(e)[k] \implies P_var\ x) \implies P_exp\ e\ k$

Pour chaque forme d'expression, l'utilisateur du schéma doit prouver que, si un ensemble d'hypothèses d'induction est vérifié, alors P_exp est vraie pour l'expression. Par exemple, dans le cas d'une variable, l'utilisateur doit prouver $P_exp\ x\ 0$, sous l'hypothèse que l'étiquette associée à x respecte P_var . Pour le `if-then-else`, on a une hypothèse pour la condition, et une pour chacun des deux opérandes. Le cas du `fby` est plus intéressant : conformément à la définition de `UsedInst`, on n'a qu'une hypothèse d'induction pour l'opérande de gauche. Quelque soit la propriété P_exp considérée, elle doit donc pouvoir être prouvée indépendamment de l'opérande de droite du `fby`. C'est le cas dans la preuve de déterminisme.

4.3 Application : déterminisme de la sémantique

On s'intéresse maintenant à la propriété de déterminisme du modèle sémantique, qui peut être exprimée comme « l'exécution d'un nœud pour une entrée donnée produit toujours la même sortie ». Cette propriété est formalisée par le [théorème 1](#).

Théorème 1 (Déterminisme de la sémantique).

si $\text{node_causal}\ G(f)$
et $G \vdash f(xs) \Downarrow ys_1$ *et* $G \vdash f(xs) \Downarrow ys_2$
alors $ys_1 \equiv ys_2$

Ce théorème fait l'hypothèse de deux exécutions possibles du nœud. Cela signifie qu'il existe deux historiques distincts H_1 et H_2 qui satisfont les contraintes de ce nœud. Le but est de prouver qu'en réalité, ces historiques correspondent, pour chaque variable du nœud. Ce n'est vrai que pour les nœuds causaux. En effet, dans un nœud contenant la déclaration cyclique $x = x$, on peut associer n'importe quel flot à x .

Pour illustrer la structure générale de la preuve, on s'appuie sur le programme `after_n` présenté en [figure 21](#). Ce programme renvoie `false` pendant n instants, puis `true`. Après cela, le compteur interne c n'est plus mis à jour pour éviter un dépassement d'entier négatif.

Dans ce programme, les valeurs de c et b à l'instant $n + 1$ dépendent de leurs valeurs à l'instant n . Pour prouver que les flots associés à c et b dans H_1 et H_2 correspondent, on doit donc raisonner par induction sur n . L'induction doit être globale au nœud puisque la définition d'une variable peut dépendre de la valeur précédente de n'importe quelle autre : dans l'exemple, c dépend de la valeur précédente de b . On utilise la relation d'équivalence jusqu'à n définie ci-dessous. Si on peut prouver $xs_1 \equiv_n xs_2$ pour un n arbitraire, alors on peut prouver $xs_1 \equiv xs_2$.

```

node after_n(n : int) returns (b : bool)
var c : int;
let
  c = n fby (if b then c else (c - 1));
  b = c <= 0;
tel

```

FIGURE 21 – Programme `after_n`

Définition 3 (Equivalence jusqu'à n).

$$xs \equiv_0 ys \qquad x \cdot xs \equiv_{n+1} y \cdot ys \quad \text{ssi} \quad x = y \quad \text{et} \quad xs \equiv_n ys$$

Le cas $n = 0$ de l'induction est trivial. Pour le cas inductif, on fait l'hypothèse que tous les flots de H_1 et H_2 sont égaux jusqu'à n , et on veut prouver qu'ils le sont toujours jusqu'à $n + 1$. La propriété qu'on veut prouver pour chaque étiquette `P_var` dépend donc du rang n , et des deux historiques H_1 et H_2 . Elle dépend aussi d'un environnement Γ pour faire le lien entre les noms des flots et leurs étiquettes. Cet environnement est déduit des déclarations des entrées, sorties, et variables locales du nœud. On définit donc

$$\text{P_var } \alpha_x := \forall x, \Gamma(x) = \alpha_x \implies H_1(x) \equiv_{n+1} H_2(x)$$

La preuve procède par induction sur l'ordre topologique des variables, comme décrit dans le [lemme 1](#). Dans l'exemple, `P_var` est immédiatement vraie pour `n` qui est une entrée. On la prouve ensuite pour `c` qui ne dépend instantanément que de `n`, puis pour `b` qui dépend de `c`.

Induction sur les expressions Pour chaque étiquette, il y a dans le nœud une équation $(x_1, \dots, x_m) = \text{es}$ définissant le flot correspondant. Par la règle sémantique de l'équation, on sait que le flot associé à l'étiquette de x_k est le k -ième flot produit par `es`. On montre donc le lemme ci-dessous qui indique que les k -ième flots produits correspondent jusqu'à $n + 1$. Pour le prouver, on utilise le principe d'induction du [lemme 2](#).

Lemme 3 (Equivalence jusqu'à $n + 1$ pour les expressions).

$$\begin{aligned}
& \text{si } \forall x, H_1(x) \equiv_n H_2(x) \\
& \text{et } \forall x \alpha_x, \alpha_x \in \text{UsedInst}_\Gamma(e)[k] \implies \Gamma(x) = \alpha_x \implies H_1(x) \equiv_{n+1} H_2(x) \\
& \text{et } bs_1 \equiv_{n+1} bs_2 \\
& \text{et } G, H_1, bs_1 \vdash e \Downarrow \mathbf{vss}_1 \quad \text{et } G, H_2, bs_2 \vdash e \Downarrow \mathbf{vss}_2 \\
& \text{alors } \mathbf{vss}_1[k] \equiv_{n+1} \mathbf{vss}_2[k]
\end{aligned}$$

Pour prouver ce résultat, il faut que les flots de H_1 et H_2 soient égaux jusqu'à n , ce qu'on sait par hypothèse d'induction. Il faut aussi que les flots associés aux variables utilisées instantanément pour le k -ième flot soient égaux jusqu'à $n + 1$. Cette différence vient du traitement de `fby`. Considérons l'équation `c = n fby (if b then c else (c - 1))` de l'exemple. Pour prouver que les flots associés à `c` sont égaux jusqu'à $n + 1$, il faut prouver que les flots associés à `n` le sont, puisque `c` en dépend instantanément. En revanche, on a seulement besoin de savoir que les flots associés aux variables à droites du `fby` sont égaux jusqu'à n , puisque leurs valeurs à l'instant $n + 1$ ne sont pas utilisées instantanément. Cette intuition est formalisée dans [lemme 4](#), qui est prouvé par co-induction sur les définitions de `fby` et `fby1`.

Lemme 4 (Equivalence jusqu'à $n + 1$ pour `fby`).

$$\text{si } xs_1 \equiv_{n+1} xs_2 \quad \text{et } ys_1 \equiv_n ys_2 \quad \text{alors } \text{fby } xs_1 \ ys_1 \equiv_{n+1} \text{fby } xs_2 \ ys_2$$

Induction sur la sémantique des déclarations locales Pour trouver l'équation définissant un flot, on procède par induction sur l'arbre de syntaxe des blocs. On rencontre une difficulté pour les flots déclarés localement, comme `c` dans l'exemple. En effet, la règle pour les déclarations locales de la [figure 10](#) indique seulement qu'un historique interne existe et associe un flot à chaque déclaration interne, mais n'expose pas ces flots. Ainsi, la propriété `P_var` globale au nœud n'est pas suffisante pour traiter ces étiquettes locales.

Pour surmonter cette complication, on définit une version instrumentée de la sémantique des blocs. Elle met en parallèle les deux exécutions du bloc, en ajoutant localement des informations sur la correspondance des flots associés aux déclarations locales. Ces informations sont préservées pendant les inductions imbriquées sur n et sur l'ordre topologique.

$$\begin{array}{c}
\forall x, x \in \text{dom}(H'_1) \iff x \in \text{locs} \quad \forall x, x \in \text{dom}(H'_2) \iff x \in \text{locs} \\
\forall x e, (\text{last } x = e) \in \text{locs} \implies G, H_1 + H'_1, bs \vdash \text{last } x = e \\
\forall x e, (\text{last } x = e) \in \text{locs} \implies G, H_2 + H'_2, bs \vdash \text{last } x = e \\
G, H_1 + H'_1, H_2 + H'_2, bs_1, bs_2 \vdash_{n, \text{lord}} \text{blks} \quad \forall x, x \in \text{locs} \implies H'_1(x) \equiv_n H'_2(x) \\
\forall x, \text{locs}(x) = \alpha \implies \alpha \in \text{lord} \implies H'_1(x) \equiv_{n+1} H'_2(x) \\
\hline
G, H_1, H_2, bs_1, bs_2 \vdash_{n, \text{lord}} \text{var locs let blks tel}
\end{array}$$

FIGURE 22 – Sémantique instrumentée, cas de la déclaration locale (cf. [figure 10](#))

On note le jugement instrumenté $G, H_1, H_2, bs_1, bs_2, \vdash_{n, \text{lord}} \text{blk}$. La règle pour les déclarations locales est donnée en [figure 22](#). Elle suit la règle de sémantique donnée plus tôt pour les déclarations locales. Les deux dernières prémisses établissent la correspondance entre les deux exécutions. La prémisses $\forall x, x \in \text{locs} \implies H'_1(x) \equiv_n H'_2(x)$ impose que tous les flots locaux soient égaux jusqu'à n . Cela correspond à notre invariant d'induction sur n . De plus, la prémisses $\forall x, \text{locs}(x) = \alpha \implies \alpha \in \text{lord} \implies H'_1(x) \equiv_{n+1} H'_2(x)$ impose que les flots associés aux étiquettes de `lord` soient égaux jusqu'à $n+1$. Cette liste `lord` est un préfixe de l'ordre topologique manipulé par le principe d'induction du [lemme 1](#). Cela correspond donc à notre invariant d'induction sur l'ordre topologique.

La structure de la preuve du [théorème 1](#), en prenant en compte le modèle sémantique instrumenté, est la suivante : On procède par induction sur n . Par hypothèse d'induction, on sait que tous les flots du nœud sont égaux au rang n . On sait donc en particulier que tous les flots locaux le sont, mais on ne sait pas encore qu'ils sont égaux jusqu'à $n+1$. On pose `lord` = `[]`, la liste (vide) des étiquettes pour lesquelles on sait que les flots sont égaux jusqu'à $n+1$, et on a donc $G, H_1, H_2, bs_1, bs_2, \vdash_{n, []} \text{blk}$. On utilise alors le principe d'induction du [lemme 1](#) pour prouver que les flots associés à chaque étiquette du nœud sont égaux jusqu'à $n+1$. Pour chaque étiquette α , on trouve l'équation définissant le flot correspondant et on utilise le [lemme 3](#) comme décrit plus haut. Si l'étiquette est celle d'une déclaration locale, on ajoute α à la liste `lord`, et on obtient $G, H_1, H_2, bs_1, bs_2, \vdash_{n, \alpha::\text{lord}} \text{blk}$. À la fin de l'induction, on a $G, H_1, H_2, bs_1, bs_2, \vdash_{n, (\text{locals } \text{blk})} \text{blk}$. Autrement dit, on sait que tous les flots locaux sont égaux jusqu'à $n+1$, parce qu'ils sont tous dans la liste. C'est équivalent à incrémenter n , et vider la liste, comme énoncé formellement dans le [lemme 5](#), ce qui termine la preuve.

Lemme 5 (Sémantique instrumentée, passage de n à $n+1$).

$$\text{si } G, H_1, H_2, bs_1, bs_2, \vdash_{n, (\text{locals } \text{blk})} \text{blk} \quad \text{alors } G, H_1, H_2, bs_1, bs_2 \vdash_{n+1, []} \text{blk}$$

5 Conclusion

Dans cet article, nous avons décrit l’ajout de deux nouvelles structures d’activation dans Vélus : le `switch` et les déclarations locales. Nous avons aussi ajouté à Vélus des variables partagées au moyen de l’opérateur `last`. Nous avons modélisé la sémantique à flot de données de ces structures. Les nouvelles règles sémantiques s’intègrent modulairement au modèle de Vélus, c’est-à-dire sans modifier les règles existantes. Nous avons aussi adapté l’analyse de dépendance de Vélus pour traiter ces nouvelles constructions. Nous avons présenté en détail un algorithme d’analyse de graphe générant un témoin de l’absence de cycle dans le graphe de dépendance d’un nœud. Enfin, nous avons défini un principe d’induction tirant parti de cette analyse.

Travaux connexes Les auteurs de [7] proposent une sémantique par réaction pour un langage synchrone à flots de données avec structures d’activation. Dans [8], la sémantique des structures d’activation est donnée « par traduction » dans le langage noyau. De la même manière, notre règle pour le `switch` est définie en fonction de l’opérateur d’échantillonnage `when`, qui est aussi utilisé lors de sa compilation. Ces articles formalisent également la sémantique et la compilation de machines à états hiérarchiques dans un langage synchrone à flots de données. Nous travaillons actuellement à l’ajout de cette construction au compilateur Vélus. Du point de vue de l’analyse de dépendance, elle ne pose a priori pas plus de difficultés que les `switch`. Sa sémantique est en revanche plus difficile à exprimer dans notre modèle relationnel à flots de données.

Notre implémentation de l’algorithme de recherche en profondeur a suivi, sans le savoir, le travail de F. Pottier. Il note [15, §4] que son algorithme, comme le notre, utilise sur la pile d’appel un espace proportionnel à la profondeur maximale de recherche, et qu’une implémentation en style récursif terminal avec pile explicite éviterait ce problème. Dans notre cas les graphes manipulés sont issus de programmes écrits par l’utilisateur et il y a donc peu de chance d’arriver à un débordement de la pile. Comme dans ce travail on utilise des types dépendants pour justifier la terminaison de l’algorithme, mais contrairement à lui on utilise la bibliothèque `Program` de M. Sozeau [19] pour éviter de programmer avec des tactiques.

Discussion Dans notre développement Coq, les règles sémantiques de Vélus présentées dans cet article sont encodées par des prédicats inductifs. Manipuler cet encodage dans un script de preuve Coq est commode, puisqu’on peut raisonner par introduction, inversion et induction. Entre autres, la preuve de correction des algorithmes de compilation pour ces structures, qui n’a pas été présentée dans cet article, est relativement directe.

Le principe d’induction sur un nœud causal présenté dans cet article est une idée centrale pour le langage. Si, dans cet article, nous ne l’avons appliqué qu’à la preuve du déterminisme du modèle sémantique, il nous a été utile pour établir d’autres propriétés. Prouver la correction du système d’horloge, c’est-à-dire l’adéquation du rythme des flots produits aux annotations statiques d’horloges, nécessite par exemple l’utilisation de ce principe. Cette propriété est indispensable pour démontrer la correction du compilateur.

Remerciements Nous remercions L. Brun pour ses contributions à Vélus, et en particulier son travail sur l’intégration des types énumérés, sans lesquels l’ajout de `switch` n’aurait pas été possible. Nous remercions également nos rapporteurs pour leurs remarques qui ont permis d’améliorer le contenu de cet article. Ce travail a été soutenu par le projet ANR JCJC FidelR 19-CE25-0014-01 « Fidelity in Reactive Systems Design and Compilation ».

Bibliographie

- [1] Albert BENVENISTE, Timothy BOURKE, Benoit CAILLAUD, Bruno PAGANO et Marc POUZET : A type-based analysis of causality loops in hybrid modelers. *In Proceedings of the 17th International Conference on Hybrid Systems : Computation and Control (HSCC 2014)*, pages 71–82, Berlin, Allemagne, avril 2014. ACM Press.
- [2] Dariusz BIERNACKI, Jean-Louis COLAÇO, Gregoire HAMON et Marc POUZET : Clock-directed modular code generation for synchronous data-flow languages. *In Proceedings of the 9th ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, pages 121–130, Tucson, AZ, États-Unis, juin 2008. ACM Press.
- [3] Timothy BOURKE, Lélío BRUN, Pierre-Évariste DAGAND, Xavier LEROY, Marc POUZET et Lionel RIEG : A formally verified compiler for Lustre. *In Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 586–601, Barcelone, Espagne, juin 2017. ACM Press.
- [4] Timothy BOURKE, Lélío BRUN et Marc POUZET : Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proceedings of the of the ACM on Programming Languages*, 4(POPL):1–29, janvier 2020.
- [5] Timothy BOURKE, Paul JEANMAIRE, Basile PESIN et Marc POUZET : Verified Lustre normalization with node subsampling. *ACM Transactions on Embedded Computing Systems*, 20(5s):Article 98, octobre 2021.
- [6] Paul CASPI, Daniel PILAUD, Nicolas HALBWACHS et John A. PLAICE : LUSTRE : A declarative language for programming synchronous systems. *In Proceedings of the 14th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1987)*, pages 178–188, Munich, Allemagne, janvier 1987. ACM Press.
- [7] Jean-Louis COLAÇO, Grégoire HAMON et Marc POUZET : Mixing signals and modes in synchronous data-flow systems. *In Proceedings of the 6th ACM International Conference on Embedded Software (EMSOFT 2006)*, pages 73–82, Séoul, Corée du Sud, octobre 2006. ACM Press.
- [8] Jean-Louis COLAÇO, Bruno PAGANO et Marc POUZET : A conservative extension of synchronous data-flow with state machines. *In Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT 2005)*, pages 173–182, Jersey City, États-Unis, septembre 2005. ACM Press.
- [9] Jean-Louis COLAÇO, Bruno PAGANO et Marc POUZET : Scade 6 : A formal language for embedded critical software development. *In Proceedings of the 11th International Symposium on Theoretical Aspects of Software Engineering (TASE 2017)*, pages 4–15, Nice, France, septembre 2017. IEEE Computer Society.
- [10] Jean-Louis COLAÇO et Marc POUZET : Clocks as first class abstract types. *In Proceedings of the 3rd International Conference on Embedded Software (EMSOFT 2003)*, volume 2855 de *Lecture Notes in Electrical Engineering*, pages 134–155, Philadelphie, PA, États-Unis, octobre 2003. Springer.
- [11] COQ DEVELOPMENT TEAM : *The Coq proof assistant reference manual*. Inria, 2020.
- [12] Pascal CUOQ et Marc POUZET : Modular causality in a synchronous stream language. *In 10th European Symposium on Programming (ESOP 2001), part of European Joint Conferences on Theory and Practice of Software (ETAPS 2001)*, volume 2028 de *Lecture Notes in Electrical Engineering*, pages 237–251, Gênes, Italie, avril 2001. Springer.
- [13] Xavier LEROY : Formal verification of a realistic compiler. *Comm. ACM*, 52(7):107–115, 2009.
- [14] Roberto LUBLINERMAN, Christian SZEGEDY et Stavros TRIPAKIS : Modular code generation from synchronous block diagrams : Modularity vs. code size. *In Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2009)*, pages 78–89, Savannah, GA, États-Unis, janvier 2009. ACM Press.
- [15] François POTTIER : Depth-first search and strong connectivity in Coq. *In Journées Francophones des Langages Applicatifs (JFLA)*, janvier 2015.

- [16] Marc POUZET : *Lucid Sychrone, version 3. Tutorial and reference manual*. Université Paris-Sud, avril 2006.
- [17] Marc POUZET : ZRun. <https://github.com/marcpouzet/zrun/blob/master/src/coiteration.ml>, 2021.
- [18] Marc POUZET et Pascal RAYMOND : Modular static scheduling of synchronous data-flow networks : An efficient symbolic representation. *In Proceedings of the 9th ACM International Conference on Embedded Software (EMSOFT 2009)*, pages 215–224, Grenoble, France, octobre 2009. ACM Press.
- [19] Matthieu SOZEAU : Subset Coercions in Coq. *Lecture Notes in Computer Science*, (4502):237–252, 2007.

Formal proofs applied to system models.

Évelyne Contejean, Andrei Samokish

LMF – Université Paris-Saclay, CNRS, ENS Paris-Saclay

Abstract

Usually, the description of nuclear equipment by the FMEA (Failure Mode and Effects Analysis) method can be of considerable length (up to 5,000 lines); on the other hand, the number of rules used for the verification of this equipment is small. In addition, upstream, there is the question of trust in the tools that generate these descriptions for complex equipment, that is to say, made up of several thousand objects (requirements, functions, interfaces, behaviors).

Our objective is to formally prove in the Coq proof assistant the accuracy and exhaustiveness of the safety analysis for these nuclear equipments, by proposing a more modular and more abstract framework which is based on an axiomatic description of the systems. - these axioms being either accepted or proven independently of the verification itself. The very nature of the inductive types of Coq is perfectly suited to the description of complex systems, which are systems of systems in interaction. We have a first prototype that demonstrates the feasibility of this approach, for simple properties which are effectively proven by reflection - for example, the study of flows, which must first be produced and then consumed exactly once in the whole system.

We open the way to a general approach for the proofs of the properties of a (model of) systems and study the kind of properties that can be demonstrated in this way. Finally, we propose the approach for faithful translation of a DSL (domain-specific language) tool into Coq and the proof of its correctness.

1 Motivation

For about 15 years, the idea that systems can be modeled has been developing, and more and more processes, including safety and security analysis, have started to be backed by models. Such models can be used to perform simulations but may also reduce complexity and turn complex problems into smaller ones from a logical point of view. This is the case for Failure Mode and Effect Analysis (FMEA), where the risk analysis of a complex system is turned into an extensive list of risk reduction means for each couple of functions and components in the system. More generally, for complex systems, the safety case becomes a complicated document or set of information that transforms a high-level goal into a huge number of much simpler goals. An FMEA for a Nuclear Steam Generator (used as an example in this paper) is in the order of magnitude of thousands of lines.

The idea that software tools can help handle this complexity has been in the air for a long time. However, software tools are complex systems that can be challenged as well as any other complex software. When dealing with complex system models, the question of how confident we can be in the tools to handle all situations and faithfully describe the system of interest is at stake. In the software domain, Formal Methods (FM) are already used to verify that complex software performs correctly (see proof of C compiler by X. Leroy [10]), so it seems interesting

to investigate a similar approach for systems models and related tools.

We notice that the general definition of a system is inductive: broadly speaking, systems are made of (sub)systems in interactions, so when trying to prove some property on a system, it is logical to deduce that property from some properties of its subsystems. More generally, systems engineering is mainly built upon inductive breakdown structures: Relation Block Diagram (RBS), Functional Block Diagram (FBS), Project breakdown structure (PBS)... Classical safety methods, although inductive or deductive, are also based on the inductive design of systems. For this reason, we decided to investigate inductive types as a foundation for the formal definition of a system.

With this objective in mind, we should take care of two critical issues:

First, FMs are usually applied to detailed system behavior. Using FM methods seems very useful when the logic of the system is already defined. For example, formal Event-B specifications applied to FMEA give interesting results [5]. However, the later a problem is discovered, the more expensive the solution [11], so it is relevant to investigate the use of FM at the specification level, especially when high-level performance or non-functional requirements are refined into lower-level requirements. The development of Model-Based Systems Engineering provides an opportunity here because these tools and methods keep track of the refinement of the system design from the external analysis to the detailed component description. This objective is upstream w.r.t. the usual application of FM

Second, there can be significant drawbacks in FM application (a detailed description of problems can be found in [4]):

- FMs are difficult to learn and apply. Many assurance practitioners perceive the usage of FMs as negative.
- The number of practical (comparative) case studies using preliminary initial system design FMs is still too low to draw valuable and firm conclusions on FM effectiveness.
- FMs have been suffering from fragile effectiveness and productivity in the context where some physical processes are supposed to be controlled with a computer system.
- The effectiveness of formal models can be significantly reduced because of uncontrollable gaps between models and their implementations.

So, we need to provide an intermediate step between the FM-proof assistant and the system and safety model. The complexity of FM needs to be hidden from the end-user, and at the same time, it shall apply efficiently. For this reason, we propose a DSL (domain-specific language) as a front-end user interface that will be accessible to the engineer without knowledge of FM.

Our approach intends to make a bridge between system model description and formal systems, simplifying the dialog between engineers and mathematicians and allowing them to investigate system models with FM in the initial design stages.

2 Process and vocabulary

Fig. 1. Explains the different transformations that allow translating a system model faithfully into a Proof assistant program. In this section, we introduce both the process and the vocabulary corresponding to each translation step.

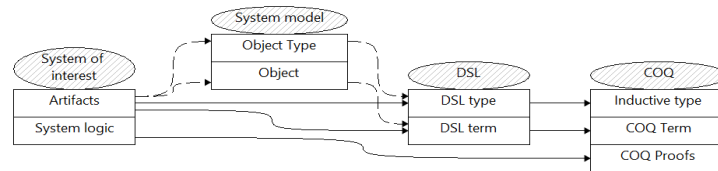


Fig. 1. Model formalization process

1. At the very beginning, Systems engineers describe systems through *System item*, *system interactions* (flows), *system artifacts* (used to describe, specify, and communicate about the system) (see INCOSE handbook [13]) - *artifacts* includes requirements, functions, flows, components, systems (in the meaning combination of functions and components). For systems, we may differentiate artifact categories or types from artifact instances. An artifact category may be ‘function’ while an artifact instance of “function” would be ‘function for heat transfer between primary and secondary circuit’.
2. Modeling the System of Interest is getting a standard way for specifying. During model formalization, engineers describe system *Artifacts* representing parts of the system of interest. Artifact instances are usually called objects, and Artifact categories are Object Types. Examples of model languages are UML, SysML, Domain Specific Languages (DSL)... Although some of these formalisms used to be partly diagram based, there is currently an effort to provide them with a complete syntactic definition.
3. Definition of Domain Specific Language (DSL) is a particular kind of model. At this step, we do use two main notions:
 - DSL types* model artifacts categories
 - DSL terms* model artifact instance and get a DSL type (i.e., DSL term ‘Steam Generator’ of *DSL type* ‘System’). In our process, we will consider a particular DSL that is simple enough to ease translation into a theorem prover and powerful enough to capture systems complexity. If we first define a model, we need to translate it into our DSL. The translation will associate to an object a DSL term and to an object type a DSL type.
4. The next step is the translation of the DSL into COQ. In COQ also, we can distinguish between terms and types. In the paper, we show it’s possible to translate the DSL types into a single Inductive Type that we will name COQ_DSL. COQ_DSL is, in fact, a mutually inductive type, and each mutually defined inductive type in COQ_DSL corresponds to a single DSL type. So it’s possible to make a one-to-one correspondence between DSL types and some COQ types. Once this is done, DSL terms can also be translated into COQ terms (see Fig.2). At this step, the System Model is defined as a set of artifacts with DSL types assigned and translated to COQ terms with appropriate inductive type constructors.



Fig. 2. Commutativity diagram between DSL and COQ terms

- Finally, we can specify and prove the properties of the system and its artifact. The main task at this step is to translate system and model properties and rules into COQ theorems and prove them.

3 Model example

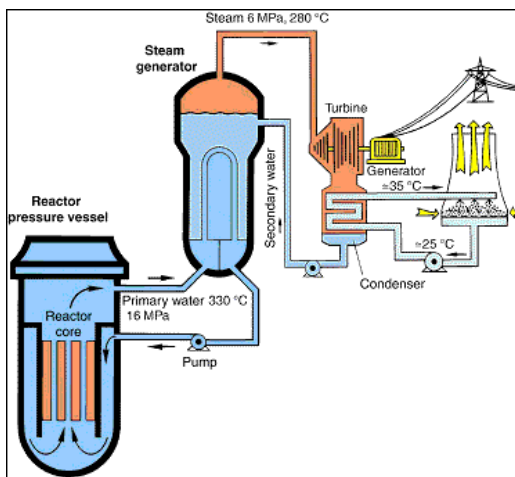


Fig. 3. Nuclear plant principle

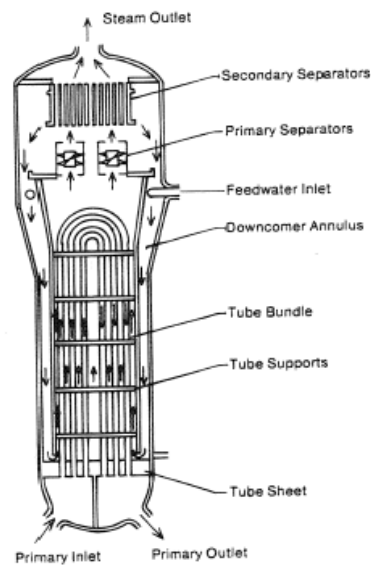


Fig. 4. Steam generator preliminary design

As example of a system model to be described, we took the general Nuclear Steam Generator (fig. 3,4) description represented in the AFCEN* guide [12]. Steam generators are heat exchangers used to make steam from water with the heat produced in a nuclear reactor core and are used in pressurized water reactors between the primary and secondary coolant loops. This is a good example of a system with a lot of safety requirements, a lot of physical relations between components, and some functional data flow between them. We will introduce DSL (Domain Specific Language) to describe the Generator model and then translate it into a formal description.

* AFCEN – French Association for Design, Construction and In-Service Inspection Rules for Nuclear Steam Supply System Components

In describing this example, we need to select the formal syntax for denoting system items and their hierarchical relationships and the formal syntax for denoting relations between system items. We can then introduce some inductive breakdown structure denoting the system as a whole. In our particular example, we introduce several Object Types like ‘System’, ‘Components’, ‘Function’, ‘Requirement’, and several categories for flows/relationships between these Objects Types: ‘Data flow’, ‘Physical flow’, and ‘Refine’. Some system logic terms are also defined: ‘System’ can contain objects of ‘Component’ only, ‘Component’ can contain ‘Component’ or ‘Function’, and so on... In fig.5, we provide a simplified model implemented with the given Object Types above. ‘Steam Generator Boiler’ of type ‘System’ is our system of interest; this object contains children object of type ‘Component’ exchanging flows of type ‘Physical flow’. The example presented is simplified; it has only several objects on the diagram. Our real example is based on a Use Case in the AFCEN guide [12] for risk analysis of nuclear pressurized equipment and consists of 66 Components, 153 Functions, 160 Physical flows, and 495 Requirements.

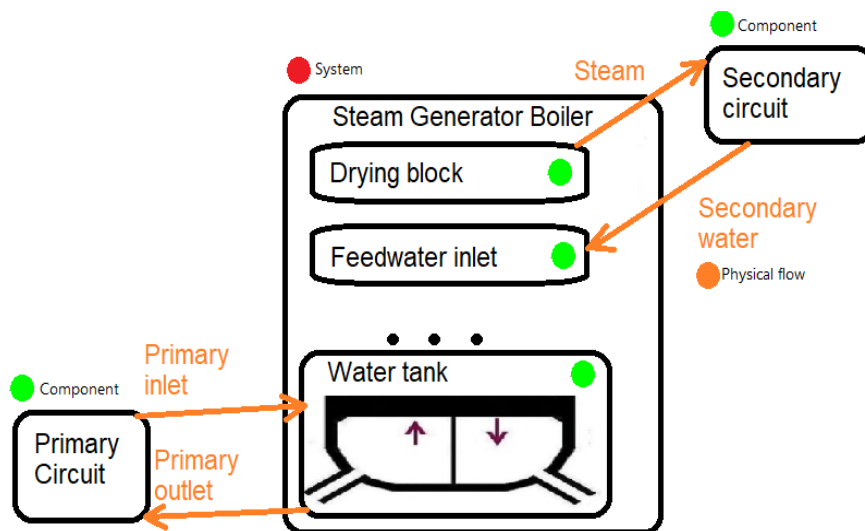


Fig. 5. Simplified model of Nuclear Steam Generator

4 Domain-Specific Language

Now we introduce the DSL supporting the syntactic description of our system through its hierarchy of objects as shown in Fig.5. Each object in the system model has a type, and hierarchical relations between types are strictly defined (some types can be recursive, e.g., a system may be composed of components and components may be composed of components). We propose interpreting these relations between DSL types as a Regular language (generated by type-3 grammar in Chomsky hierarchy [14]). I.e., if we have such a hierarchy of DSL terms: ‘system 1’ → ‘function A’ → ‘requirement X’, which barely denotes that requirement X is allocated to function A that is allocated to the system, then an appropriate list of types for the

‘requirement X’ position in the hierarchy will be [system] — [function] — [requirement] and it is an accepted word for the language. Regular Grammar and Finite Automata(FA) being equivalent [1], we propose to describe our DSL language with the FA ($\Sigma, S, s_0, \delta, F$) such that:

- Σ – list of input alphabet = list of DSL types,
- S is a set of states (one state per DSL type),
- s_0 – initial state (represents the root of objects DSL types hierarchy),
- δ is the state-transition function (displayed graphically in fig.6),
- F is the set of final states, equal to S to represent the simple DSL (hierarchical description in our example can be stopped at any level)

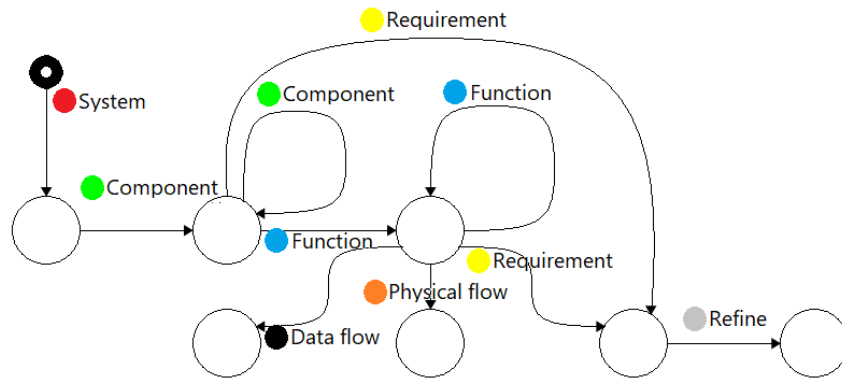


Fig. 6. Finite automata presentation of DSL types

Our sample DSL here is not universal. There can be other DSLs based on different types of grammar, some specific restrictions can be applied (like the rule ‘each Component should contain a function’), and so on. But most systems can be defined with such a simple breakdown. Note that DSL represented in Fig.6 is a type relation diagram, so having an arrow means that in a concrete example, there can be several children object instances for the given type (possibly no children).

At this step, our system of interest could be modeled either directly in a tool supporting the DSL in Fig 6 faithfully or by translating from an existing MBSE tool into our DSL. Our objective is to start from a DSL that is simple enough to enable a faithful translation into a theorem prover but smart enough to capture the complexity of a system model. The power of our approach relies on this trade-off between simplicity and faithfulness.

5 DSL model Translation

We show now that our DSL can be translated to an inductive type [8] which is a well-founded logical concept preserving the original semantics of DSL. It is possible to find a theorem prover supporting Inductive Types. We decided to use the well-known proof assistant COQ[6], based on the Calculus of Inductive Constructions (CIC). To reach our goal, we need to translate our DSL faithfully into COQ (according to Fig.2 in section 2), to define a list of formal rules that a system description in the DSL should comply with, and prove them with COQ tactics.

First, we define an inductive type “Direction” with possible values ‘input’ and ‘output’ to encode the direction of functional or requirement data flow messages. “direction” will be useful for any translation of DSL terms denoting messages or interactions between system items.

```
Inductive direction : Set := input | output.
```

Next, DSL types are translated into COQ types in a one-to-one manner, and these COQ types will be defined as a mutually inductive type that we call COQ_DSL. As an inductive type, each COQ type is built with a set of constructors.

Consider the DSL type ‘system’; the first constructor CSystem in Coq allows creating new COQ terms of type ‘system’. Then CComponent_System allows aggregating a set COQ_DSL ‘components’ to a COQ_DSL ‘system’. CFunction_System allows aggregating ‘functions’ in a ‘system’ and CSystem_System allows adding a set of terms of type ‘system’ in a ‘system’ (fig. 7). The constructors mimic the way we can build systems with the DSL and translate them into COQ.

With this translation specification in mind, we developed an automated generation of the type system in Coq based on a DSL with a one-to-one translation of DSL types into COQ types. Below you will find a pseudo-code example for translating a DSL into a COQ script file. Assume that all DSL types exist as a list in the typeCollection, and we have functions to get a list of children DSL types and to understand if the relation describes some interface (flow).

This way, the definition of DSL described in the first part of the paper can be translated into an inductive types in COQ, as presented in Fig. 7. On the image, you can see the top DSL type ‘System’, with all related DSL types reminded regarding each type in COQ (see fig.6 for DSL type relations). The ‘with’ keyword in the script remind all types are mutually inductive. We use natural numbers to encode unique identifiers assigned to each type, this is done to support the ability to distinguish objects instances. Object attributes describing some properties can be assigned similarly; a small example of properties will be shown in the next section.

```
Definition Identifier : Type := N.
Inductive System : Type :=
| CSystem : Identifier -> System
| CComponent_System : list Component -> System -> System
with Component : Type :=
| CComponent : Identifier -> Component
| CComponent_Component : list Component -> Component -> Component
| CRequirement_Component : list Requirement -> Component -> Component
| CFunction_Component : list Function -> Component -> Component
with Function : Type :=
| CFunction : Identifier -> Function
| CFunction_Function : list Function -> Function -> Function
| CRequirement_Function : list Requirement -> Function -> Function
| CData_flow_Function : list Data_flow -> direction -> Function -> Function
| CPhysicla_flow_Function : list Physical_flow -> direction -> Function -> Function
with Requirement : Type :=
| CRequirement : Identifier -> Requirement
| CRefine_Requirement : list Refine -> direction -> Requirement -> Requirement
with Data_flow : Type :=
| CData_flow : Identifier -> Data_flow
with Physical_flow : Type :=
| CPhysical_flow : Identifier -> Physical_flow
with Refine : Type :=
| CRefine : Identifier -> Refine.
```

Fig. 7. Translation DSL into COQ inductive types

6 Translation of system model to COQ

Having defined the translation of DSL types into COQ types, we now address the translation of DSL terms into COQ terms. We do use a recursive bottom-up algorithm starting from leaf DSL items and then translating upper DSL items with the set of children COQ terms already translated. After translation, we do compile the resulting files into a COQ module. This step is already a proof of correctness of the export process because COQ will warn if any inconsistency is found between translated DSL terms and types.

In the example given (fig.8), you can see the part of the hierarchy of objects and the flow input (the real example of this system contains more than 1000 lines in a COQ file):

- Steam_Generator_Boiler is an object of type System, is a parent of several Components: Drying_block, Feedwater_inlet, Water_tank (and some others in real code)
- Drying_block in an object of type component, it has some children of type Function and some children of type Component
- Feedwater_inlet is also an object of type Component, consuming 'Secondary_water' flow of type 'Physical flow.'

```
...
Definition Feedwater_inlet_27 := CFunction_Component [ ... ]
  (CPhysical_flow_Component [ Secondary_water; ... ] input.
   (CComponent 2194728288258)).
...
Definition Drying_block_11 := CFunction_Component [... ].
  (CComponent_Component [... ].
   (CComponent 1103806595096)).
...
Definition Steam_Generator_Boiler_1 := CComponent_System.
  [ Drying_block_11;
    Feedwater_inlet_27;
    Water_tank_32;
    ... ]
  (CSystem 1013612281857).
```

Fig. 8. shortened representation of data model in the COQ

The following pseudo python code is intended to walk through all objects of DSL types and generate appropriate commands into the COQ script resulting in the definition of the system with Coq inductive type:

```
Walk recursively through hierarchical object structure:
When all children are parsed:
  Write 'Definition ' + object name + ':='
For each child type:
  Write Child Type list Constructor +
```

```
List of children of the given type
Write Object Type constructor + object Identifier
Write \.'
```

To check that translation of the system model is robust, we did define a backward-translation automatic script, converting COQ types into DSL and creating a DSL terms model according to COQ data. To make a fair transition, this script calls COQ to load modules containing data and reads defined type contents with COQ ‘Print’ command. It allows us to compare initial and resulting data in the system definition tool we use to show that all translations are bijective.

7 Verification

Once our system description is translated into COQ, we benefit from the advantages of a theorem prover. We can formalize properties that should be met by the system, hypotheses and demonstrate these properties.

For example, let’s consider a property about data flows between Systems, Components, and Functions. An expected property of flows in the data model is that each produced flow should be consumed somewhere and conversely in the hierarchy of instances. Let’s determine how to translate this property into COQ and make the proof that the model satisfies this property. We need to define the predicate as “any consumed data is produced” (“any produced data is consumed” is defined in the same way). Before specifying the goal, we need to create some recursive functions, walk through the hierarchy and look for flow objects. Here is the automatically created set of functions required for performing the verification (we display only a part of the proof).

"msgfunction" holds for a data d, a function g, and a production ‘direction’ (introduced in section 5) p if g contains (possibly in its subparts) the data flow d with direction p. Other definitions play a similar role for described COQ types. They study the given COQ type according to its definition and call appropriate functions recursively (to answer if the given flow with its direction is found in the term hierarchy or not):

```
Fixpoint msgfunction (d:Physical_flow) (p:direction) (g:Function): Prop :=
match g with
| CFunctionMM n => False
| CPhysical_flow_Function listData prod obj => (In d listData) /\ (prod = p) \/ (msgfunction d p obj)
...
end.
Fixpoint msgComponent (d:Physical_flow) (p:direction) (g:Component) {struct g}: Prop :=..
match g with
| CComponent n => False
| CFunction_Component lf s => ( fold_left or (map (msgFunction d p) lf ) False ) \/ (msgComponent p s)
...
end.
Fixpoint msgsystem (d:Physical_flow) (p:direction) (g:System) {struct g}: Prop :=..
match g with
| CSystemMM n => False
| CComponent_SystemMM lf s => ( fold_left or (map (msgComponent d p) lf ) False ) \/ (msgsystem d p s)
end.
```

Finally, we can describe the goal, a property that can be proved by Coq tactics. Normally

the definition and Goal below should be understandable intuitively from the previous definitions.

```
Definition EveryConsumedDataIsProduced(s: System) :=  
forall (d:Physical_flow), (msgsystem d input s)-> (msgsystem d output s).  
  
Goal (EveryConsumedDataIsProduced Steam_Generator_Boiler_1).  
intro. simpl. tauto..  
Qed.  
  
Definition EveryProducedDataIsConsumed(s: System) :=  
forall (d:Physical_flow), (msgsystem d output s)-> (msgsystem d input s).
```

With our experiments, we found that if the model is correct, it can be proved by Coq with simple tactics (automatically, as the set of tactics required is trivial), in case of missed producer/consumer COQ will simply warn about the failed theorem proof. In that case, we need to perform some investigation to pinpoint the exact problem in the model. To find the problem, we wrote a script for an ‘interactive’ search for the problem in COQ terms, calling some COQ commands to perform checks and generating the next request with defined patterns.

8 FMEA

One interesting application of proofs of correctness and completeness is the safety (hazard) analyses. There are several hazard analysis methods available for investigating possible failures and describing ways to reduce risks. The most known ones are Failure Mode and Effect Analysis [3] and System Theoretic Process Analysis (STPA). The FMEA safety analysis method uses a physical component diagram (splitting the investigated system into parts, starting from low-level components and proceeding up to the failure effect of the overall system). At the same time, SPTA is based on a functional control diagram, analyzing the dynamic behavior of the system [2]. FMEA is an inductive procedure summarizing upward effects on the system from subsystems. So, the FMEA analysis is very well-structured, allowing to use of some taxonomy to define the order of analysis actions and relations between them. A good example of detailed taxonomy and conditions to be verified can be found in [7].

The DSL proposed in the first part of the paper is designed to describe the hierarchical decomposition of a system into basic elements, which is the source for FMEA analysis. In the image below, you can see the standard FMEA table according to (AFCEN guide [12]) with some assigned DSL types for objects (with some properties added), representing data for columns. The table (fig.9) is generated according to hierarchical object data stored in the system model.

● Component
● Function
➤ Failure mode
➤ Failure effect
■ Cause
● Requirement
➤ Risk reduction means

Component	Function	Failure mode	Failure effects			Essential safety requirements (order, decree)	Risk reduction means	
			Description	Pressure / Radioprotection risk	Phase		Proof means	Proof
U-tubes&Tube Sheet welding	Confine fluids in all Temp/Pressure situations	Plastic Instability	Loss of containment of Primary fluid in any situation	Pressure risk	Design	A1-2 A1-3.6	Use of a proven calculation method	Dimensioning reviews
U-tubes&Tube Sheet welding	Confine fluids in all Temp/Pressure situations	Fast fracture	Leak of radioactive substances	Radioprotection risk	Procurement	A1-3.1 A1-3.2 A1-3.5 A1-4.3	Drafting a procurement specification for nuclear pressure	Supply specification
U-tubes	Confine fluids in all Temp/Pressure situations	Fast fracture	Leak of radioactive substances	Radioprotection risk	Procurement	A1-3.1 A1-3.2 A1-3.5 A1-4.3	Drafting a procurement specification for nuclear pressure	Supply specification

Fig. 9. FMEA tale with DSL language types assigned

Having the data structure required and FMEA logic, we can define some completeness formal rules to prove in COQ. As an example of property defined by law, we consider the classification of equipment parts (depending on their role with regards to pressure resistance) defined by Autorité de sûreté nucléaire (Nuclear Safety Authority, ASN), represented in fig.10 (shortened comparing to [9]):

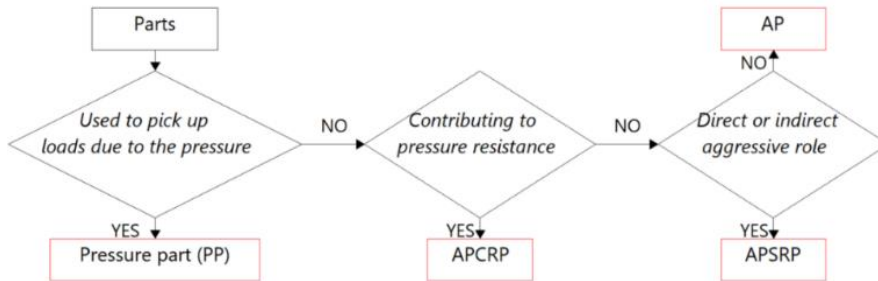


Fig. 10. Component classification logic

To represent those properties in our model, we need first to define appropriate attributes for each Component object (with a defined list of possible values for each attribute):

- ‘is used to pick up loads due to the pressure’ – yes/no,
- ‘is contributing to pressure resistance’ – yes/no,
- ‘is direct or indirect aggressive role’ – yes/no,
- ‘classification’ – PP/AP/APCRP/APSRP.

Then those attribute values can be translated to the coq model as several sets and assigned

to object instances:

```
Inductive PressureLoads : Set := P_yes | P_no..
Inductive PressureResistance : Set := R_yes | R_no..
Inductive DirectAggressiveRole : Set := A_yes | A_no..
Inductive ObjType : Set := APCRP | PP | APSRP | AP..
Inductive Component : Type :=
| CComponent : nat -> PressureLoads -> PressureResistance.
  -> DirectAggressiveRole -> ObjType -> Component
| CFunction_Component : list Function -> Component -> Component.
Definition Component3 := CComponent 3 P_no R_no A_yes AP.
```

From a practical point of view, identifying all pressure parts requires identifying parts that have an interface with primary or secondary circuits water and steam. So starting from a specification of all systems and parts interfaces, it is possible to formally identify all PP parts from the Coq Model. Explaining this in detail goes beyond the objective of this paper, but this is a good example of a part of the safety case that could be supported automatically and faithfully by a proof system. Of course, the identification of interfaces is based on systems engineer analysis and may be the weak spot of the proof.

9 Conclusion

The paper shows how system models can be translated into COQ definitions. Making proofs using only system descriptions allows us to work on a high abstract level (without detailed system behavior descriptions). This is especially major for investigating the correctness and completeness of the system with bottom-up safety analysis like FMEA.

Using a general system definition language (DSL) including recursive types definition, we did turn the DSL into a COQ inductive types; having this, we can translate the syntax of the system model (DSL terms) into definitions of COQ terms. Starting from system properties to be proven, we can translate them formally into COQ theorems and prove them either automatically or not. COQ is also capable of finding counterexamples in many cases if a property is not true (a small description is given at the end of section 7).

With this approach, systems and safety engineers get the possibility to back their systems and safety analysis with formal proofs of correctness and completeness that have to be stated in parallel.

Now we plan to investigate further topics to push the limits on this approach - investigate the completeness and correctness properties that systems and safety analysis should comply with. We started with simple properties in this paper, but generally, the Safety case comprises high-level goals that are decomposed into simpler elementary goals and action plans. We will investigate the formal rules behind such a decomposition. In parallel, we will look for a formal analysis regarding the completeness and correctness of a system design.

Then we need to check the resilience of a proof system like Coq when investigating models made of 10th or 100th thousands of objects, as this is the size of the complex system models we discuss.

After that, we would like to investigate the user's assistance when performing a system or safety analysis. For instance, in case we already know the functional and system architecture of the system, we can check for completeness and correctness properties and raise anomalies to the user. In case a safety goal is being decomposed, we can identify automatically if all subgoals are covering all the possibilities or not.

Finally, we will investigate how to connect MBSE tools to our system and safety formal DSL description, allowing the formal models to be filled by an existing modeler. Of course, we will have to prove that the translation is faithful, which could be made by bi-directional translation between the MBSE and the formal DSL and proof of evidence that there is a bijection.

References

- [1] Grune, D., Jacobs, C.J.H. (2008). Regular Grammars and Finite-State Automata. In: Parsing Techniques. Monographs in Computer Science. Springer, New York
- [2] Sulaman, S., Beer, A., Felderer, M. et al. Comparison of the FMEA and STPA safety analysis methods—a case study. *Software Qual J* 27, 349–387 (2019).
- [3] N. Storey, "Safety-critical computer systems", Addison-Wesley, 1996.
- [4] Gleirscher M, Foster S, Woodcock J (2019) New opportunities for integrated formal methods. *ACM Comput Surv* 52 (6):117:1–117:36.
- [5] I. Lopatkin, Y. Prokhorova, E. Troubitsyna, A. Iliasov, and A. Romanovsky. Patterns for Representing FMEA in Formal Specification of Control Systems. TUCS Technical Report 1003, 2011.
- [6] COQ, <https://coq.inria.fr/documentation>, last accessed 2022/03/29.
- [7] E. Piljugin, S. Authén, J-E. Holmberg. Proposal for the Taxonomy of Failure Modes of Digital System Hardware for PSA. 11th International Probabilistic Safety Assessment and Management Conference & The Annual European Safety and Reliability Conference At: Helsinki, 2012
- [8] Pfenning, F., Paulin-Mohring, C. (1990). Inductively defined types in the Calculus of Constructions. In: Main, M., Melton, A., Mislove, M., Schmidt, D. (eds) *Mathematical Foundations of Programming Semantics*. MFPS 1989.
- [9] Conformity assessment of nuclear pressure equipment. Autorite de surite nucleaire. Guide N8. 2012.
- [10] S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, volume 4085 of LNCS, pages 460–475. Springer, 2006.
- [11] National Institutes of Standard Technologies, Planning Report 02-3 The Economic Impacts of Inadequate Infrastructure for Software, 2002
- [12] Guide ADR (Analyse de risques) pour ESPN N1, AFCEN, 2018
- [13] INCOSE Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities, 4th Edition, INCOSE, 2015
- [14] Chomsky, Noam. "Three models for the description of language". *IRE Transactions on Information Theory*, 1956

Do CPS translations also translate realizers?

Samuel Gardelle¹ and Étienne Miquey²

¹ ENS de Lyon, France

`samuel.gardelle@ens-lyon.fr`

² Aix-Marseille Université, France

`etienne.miquey@univ-amu.fr`

Abstract

In the realm of the proofs-as-programs correspondence, continuation-passing style (CPS) translations are known to be twofold: they bring both a program translation and a logical translation. In particular, when using the former to compile a language with a control operator, the latter ensures the soundness of the compilation with respect to types.

This work is inspired by [OS08], in which Oliva and Streicher explained how Krivine realizability could be rephrased as the composition of a CPS and an intuitionistic realizability model. In this paper, we propose to push one step forward the analysis of the relation between realizability models and CPS translations to investigate the following question: assume that two realizability models are defined using the source and the destination of a CPS translation, is it the case that the CPS translates realizers of a given formula into realizers of the translated formulas?

1 Introduction

Continuation-passing style translations, which were first introduced by Sussman and Steel [SS75], constitute a great tool when it comes to studying operational semantics of calculi: by making explicit the order in which reduction steps are computed, CPS translations indirectly specify an evaluation strategy for the translated calculus. In particular, continuation-passing style translations have a lot of applications in terms of compilation and have been widely studied for call-by-name and call-by-value strategies of the λ -calculus [Pl075, App92, SF93].

From a logical perspective, CPS translations are also very informative insofar as they induce a translation at the level of types that mostly amounts to a syntactical model allowing to transfer logical properties (coherence, normalization) from the target calculus [BPT17]. For instance, standard CPS translations are known to correspond to embeddings of classical logic into intuitionistic logic through variants of Gödel's negative translation [Gri90, Mur90]. Computationally, the latter corresponds to Griffin's seminal observation that a classical Curry-Howard correspondence can be obtained by extending the λ -calculus with control operators, *e.g.* Scheme's `call/cc`. These operators provide a *direct* handle on continuations (allowing in particular the definition of backtracking programs), as opposed to the *indirect* one provided by CPS translations. Several calculi were born from this idea, amongst which Krivine's λ_c -calculus [Kri04].

Elaborating on this calculus, Krivine developed in the late 90s the theory of classical realizability, which is a complete reformulation of its intuitionistic twin. This theory has shown to be particularly fruitful, both to analyze the computational content of proofs [Kri03, Miqu11b, Miqu18] or to define new models of classical theories [Kri12, Kri21].

Studying the structure of Krivine's classical realizability, Oliva and Streicher showed how the latter could in fact be viewed as the composition of a CPS with a traditional intuitionistic realizability interpretation [OS08]. This observation unveils a somewhat surprising situation: the very nature of a CPS translation is *syntactical* and as such, it is quite unexpected that this

turns out to be well-behaved with respect to realizability, a *semantic* notion. In fact, taking a closer look at Oliva and Streicher’s work [OS08], their (classical) realizers are defined through the computational behavior of their CPS translation. In line with this work, Frey also defines a notion of classical realizability directly within the target of a CPS translation [Fre16]. In both cases, realizers are thus compatible with the CPS *by definition*. In a slightly different setting, Miquel studied the witness extraction mechanism of classical realizability for Σ_1^0 formulas through the CPS translation, but his results only apply to typed terms [Miq11b].

Therefore, none of these works tackle the following question: *do CPS translations also translate realizers?* To phrase it a bit more precisely, let us consider a CPS translation $[\cdot]$ from a (classical) source calculus, say Krivine λ_c -calculus, to an intuitionistic calculus, say the λ -calculus, and let us write $\llbracket \cdot \rrbracket$ for the translation on types it induces. Assume besides that these two calculi serve as the underlying language of realizers for two realizability interpretations, say for second-order classical and intuitionistic arithmetic (**PA2** and **HA2**) respectively. Now, if t is a term of type A (in the source), then $\llbracket t \rrbracket$ is of type $\llbracket A \rrbracket$ (in the destination); but *is it the case that if t realizes A then $\llbracket t \rrbracket$ realizes $\llbracket A \rrbracket$?*

To investigate this question, we will use a very convenient tool to reason about realizability models: *evidenced frames* [CMT21] which, as we will see in Section 2.2, capture the algebraic structure of realizability interpretations. In fact, this work was also an excuse to put this recent notion in practice and to test the companion notion of morphism against a concrete candidate. One can think of an evidenced frames morphism as a functional embedding of a realizability interpretation into another one, and as such, CPS translations provide us with very natural nontrivial candidates. The main question of this paper could indeed be rephrased in these terms: *do CPS translations define evidenced frame morphisms?* As we shall see in Section 3.2, the answer is nuanced in that in general a CPS translation does not induce an evidenced frame morphism, but we can nonetheless introduce another evidenced frame, which corresponds to the image of the source calculus through the translation and can serve as the codomain of an evidenced frame morphism.

Outline of the paper We start by giving a standard example of a realizability interpretation for **HA2** based on the λ -calculus in Section 2, which we will later use as the destination of the CPS translation we will consider. We take advantage of this section to recall the definition of evidenced frames, and illustrate how the interpretation for **HA2** naturally induces an evidenced frame $\mathcal{E}\mathcal{F}_{\text{HA2}}$. We then introduce (a call-by-value presentation of) Curien-Herbelin’s $\lambda\mu\tilde{\mu}$ -calculus in Section 3, together with its CPS translation into the (pure) λ -calculus. Finally, in Section 4 we define an evidenced frame $\mathcal{E}\mathcal{F}_{\mu\tilde{\mu}}^{\text{bv}}$ corresponding to a Krivine realizability interpretation based on the $\lambda\mu\tilde{\mu}$ -calculus, and we investigate the CPS translation in terms of evidenced frame morphisms. In particular, we show that in general it does not define a morphism from $\mathcal{E}\mathcal{F}_{\mu\tilde{\mu}}^{\text{bv}}$ to $\mathcal{E}\mathcal{F}_{\text{LJ2}}$, but that another evidenced frame $\mathcal{E}\mathcal{F}_{\text{fw}}$ (which also defines a realizability interpretation for **HA2**) can be introduced for the CPS to define an appropriate morphism $\mathcal{E}\mathcal{F}_{\mu\tilde{\mu}}^{\text{bv}} \rightarrow \mathcal{E}\mathcal{F}_{\text{fw}}$.

Due to the page limit, some proofs are only sketched and others have been omitted, an extended version with complete proofs is accessible at: <https://hal.inria.fr/hal-03910311>.

2 Evidenced frames

Before introducing evidenced frames, we give a first example of a (very standard) realizability interpretation that will serve both as the destination of the CPS translation considered in the

$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \text{ (Ax)}$	$\frac{\Gamma \vdash t : A \rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B} \text{ (}\rightarrow_E\text{)}$	$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : A \rightarrow B} \text{ (}\rightarrow_I\text{)}$
$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A \wedge B} \text{ (}\wedge_I\text{)}$	$\frac{\Gamma \vdash t : A \wedge B \quad \Gamma, x : A, y : B \vdash u : C}{\Gamma \vdash \mathbf{let} (x, y) = t \mathbf{ in } u : C} \text{ (}\wedge_E\text{)}$	$\frac{\Gamma \vdash t : A[x := n]}{\Gamma \vdash t : \exists x. A} \text{ (}\exists_I^1\text{)}$
$\frac{\Gamma \vdash t : \forall x. A}{\Gamma \vdash t : A[x := n]} \text{ (}\forall_E^1\text{)}$	$\frac{\Gamma \vdash t : A \quad x \notin FV(\Gamma)}{\Gamma \vdash t : \forall x. A} \text{ (}\forall_I^1\text{)}$	$\frac{\Gamma \vdash t : A[X(x_1, \dots, x_n) := B]}{\Gamma \vdash t : \exists X. A} \text{ (}\exists_I^2\text{)}$
$\frac{\Gamma \vdash t : \forall X. A}{\Gamma \vdash t : A[X(x_1, \dots, x_n) := B]} \text{ (}\forall_E^2\text{)}$	$\frac{\Gamma \vdash t : A \quad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X. A} \text{ (}\forall_I^2\text{)}$	$\frac{\Gamma \vdash t : A' \quad A \cong A'}{\Gamma \vdash t : A} \text{ (}\cong\text{)}$

Figure 1: A type system for **HA2**

sequel and as an introducing example for evidenced frames.

2.1 Realizability interpretation of HA2

2.1.1 Heyting second-order arithmetic

We start by introducing the terms and formulas of Heyting second-order arithmetic, for which we mostly follow Miquel's presentation [Miq11a]. Second-order formulas are built on top of first-order arithmetical expressions, by means of logical connectives, first- and second-order quantifications and primitive predicates. We use upper case letters for second-order variables and lower case letters for first-order ones.

We consider the usual λ -calculus terms extended with (positive) pairs and the corresponding destructor (written $\lambda(x, y).t$). In the last sections of the paper, we will also include primitives for booleans for technical purposes. The syntax of formulas and terms is given by

1st-order exp.	$e ::= x \mid f(e_1, \dots, e_n)$
Formulas	$A, B ::= X(e_1, \dots, e_n) \mid A \rightarrow B \mid A \wedge B \mid \forall x. A \mid \exists x. A \mid \forall X. A \mid \exists X. A$
Terms	$t, u ::= \lambda x. t \mid t u \mid (t, u) \mid \mathbf{let} (x, y) = t \mathbf{ in } u$

where $f : \mathbb{N}^n \rightarrow \mathbb{N}$ is any primitive recursive function. We write Λ for the set of all closed λ -terms, and we may use the following usual shorthands: $\top \triangleq \exists X. X$, $\perp \triangleq \forall X. X$ and $\neg A \triangleq A \rightarrow \perp$.

To simplify the use of existential quantifiers, as in [Miq11a], we introduce the following congruence rules, where the variables x, X are not free in B

$$(\exists x. A) \rightarrow B \cong \forall x. (A \rightarrow B) \qquad (\exists X. A) \rightarrow B \cong \forall X. (A \rightarrow B) \quad (1)$$

These congruences allow us to avoid having elimination rules for the existential quantifiers, thus simplifying the resulting type system. The type system, which is given in Figure 1, corresponds to the usual rules of natural deduction. The reader may observe that in particular, no computational content is given to quantifiers in the type system.

The one-step (weak head-) reduction over terms is defined by the following rules:

$$\overline{(\lambda x. t)u \triangleright_\beta t[u/x]} \qquad \overline{\mathbf{let} (x, y) = (u, v) \mathbf{ in } t \triangleright_\beta t[u/x][v/y]} \qquad \frac{t \triangleright_\beta t'}{C[t] \triangleright_\beta C[t']}$$

where $C[] ::= [] \mid C[[] u] \mid C[\mathbf{let} (x, y) = [] \mathbf{in} t]$. We write \rightarrow_β for the reflexive-transitive closure of \triangleright_β , which is known to be deterministic¹, type-preserving and normalizing on typed terms [Bar92].

2.1.2 Realizability interpretation

We will now see how to define a realizability interpretation relying on the type system defined in Figure 1. Formulas are interpreted as *saturated sets of terms*, i.e. as sets of closed terms $S \subseteq \Lambda$ such that $t \rightarrow_\beta t'$ and $t' \in S$ imply that $t \in S$. We write \mathbf{SAT} to denote the set of all saturated sets and, given a formula A , we call *truth value* its realizability interpretation.

Definition 1 (Valuation). A *valuation* is a function ρ that associates a natural number $\rho(x)$ to every first-order variable x and a *truth value function* $\rho(X)$, i.e. a function in $\mathbb{N}^k \rightarrow \mathbf{SAT}$ to every second-order variable X of arity k .

1. Given a valuation ρ , a first-order variable x and a natural number n , we denote by $\rho, x \mapsto n$ the valuation defined by $(\rho, x \mapsto n) \triangleq \rho|_{\text{dom}(\rho) \setminus \{x\}} \cup \{x \mapsto n\}$.
2. Given a valuation ρ , a second-order variable X of arity k and a truth value function $F : \mathbb{N}^k \rightarrow \mathbf{SAT}$, the valuation defined by $(\rho, X \mapsto F) \triangleq \rho|_{\text{dom}(\rho) \setminus \{X\}} \cup \{X \mapsto F\}$ will be denoted by $\rho, X \mapsto F$.

We say that a valuation ρ is *closing* the formula A if $FV(A) \subseteq \text{dom}(\rho)$.

Definition 2 (Realizability interpretation). We interpret closed arithmetical expressions e in the standard model of first-order Peano arithmetic \mathbb{N} . Given a valuation ρ and a first-order expression e (whose variables are in the domain of ρ) we denote its interpretation by $\llbracket e \rrbracket_\rho$. The interpretation of a formula A together with a valuation ρ closing A is the set $|A|_\rho$ defined inductively according to the following clauses:

$$\begin{array}{l} |X(e_1, \dots, e_n)|_\rho \triangleq \rho(X)(\llbracket e_1 \rrbracket_\rho, \dots, \llbracket e_n \rrbracket_\rho) \\ |A \rightarrow B|_\rho \triangleq \{t \in \Lambda : \forall u \in |A|_\rho. (tu \in |B|_\rho)\} \\ |A \wedge B|_\rho \triangleq \{t \in \Lambda : \exists u \in |A|_\rho. \exists v \in |B|_\rho. t \rightarrow_\beta (u, v)\} \end{array} \quad \left| \begin{array}{l} |\forall x. A|_\rho \triangleq \bigcap_{n \in \mathbb{N}} |A|_{\rho, x \mapsto n} \\ |\exists x. A|_\rho \triangleq \bigcup_{n \in \mathbb{N}} |A|_{\rho, x \mapsto n} \\ |\forall X. A|_\rho \triangleq \bigcap_{F: \mathbb{N}^k \rightarrow \mathbf{SAT}} |A|_{\rho, X \mapsto F} \\ |\exists X. A|_\rho \triangleq \bigcup_{F: \mathbb{N}^k \rightarrow \mathbf{SAT}} |A|_{\rho, X \mapsto F} \end{array} \right.$$

Observe that in the previous definition, the universal quantification cannot be seen as a generalized conjunction. Indeed, the conjunction is given computational content through pairs, while the universal quantifications are defined as intersections of truth values.

It is easy to see that for any formula A and any valuation ρ closing A , one has $|A|_\rho \in \mathbf{SAT}$. As it turns out, the congruences defined by Equation (1) are sound w.r.t. the interpretation.

Proposition 3 ([Miq11a]). *If A and A' are two formulas of HA2 such that $A \cong A'$, then for all valuations ρ closing both A and A' we have $|A|_\rho = |A'|_\rho$.*

To express that the realizability interpretation is sound with respect to the type system we need the following preliminary notions.

¹We also could have considered a non-deterministic reduction relation (i.e.. without enforcing any evaluation strategy) without altering the foregoing definition of the realizability interpretation. Nonetheless, this choice will provide us with a tighter preservation of reduction through the CPS translation.

Definition 4 (Substitution). A *substitution* is a finite function σ from λ -variables to closed λ -terms. Given a substitution σ , a λ -variable x and a closed λ -term u , we denote by $(\sigma, x := u)$ the substitution defined by $(\sigma, x := u) \triangleq \sigma \upharpoonright_{\text{dom}(\sigma) \setminus \{x\}} \cup \{x := u\}$.

Definition 5. Given a context Γ and a valuation ρ closing the formulas in Γ , we say that a substitution σ *realizes* $\rho(\Gamma)$ and write $\sigma \Vdash \rho(\Gamma)$ if $\text{dom}(\Gamma) \subseteq \text{dom}(\sigma)$ and $\sigma(x) \in |A|_\rho$ for every declaration $(x : A) \in \Gamma$.

Definition 6. A typing judgement $\Gamma \vdash t : A$ is *adequate* if for all valuations ρ closing A and Γ and for all substitutions $\sigma \Vdash \rho(\Gamma)$ we have $\sigma(t) \in |A|_\rho$. More generally, we say that an inference

rule $\frac{J_1 \quad \cdots \quad J_n}{J_0}$ is adequate if the adequacy of all typing judgements J_1, \dots, J_n implies the adequacy of the typing judgement J_0 .

Theorem 7 (Adequacy [Miq11a]). *The typing rules of Figure 1 are adequate.*

The adequacy theorem is the key result when defining realizability interpretations in that fundamental properties stem from it. For example, we have the following corollary.

Corollary 8 (Consistency). *There is no proof term t such that $\vdash t : \perp$.*

Proof. The proof is by *reductio ad absurdum*. It follows from Theorem 7 that if $\Gamma \vdash t : A$ is derivable, then it is adequate. In this case, this entails $t \in |\perp|_\rho = |\forall X.X|_\rho = \bigcap_{S \in \mathbf{SAT}} S = \emptyset$. \square

While a complete introduction to realizability interpretations and their benefits to prove properties such as soundness or normalization of typed calculi is out of the scope of this paper², we would like to point out nonetheless that the proof of adequacy is very flexible. Indeed, if one wants to add a new instruction to the language of terms via its typing rule, it is enough to check that this typing rule is adequate while the remainder of the proof is exactly the same. For instance, to extend the present setting with booleans (as we shall do later on) it is enough to introduce terms $\text{tt}, \text{ff}, \text{if } b \text{ then } t \text{ else } u$ with their typing rules and to prove that the latter are adequate with the realizability interpretation.

2.2 Evidenced frames

In the previous section, we have seen an example of a proof system labelled with proof terms derived from the λ -calculus. Note that if proof terms are indeed realizers, there exists realizers that are not typable. Take for example a non-typable term Ω , and observe that $(\lambda _ . \lambda x.x) \Omega$ is not typable but realizes $\forall X.X \rightarrow X$ because $\lambda x.x$ does. The realizability interpretation generalizes this proof system in that it is only concerned with the behavior (semantic) of proof terms and not their syntax. Note that we loose decidability but we are not concerned about it since we only want to build interpretations. The essence of a realizability interpretation lies between the interaction of a programming language and a language of formulas. The formalism of evidenced frames seeks to abstract in a unified way this structure. It is composed of a triple (with axioms) that contains two languages: evidences (programs) and proposition (formulas) as well as a relation $\cdot \dot{\rightarrow} \cdot$ that connects them.

²We refer the reader interested in this to the existing literature on classical realizability, e.g. [Rie14, Lep17, Miq17].

Definition 9 ([CMT21]). An *evidenced frame* is a triple $(\Phi, E, \cdot \dot{\rightarrow} \cdot)$, where Φ is a set of propositions, E is a collection of evidences, and $\phi_1 \xrightarrow{e} \phi_2$ is a ternary evidence relation on $\Phi \times E \times \Phi$, along with the following³:

Reflexivity There exists evidence $e_{\text{id}} \in E$:

$$- \forall \phi. \phi \xrightarrow{e_{\text{id}}} \phi$$

Transitivity There exists an operator $;\in E \times E \rightarrow E$:

$$- \forall \phi_1, \phi_2, \phi_3, e, e'. \phi_1 \xrightarrow{e} \phi_2 \text{ and } \phi_2 \xrightarrow{e'} \phi_3 \implies \phi_1 \xrightarrow{e;e'} \phi_3$$

Top A proposition $\top \in \Phi$ such that there exists evidence $e_{\top} \in E$:

$$- \forall \phi. \phi \xrightarrow{e_{\top}} \top$$

Conjunction An operator $\wedge \in \Phi \times \Phi \rightarrow \Phi$ such that there exists an operator $\langle \cdot, \cdot \rangle \in E \times E \rightarrow E$ and evidences $e_{\text{fst}}, e_{\text{snd}} \in E$:

$$\begin{aligned} - \forall \phi_1, \phi_2. \phi_1 \wedge \phi_2 \xrightarrow{e_{\text{fst}}} \phi_1 & \quad - \forall \phi, \phi_1, \phi_2, e_1, e_2. \phi \xrightarrow{e_1} \phi_1 \text{ and } \phi \xrightarrow{e_2} \phi_2 \implies \phi \xrightarrow{\langle e_1, e_2 \rangle} \phi_1 \wedge \phi_2 \\ - \forall \phi_1, \phi_2. \phi_1 \wedge \phi_2 \xrightarrow{e_{\text{snd}}} \phi_2 & \end{aligned}$$

Universal Implication An operator $\supset \in \Phi \times \mathcal{P}(\Phi) \rightarrow \Phi$ such that there exists an operator $\lambda \in E \rightarrow E$ and evidence $e_{\text{eval}} \in E$:

$$\begin{aligned} - \forall \phi_1, \phi_2, \vec{\phi}, e. (\forall \phi \in \vec{\phi}. \phi_1 \wedge \phi_2 \xrightarrow{e} \phi) \implies \phi_1 \xrightarrow{\lambda e} \phi_2 \supset \vec{\phi} \\ - \forall \phi_1, \vec{\phi}, \phi \in \vec{\phi}. (\phi_1 \supset \vec{\phi}) \wedge \phi_1 \xrightarrow{e_{\text{eval}}} \phi \end{aligned}$$

Given an evidenced frame $(\Phi, E, \cdot \dot{\rightarrow} \cdot)$, we say that $e \in E$ is *evidence* of $\phi \in \Phi$ if $\top \xrightarrow{e} \phi$ holds. The evidenced frame is said to *model* ϕ if it has evidence of ϕ . An evidenced frame is *consistent* if it does not model \perp .

Note that contrary to cartesian closed categories, this formalism does not enforce any equation between arrows, in fact it does not allow for the axiomatization of reductions: we only require that the languages of propositions and evidences are expressive enough.

Remark 10. If there exists a huge literature describing realizability interpretations for different theories based on different notions of computations, the notion of “*realizability interpretation*” itself does not have a formal definition. The best approximation that one could come up with would probably be *an interpretation of formulas as sets of computing terms* plus some extra intuitions on how *terms should compute accordingly to the connectives they realize*. The study of its categorical counterpart gives a more precise picture: the interpretation should induce a tripos [Pit02, vO08]. As shown in [CMT21], evidenced frames are complete with respect to triposes, and reflect the structure of a realizability interpretations in a much more faithful way than triposes do. As such, “*it is an evidenced frame*” is probably the best definition one could give of a realizability interpretation, the definition of an evidenced frame and of its different components specifying how formulas are interpreted and what “*to realize*” means. In the next section, we will show how the realizability interpretation given for **HA2** indeed defines an evidenced frame, but in the sequel of the paper, the reader should understand the existence of an evidenced frame as the definition of a realizability interpretation.

³Observe that the different construct on propositions and evidences are actually part of the definition of an evidenced frame (in particular, several different evidenced frames may be induced from one given triple $(\Phi, E, \cdot \dot{\rightarrow} \cdot)$). In the sequel, for conciseness we may nonetheless only state the existence of evidenced frames through this triple, giving the other defining constructs in the proofs.

Definition 11. A *morphism*⁴ from $\mathcal{EF}_1 = \langle \Phi_1, E_1, \cdot \dot{\rightarrow}_1 \cdot \rangle$ to $\mathcal{EF}_2 = \langle \Phi_2, E_2, \cdot \dot{\rightarrow}_2 \cdot \rangle$ is a function $F : \Phi_1 \rightarrow \Phi_2$ satisfying the following properties:

1. $\forall e_1. \exists e_2. \forall \varphi_1, \varphi'_1. \varphi_1 \xrightarrow{e_1} \varphi'_1 \implies F(\varphi_1) \xrightarrow{e_2} F(\varphi'_1)$
2. $\exists e_2. \top_2 \xrightarrow{e_2} F(\top_1)$
3. $\exists e_2. \forall \varphi_1, \varphi'_1. F(\varphi_1) \wedge_2 F(\varphi'_1) \xrightarrow{e_2} F(\varphi_1 \wedge_1 \varphi'_1)$
4. $\exists e_2. \forall \varphi_1, \vec{\phi}'_1. F(\varphi_1) \supset_2 \{F(\varphi'_1) \mid \varphi'_1 \in \vec{\phi}'_1\} \xrightarrow{e_2} F(\varphi_1 \supset_1 \vec{\phi}'_1)$
5. $\exists f \in \Phi_2 \rightarrow \Phi_1. \left(\exists e_2. \forall \varphi_2. \varphi_2 \xrightarrow{e_2} F(f(\varphi_2)) \right) \wedge \left(\exists e_2. \forall \varphi_2. F(f(\varphi_2)) \xrightarrow{e_2} \varphi_2 \right)$

In broad lines, an evidenced frame morphism F from \mathcal{EF}_1 to \mathcal{EF}_2 mostly ensures (first item) that if any two propositions that are logically connected by an evidence, so are their images through F , and guarantees (second to fourth item) the existence of a *uniform* evidence that witnesses that the image of the conjunction of two propositions (and similarly for other connectives) is the logical consequence of the conjunction of images of these propositions. The last condition in turns provides us with a constructive mean to relate any proposition in the codomain Φ_2 of the morphism with the image of a proposition in Φ_1 that is logically equivalent to it (which is, again, witnessed by a pair of *uniform* evidences that do not depend of the considered proposition).

This notion of morphism provides us with the definition of a category **EF**, whose objects are evidenced frames and whose structure can be further enriched, for instance to equip morphisms from \mathcal{EF}_1 to \mathcal{EF}_2 with a preorder relation $F \preceq G$, defined to hold when there exists evidence $e_2 \in E_2$ satisfying $\forall \varphi_1 \in \Phi_1. F(\varphi_1) \xrightarrow{e_2} G(\varphi_1)$. Again, for a more detailed introduction on evidenced frame we refer the reader to the corresponding paper [CMT21], but it is worth mentioning that any evidenced frame induces a tripos (via a uniform construction that does not depend on the considered evidenced frame), and that evidenced frames are complete with respect to triposes in the sense that the category **EF** is actually equivalent to the category **Trip** of triposes.

2.3 The induced evidenced frame $\mathcal{EF}_{\mathbf{HA2}}$

The interpretation of **HA2** given above induces an evidenced frame $\mathcal{EF}_{\mathbf{HA2}}$ whose definition simply reflects the structure of the interpretation: propositions are defined by saturated sets of terms, evidences are just λ -terms and the evidence relation is given by:

$$\psi \xrightarrow{t} \varphi \iff \forall u \in \psi. (t u) \in \varphi$$

This definition is reminiscent of the ordering relation on predicates induced by realizability interpretations, for instance to define triposes [Pit02].

Proposition 12 ($\mathcal{EF}_{\mathbf{HA2}}$). *The triple $(\mathbf{SAT}, \wedge, \cdot \dot{\rightarrow} \cdot)$ defines an evidenced frame.*

Proof. As hinted by the realizability interpretation in Definition 2, one can simply define

$$\rho \wedge \theta \triangleq \{t \in \Lambda : \exists u \in \rho. \exists v \in \theta. t \rightarrow_\beta (u, v)\} \quad \rho \supset \vec{\theta} \triangleq \{t \in \Lambda : \forall u \in \rho. \forall \theta \in \vec{\theta}. (t u) \in \theta\}$$

which both define saturated sets. Proving the existence of the required evidences is then an easy exercise of λ -calculus, which amounts to proving that the corresponding realizers exist. \square

⁴For simplicity reasons, we adopt here an extensional presentation of evidenced frames and thus of morphisms, see [CMT21] for further discussion on intensional/extensional aspects of evidenced frames and their morphisms.

Remark 13. Even though we started from an interpretation of second-order Heyting arithmetic, we should insist on the fact that an evidenced frame always provides us with a model of higher-logic. Indeed, since propositions are viewed through their semantic counterpart (here as saturated sets of terms), this allows us to define a (semantic) quantification over any set of propositions regardless of what the syntax of the language accounts for. In fact, should we have considered a concrete example for an even simpler theory (say an interpretation of intuitionistic propositional logic based on the simply-typed λ -calculus), the induced evidenced frame would still give us an interpretation of higher-order logic.

Conversely, the definition of an evidenced frame only specifies the minimal requirements for the interpretation to give such a model, but the language of propositions and evidences can actually be richer. For instance, in Section 4.3 we shall use a primitive data type for booleans, while in [CMT21] an evidenced frame is build from a computational system allowing for stateful computations.

3 Classical logic, the $\lambda\mu\tilde{\mu}$ -calculus and CPS translation

We present Curien-Herbelin’s $\lambda\mu\tilde{\mu}$ -calculus [CH00], which we use afterwards as the source calculus of a CPS translation. To illustrate the flexibility of evidenced frame, we chose on purpose to pick a call-by-value source calculus. Besides, to ease the later definition of the CPS translation, we opted for a sequent calculus as advocated by [DMMZ10, DMAJ16, MM13], hence the choice of Curien-Herbelin’s $\lambda\mu\tilde{\mu}$ -calculus⁵.

We first recall the syntax and operational semantics of the $\lambda\mu\tilde{\mu}$ -calculus, before defining a well-behaved CPS that translates it to the λ -calculus defined in Section 2.1.1. We will then study the corresponding evidenced frame $\mathcal{EF}_{\mu\tilde{\mu}}^{\text{bv}}$ and its relation to the CPS in Section 4. For the sake of simplicity, we will use the simply-typed version with pairs of the $\lambda\mu\tilde{\mu}$ -calculus. Quantifications (both first and second order) will be implicitly taken care of when we define the corresponding evidenced frame $\mathcal{EF}_{\mu\tilde{\mu}}^{\text{bv}}$ in the next section.

3.1 Curien-Herbelin’s $\lambda\mu\tilde{\mu}$ -calculus

We recall here the spirit of the Curien-Herbelin $\lambda\mu\tilde{\mu}$ -calculus [CH00]. The key notion of the $\lambda\mu\tilde{\mu}$ -calculus is the notion of *command*. A command $\langle t \parallel e \rangle$ can be understood as a state of an abstract machine, representing the evaluation of a *term* t (the program) against a co-proof e (the stack) that we call *context*. The syntax and reduction rules (parameterized over a subset of terms \mathcal{V} and a subset of evaluation contexts \mathcal{E}) are given in Figure 2, where $\tilde{\mu}x.c$ can be read as a context **let** $x = [\cdot]$ **in** c . The μ operator comes from Parigot’s $\lambda\mu$ -calculus [Par97], $\mu\alpha$ binds a context to a context variable α in the same way that $\tilde{\mu}x$ binds a proof to some proof variable x .

The $\lambda\mu\tilde{\mu}$ -calculus can be seen as a proof-as-program correspondence between sequent calculus and abstract machines. Right introduction rules correspond to typing rules for

⁵In fact, an even better choice when it comes to defining operational semantics and CPS translations of calculus could have been to rely upon Munch-Maccagnoni’s system L [MM13], which can be seen as a finer-grained variant of $\lambda\mu\tilde{\mu}$ -syntax where the evaluation order is driven by the polarity of terms. In such a syntax, continuation-passing style translations are really easy to define in that the operational semantics is that of an abstract machine specifying at each step whether the term or the evaluation context is given the priority. We mostly chose to stick to Curien-Herbelin’s $\lambda\mu\tilde{\mu}$ -calculus for the simplicity of the presentation, as its types system shares the same connectives than the one for HA2 (rather than their decompositions into linear logic). In fact, call-by-value/call-by-name variants of the $\lambda\mu\tilde{\mu}$ -calculus are expressible and correspond to a fixed choice of polarities when decomposing the different connectives.

Terms $t ::= x \mid \mu\alpha.c \mid \lambda x.t \mid (t, u)$ Contexts $e ::= \alpha \mid \tilde{\mu}x.c \mid t \cdot e \mid \tilde{\mu}(x, y).c$ Commands $c ::= \langle t \parallel e \rangle$	<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 40%; border-right: 1px solid black; padding: 5px;">$\langle \mu\alpha.c \parallel e \rangle$</td> <td style="padding: 5px;">$\triangleright^\dagger c[e/\alpha]$</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 5px;">$\langle t \parallel \tilde{\mu}x.c \rangle$</td> <td style="padding: 5px;">$\triangleright^\ddagger c[t/x]$</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 5px;">$\langle (t, u) \parallel \tilde{\mu}(x, y).c \rangle$</td> <td style="padding: 5px;">$\triangleright^\ddagger c[t/x][u/y]$</td> </tr> <tr> <td style="border-right: 1px solid black; padding: 5px;">$\langle \lambda x.t \parallel u \cdot e \rangle$</td> <td style="padding: 5px;">$\triangleright \langle u \parallel \tilde{\mu}x.\langle t \parallel e \rangle \rangle$</td> </tr> </table> <p style="text-align: right; margin-top: 5px;">\dagger where $e \in \mathcal{E}$, \ddagger where $t, u \in \mathcal{V}$</p>	$\langle \mu\alpha.c \parallel e \rangle$	$\triangleright^\dagger c[e/\alpha]$	$\langle t \parallel \tilde{\mu}x.c \rangle$	$\triangleright^\ddagger c[t/x]$	$\langle (t, u) \parallel \tilde{\mu}(x, y).c \rangle$	$\triangleright^\ddagger c[t/x][u/y]$	$\langle \lambda x.t \parallel u \cdot e \rangle$	$\triangleright \langle u \parallel \tilde{\mu}x.\langle t \parallel e \rangle \rangle$			
$\langle \mu\alpha.c \parallel e \rangle$	$\triangleright^\dagger c[e/\alpha]$											
$\langle t \parallel \tilde{\mu}x.c \rangle$	$\triangleright^\ddagger c[t/x]$											
$\langle (t, u) \parallel \tilde{\mu}(x, y).c \rangle$	$\triangleright^\ddagger c[t/x][u/y]$											
$\langle \lambda x.t \parallel u \cdot e \rangle$	$\triangleright \langle u \parallel \tilde{\mu}x.\langle t \parallel e \rangle \rangle$											
a) Syntax	b) Reduction rules											
<table style="width: 100%; border-collapse: collapse;"> <tr> <td colspan="2" style="text-align: center; padding: 5px;">$\frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle t \parallel e \rangle : (\Gamma \vdash \Delta)} \text{ (CvT)}$</td> </tr> <tr> <td style="padding: 5px;">$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \mid \Delta} \text{ (Ax}_r\text{)}$</td> <td style="padding: 5px;">$\frac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t : A \rightarrow B \mid \Delta} \text{ (}\rightarrow_r\text{)}$</td> <td style="padding: 5px;">$\frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta} \text{ (}\mu\text{)}$</td> </tr> <tr> <td style="padding: 5px;">$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta} \text{ (Ax}_l\text{)}$</td> <td style="padding: 5px;">$\frac{\Gamma \vdash u : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid u \cdot e : A \rightarrow B \vdash \Delta} \text{ (}\rightarrow_l\text{)}$</td> <td style="padding: 5px;">$\frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta} \text{ (}\tilde{\mu}\text{)}$</td> </tr> <tr> <td style="padding: 5px;">$\frac{c : (\Gamma, x : A, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}(x, y).c : A \wedge B \vdash \Delta} \text{ (}\wedge_l\text{)}$</td> <td colspan="2" style="padding: 5px;">$\frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \vdash u : B \mid \Delta}{\Gamma \vdash (t, u) : A \wedge B \mid \Delta} \text{ (}\wedge_r\text{)}$</td> </tr> </table> <p style="text-align: center;">(c) Typing rules</p>		$\frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle t \parallel e \rangle : (\Gamma \vdash \Delta)} \text{ (CvT)}$		$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \mid \Delta} \text{ (Ax}_r\text{)}$	$\frac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t : A \rightarrow B \mid \Delta} \text{ (}\rightarrow_r\text{)}$	$\frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta} \text{ (}\mu\text{)}$	$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta} \text{ (Ax}_l\text{)}$	$\frac{\Gamma \vdash u : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid u \cdot e : A \rightarrow B \vdash \Delta} \text{ (}\rightarrow_l\text{)}$	$\frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta} \text{ (}\tilde{\mu}\text{)}$	$\frac{c : (\Gamma, x : A, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}(x, y).c : A \wedge B \vdash \Delta} \text{ (}\wedge_l\text{)}$	$\frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \vdash u : B \mid \Delta}{\Gamma \vdash (t, u) : A \wedge B \mid \Delta} \text{ (}\wedge_r\text{)}$	
$\frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \mid e : A \vdash \Delta}{\langle t \parallel e \rangle : (\Gamma \vdash \Delta)} \text{ (CvT)}$												
$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \mid \Delta} \text{ (Ax}_r\text{)}$	$\frac{\Gamma, x : A \vdash t : B \mid \Delta}{\Gamma \vdash \lambda x.t : A \rightarrow B \mid \Delta} \text{ (}\rightarrow_r\text{)}$	$\frac{c : (\Gamma \vdash \Delta, \alpha : A)}{\Gamma \vdash \mu\alpha.c : A \mid \Delta} \text{ (}\mu\text{)}$										
$\frac{(\alpha : A) \in \Delta}{\Gamma \mid \alpha : A \vdash \Delta} \text{ (Ax}_l\text{)}$	$\frac{\Gamma \vdash u : A \mid \Delta \quad \Gamma \mid e : B \vdash \Delta}{\Gamma \mid u \cdot e : A \rightarrow B \vdash \Delta} \text{ (}\rightarrow_l\text{)}$	$\frac{c : (\Gamma, x : A \vdash \Delta)}{\Gamma \mid \tilde{\mu}x.c : A \vdash \Delta} \text{ (}\tilde{\mu}\text{)}$										
$\frac{c : (\Gamma, x : A, y : B \vdash \Delta)}{\Gamma \mid \tilde{\mu}(x, y).c : A \wedge B \vdash \Delta} \text{ (}\wedge_l\text{)}$	$\frac{\Gamma \vdash t : A \mid \Delta \quad \Gamma \vdash u : B \mid \Delta}{\Gamma \vdash (t, u) : A \wedge B \mid \Delta} \text{ (}\wedge_r\text{)}$											

Figure 2: The simply-typed $\lambda\mu\tilde{\mu}$ -calculus with pairs

proofs, while left introduction are seen as typing rules for evaluation contexts. In contrast with Gentzen’s original presentation of sequent calculus, the type system of the $\lambda\mu\tilde{\mu}$ -calculus explicitly identifies at any time which formula is being worked on. In a nutshell, this presentation distinguishes between three kinds of sequents: sequents of the form $\Gamma \vdash t : A \mid \Delta$ for typing terms, where the focus is put on the (right) formula A ; sequents of the form $\Gamma \mid e : A \vdash \Delta$ for typing contexts, where the focus is put on the (left) formula A ; sequents of the form $c : (\Gamma \vdash \Delta)$ for typing commands, where no focus is set. In a right (resp. left) sequent $\Gamma \vdash t : A \mid \Delta$, the singled out formula⁶ A reads as the conclusion “where the proof shall continue” (resp. hypothesis “where it happened before”).

Regarding the reduction rules, observe that if \mathcal{V} and \mathcal{E} are not restricted enough, these rules admit a critical pair:

$$c[\tilde{\mu}x.c'/\alpha] \triangleleft \langle \mu\alpha.c \parallel \tilde{\mu}x.c' \rangle \triangleright c'[\mu\alpha.c/a].$$

The difference between call-by-name and call-by-value can be characterized by how this critical pair is solved, by defining \mathcal{V} and \mathcal{E} in such a way that the two rules do not overlap. This justifies the definition of a subcategory V of proofs, that we call *values*, and of the dual subset E of contexts that we call *co-values* (following Downen and Ariola’s denomination [DA14]):

$$\textbf{Values} \quad V ::= x \mid \lambda x.t \mid (V_1, V_2) \qquad \textbf{Co-values} \quad E ::= \alpha \mid t \cdot e$$

The call-by-name evaluation strategy amounts to the case where $\mathcal{V} \triangleq \text{Terms}$ and $\mathcal{E} \triangleq \text{Co-values}$, while call-by-value corresponds to $\mathcal{V} \triangleq \text{Values}$ and $\mathcal{E} \triangleq \text{Contexts}$. In the sequel, we

⁶This formula is often referred to as the formula in the *stoup*, a terminology due to Girard.

will focus on the latter and we write $\triangleright_{\text{bv}}$ for the corresponding reduction steps and \rightarrow_{bv} for its reflexive-transitive closure. Since the by-value reduction rule for $\tilde{\mu}(x, y).c$ only computes in front of a pair of values, we restrict the syntax to these pairs and define pairs of (non-evaluated) terms through the shorthand (which simulate the left-to-right opening of such pairs):

$$(t, u) \triangleq \mu\alpha. \langle t \parallel \tilde{\mu}x. \langle u \parallel \tilde{\mu}y. \langle (x, y) \parallel \alpha \rangle \rangle \rangle$$

Readers more accustomed to the λ -calculus may wonder why the syntax of terms does not include the usual application (which we write $t@u$ below), but this can be expressed as a macro, as well as the **let** $\cdot = \cdot$ **in** \cdot construct:

$$t@u \triangleq \mu\alpha. \langle t \parallel u \cdot \alpha \rangle \quad \text{let } x = t \text{ in } u \triangleq \mu\alpha. \langle t \parallel \tilde{\mu}x. \langle u \parallel \alpha \rangle \rangle$$

In particular, the β -reduction is simulated by the reduction of $\lambda\mu\tilde{\mu}$ commands, since if $t, u \in \Lambda$ are such that $t \rightarrow_{\beta} u$, then for any stack e we have $\langle t \parallel e \rangle \rightarrow_{\beta}^* \langle u \parallel e \rangle$. In our setting, we can observe that the relation $\triangleright_{\text{bv}}$ induces a (weak-head) call-by-value evaluation strategy for the application, since if u reduces (in front of any context) to value V , we have:

$$\langle \lambda x. t@u \parallel e \rangle \rightarrow_{\text{bv}} \langle u \parallel \tilde{\mu}x. \langle t \parallel e \rangle \rangle \rightarrow_{\text{bv}} \langle V \parallel \tilde{\mu}x. \langle t \parallel e \rangle \rangle \rightarrow_{\text{bv}} \langle t[V/x] \parallel e \rangle$$

that is analogous to the expected reduction thread in a call-by-value evaluated λ -calculus:

$$(\lambda x. t) u \rightarrow_{\beta} \text{let } x = u \text{ in } t \rightarrow_{\beta} \text{let } x = V \text{ in } t \rightarrow_{\beta} t[V/x]$$

Readers familiar with control operators may observe that we can implement **call/cc** as well as continuations in this calculus as follows:

$$\text{call/cc} \triangleq \lambda x. \mu\alpha. \langle x \parallel \mathbf{k}_{\alpha} \cdot \alpha \rangle \quad \mathbf{k}_e \triangleq \lambda y. \mu_{-}. \langle y \parallel e \rangle$$

Forgetting a minute about the call-by-value restriction, we can check that these definitions yield the expected computational behavior of **call/cc**: in front of a stack $t \cdot e$, it catches the context e thanks to the $\mu\alpha$ binder and reduces as follows:

$$\langle \text{call/cc} \parallel t \cdot e \rangle = \langle \lambda x. \mu\alpha. \langle x \parallel \mathbf{k}_{\alpha} \cdot \alpha \rangle \parallel t \cdot e \rangle \triangleright \langle \mu\alpha. \langle t \parallel \mathbf{k}_{\alpha} \cdot \alpha \rangle \parallel e \rangle \triangleright \langle t \parallel \mathbf{k}_e \cdot e \rangle$$

In turns, in front of a stack $u \cdot e'$, the continuation \mathbf{k}_e will now catch the context e' and throw it away to restore the former context e :

$$\langle \mathbf{k}_e \parallel u \cdot e' \rangle = \langle \lambda y. \mu_{-}. \langle y \parallel e \rangle \parallel u \cdot e' \rangle \triangleright \langle \mu_{-}. \langle u \parallel e \rangle \parallel e' \rangle \triangleright \langle u \parallel e \rangle$$

From a logical perspective, it is an easy exercise to check that **call/cc** can be typed with Peirce's law.

3.2 Continuation-passing style translation of the call-by-value $\lambda\mu\tilde{\mu}$ -calculus

We define a CPS translation for the call-by-value variant of the $\lambda\mu\tilde{\mu}$ -calculus introduced previously which is analogous to the translations in [CH00, Miqu17]. The definition of the translation can be mechanically derived from the operational semantics of the calculus, following the methodology of Danvy's semantic artifacts described in [ADH⁺12, Miqu17], or the decomposition of the $\lambda\mu\tilde{\mu}$ -calculus into Munch-Maccagnoni's system L [MM13]. As is usual in

call-by-value, the translation is defined on three layers, reflecting the three syntactic categories at play: $[\cdot]_t$ on terms, $[\cdot]_e$ on contexts and $[\cdot]_v$ on values.

$$\left. \begin{array}{l} [\langle t \parallel e \rangle]_c \triangleq [t]_t [e]_e \\ [\mu\alpha.c]_t \triangleq \lambda e.(\lambda\alpha.[c]_c) e \\ [V]_t \triangleq \lambda e.e [V]_v \\ [u \cdot e]_e \triangleq \lambda V.V [u]_t [e]_e \\ [\tilde{\mu}x.c]_e \triangleq \lambda V.(\lambda x.[c]_c) V \end{array} \right| \begin{array}{l} [\tilde{\mu}(x,y).c]_e \triangleq \lambda V.\mathbf{let} (x,y) = V \mathbf{in} [c]_c \\ [\alpha]_e \triangleq \alpha \\ [x]_v \triangleq x \\ [(V_1, V_2)]_v \triangleq ([V_1]_v, [V_2]_v) \\ [\lambda x.t]_v \triangleq \lambda u.e.u (\lambda x.[t]_t e) \end{array}$$

The computational translation induces the following translation on types:

$$\begin{array}{ll} \llbracket A \rrbracket_t \triangleq \llbracket A \rrbracket_e \rightarrow \mathcal{R} & \llbracket A \rightarrow B \rrbracket_v \triangleq \llbracket A \rrbracket_t \rightarrow \llbracket B \rrbracket_e \rightarrow \mathcal{R} \\ \llbracket A \rrbracket_e \triangleq \llbracket A \rrbracket_v \rightarrow \mathcal{R} & \llbracket A \wedge B \rrbracket_v \triangleq \llbracket A \rrbracket_v \wedge \llbracket B \rrbracket_v \\ & \llbracket X \rrbracket_v \triangleq X \end{array}$$

where \mathcal{R} is the return type of continuations, usually defined as $\mathcal{R} \triangleq \perp$. This translation extends naturally to contexts, where the translation of Γ is defined in terms of values while Δ is translated in terms of contexts:

$$\llbracket \Gamma, x : A \rrbracket_v \triangleq \llbracket \Gamma \rrbracket_v, x : [A]_v \qquad \llbracket \Delta, \alpha : A \rrbracket_e \triangleq \llbracket \Delta \rrbracket_e, \alpha : \llbracket A \rrbracket_e$$

Remark 14. Usually, the return type \mathcal{R} is chosen to be some specific formula of the target language (here **HA2**). Actually, we can even consider an even more general settings, where we extend the language of formulas with a new constant \mathcal{R} for which we only have to provide its realizability interpretation as a saturated set of terms (*i.e.* $|\mathcal{R}| \in \mathbf{SAT}$). Since no typing rule is provided for \mathcal{R} , it can be understood as a generalization of \perp : any derivation of \mathcal{R} can be turned into a derivation of \perp (and vice-versa), but contrarily to $|\perp| = |\forall X.X| = \emptyset$, the semantic interpretation of \mathcal{R} can be chosen not to be empty.

The translation of terms, contexts and commands is sound with respect both to types and computations⁷, as shown by the following propositions:

Proposition 15 ([CH00, Miq17]). *For any contexts Γ and Δ , we have*

1. *if $\Gamma \vdash t : A \mid \Delta$ then $\llbracket \Gamma \rrbracket_v, \llbracket \Delta \rrbracket_e \vdash [t]_t : \llbracket A \rrbracket_t$*
2. *if $\Gamma \mid e : A \vdash \Delta$ then $\llbracket \Gamma \rrbracket_v, \llbracket \Delta \rrbracket_e \vdash [e]_e : \llbracket A \rrbracket_e$*
3. *if $c : \Gamma \vdash \Delta$ then $\llbracket \Gamma \rrbracket_v, \llbracket \Delta \rrbracket_e \vdash [c]_c : \mathcal{R}$*

Proposition 16 (Simulation). *For any c, c' , we have: $c \rightarrow_{\text{bv}} c'$ if and only if $[c]_c \rightarrow_{\beta} [c']_c$.*

Proof. The direct implication is standard and proven by induction on $\triangleright_{\text{bv}}$ [Miq17]. The “only if” part is proven by contradiction, by considering c and c' with the shortest reduction path $[c]_c \rightarrow_{\beta} [c']_c$ possible, then reasoning by induction on c . \square

4 CPS translation of realizers

We are now ready to examine Oliva and Streicher result through the lens of evidenced frames, and investigate the main question of this paper.

⁷To be even more precise, we could restrict ourselves to a weak-head call-by-name evaluation strategy with the same result.

4.1 A call-by-value classical realizability interpretation

We begin by defining the evidenced frame induced by the realizability interpretation that the call-by-value $\lambda\mu\tilde{\mu}$ -calculus yields. The structure of this interpretation differs from the (intuitionistic) interpretation introduced in Section 2.1.2 mostly for two reasons: a) it is a classical (*à la* Krivine) interpretation, and b) it is based on a call-by-value calculus.

As in intuitionistic realizability, every formula A is interpreted in classical realizability as a set $|A|_{\mathfrak{t}}$ of terms (the realizers) that share a common computational behavior determined by the structure of the formula A [Kri04]. However the difference between intuitionistic and classical realizability is that in the latter, the set of realizers of A is defined indirectly, that is from a set $\|A\|_{\mathfrak{e}}$ of contexts that are intended to challenge the truth of A . Intuitively, the set $\|A\|_{\mathfrak{e}}$ (which we shall call the *falsity value* of A) can be understood as the set of all possible counter-arguments to the formula A . In this framework, a program realizes the formula A if and only if it is able to defeat all the attempts to refute A by a context in $\|A\|_{\mathfrak{e}}$.

When defining such an interpretation on a call-by-value calculus, the falsity value itself is in fact defined in terms of a more primitive notions of truth values of values [MM09]. This set, which we write $|A|_{\mathfrak{v}}$, can be understood as the values that any test challenging A should accept as a valid answer.

The last ingredient peculiar to Krivine realizability is the fact that realizability interpretations are parameterized by a set of commands, the *pole*, which intuitively represents the valid computations.

Definition 17 (Pole). A pole $\perp\!\!\!\perp$ is a saturated set of commands, *i.e.* a set such that if $c \rightarrow_{\text{bv}} c'$ and $c' \in \perp\!\!\!\perp$ then $c \in \perp\!\!\!\perp$. It should be seen as a set of commands whose computations end with success.

Definition 18 (Orthogonal). Given a pole $\perp\!\!\!\perp$, we define $A^{\perp\!\!\!\perp}$ to be the orthogonal of A : if A is a set of terms (resp. contexts), it is the set of contexts e (resp. terms t) such that $\langle t \| e \rangle \in \perp\!\!\!\perp$.

The complete definition of the realizability interpretation based on the call-by-value $\lambda\mu\tilde{\mu}$ -calculus introduced before would require once again to define the appropriate notions of valuations, adequate judgements, etc... Such definitions can be found for analogous interpretation in [MM09, Lep16, Miqu20], and will be somewhat hidden here between the lines of the definition of the induced evidenced frame $\mathcal{EF}_{\mu\tilde{\mu}}^{\text{bv}}$. Let us however explain how connectives are interpreted in terms of values. The aforementioned sets of truth and falsity values are then defined by orthogonality to each others, *i.e.* $|A|_{\mathfrak{t}} = \|A\|_{\mathfrak{e}}^{\perp\!\!\!\perp}$ and $\|A\|_{\mathfrak{e}} = |A|_{\mathfrak{v}}^{\perp\!\!\!\perp}$, which in particular implies that $|A|_{\mathfrak{v}} \subseteq |A|_{\mathfrak{t}}$. For A and B two closed formulas, values realizing implication (resp. the conjunction) are the expected ones, namely functions (resp. pairs):

$$|A \rightarrow B|_{\mathfrak{v}} = \{\lambda x.t : \forall V \in |A|_{\mathfrak{v}} : t[V/x] \in |B|_{\mathfrak{t}}\} \quad |A \wedge B|_{\mathfrak{v}} = \{(V_1, V_2) : V_1 \in |A|_{\mathfrak{v}} \wedge V_2 \in |B|_{\mathfrak{v}}\}$$

Recall that even if the type system introduced earlier does not include quantifiers, we can nevertheless define them through their semantic interpretation in terms of values, namely as an intersection of primitive truth values (where ρ should be the appropriate notion of valuation for this setting, and X ranges over propositions):

$$|\forall x.A|_{\mathfrak{v}}^{\rho} = \bigcap_{n \in \mathbb{N}} |A|_{\mathfrak{v}}^{\rho, x \rightarrow n} \quad |\forall X.A|_{\mathfrak{v}}^{\rho} = \bigcap_{F \in \mathcal{P}(V)} |A|_{\mathfrak{v}}^{\rho, X \mapsto F}$$

The corresponding evidenced frame thus uses sets of values as propositions, terms as evidences (we write \mathcal{T} for the set of closed terms) and the following evidence relation

$$\varphi \xrightarrow{t} \psi \iff \forall V \in \varphi : t @ V \in \psi^{\perp\!\!\!\perp}$$

Theorem 19 ($\mathcal{EF}_{\mu\bar{\mu}}^{\text{bv}}$). *The triple $\mathcal{EF}_{\mu\bar{\mu}}^{\text{bv}} \triangleq (\mathcal{P}(V), \mathcal{T}, \cdot \dot{\rightarrow} \cdot)$ defines an evidenced frame.*

4.2 CPS as a morphism

We now wish to investigate whether the CPS translation defined in Section 3.2 defines an evidenced frame morphism from $\mathcal{EF}_{\mu\bar{\mu}}^{\text{bv}}$ to $\mathcal{EF}_{\text{HA2}}$. To be precise, recall that so far our definitions for the interpretations of the source and destination of the translation leave us two degrees of liberty: the choice of pole in $\mathcal{EF}_{\mu\bar{\mu}}^{\text{bv}}$, which we shall write $\perp\!\!\!\perp_s$; and the realizability interpretation of the return type \mathcal{R} , which we write $|\mathcal{R}| = \perp\!\!\!\perp_d$. As we shall see in Section 4.3, we cannot build a morphism that works for any choice for these parameters and these two evidenced frames. However, we investigate the following questions: given an interpretation $\perp\!\!\!\perp_s$, can we find a pole $\perp\!\!\!\perp_d$ for \mathcal{R} such that $[\cdot]_{\mathcal{V}}$ defines an evidenced frame morphism? Reciprocally, given a pole $\perp\!\!\!\perp_d$, can we find an appropriate $\perp\!\!\!\perp_s$?

4.2.1 The forward evidenced frame

We first tackle the first problem, that is to define from a fixed pole $\perp\!\!\!\perp_s$ in the source an interpretation $\perp\!\!\!\perp_d$ for \mathcal{R} such that the CPS induces a morphism, we can restrict ourselves to consider instead the image of $\mathcal{EF}_{\mu\bar{\mu}}^{\text{bv}}$ through the CPS translation as an evidenced frame itself (as is done in [OS08]). To begin with, we define the interpretation of the return type \mathcal{R} as the saturation of the image of the pole $\perp\!\!\!\perp_s$ through the CPS:

$$\perp\!\!\!\perp_d \triangleq \{t : \exists c \in \perp\!\!\!\perp_s. t \rightarrow_{\beta} [c]_c\}$$

We take as propositions the images of sets of values through the translation of values $[\cdot]_{\mathcal{V}}$, i.e. $\Phi_{\text{fw}} \triangleq [\mathcal{P}(\text{values})]_{\mathcal{V}}$. Similarly, we take evidences to be translated terms: $E_{\text{fw}} \triangleq [\text{terms}]_{\mathcal{T}}$. As for the evidence relation, we can see it as the image of the evidence relation in $\mathcal{EF}_{\mu\bar{\mu}}^{\text{bv}}$ through the translation (up to some technical details), that is:

$$\varphi \xrightarrow{t}_{\text{fw}} \psi \triangleq \forall V \in \phi. \forall e \in \text{contexts}. [(\forall [V']_{\mathcal{V}} \in \psi : [V']_{\mathcal{T}} [e]_e \in \perp\!\!\!\perp_d) \implies (t \# V) [e]_e \in \perp\!\!\!\perp_d]$$

where $a \# b \triangleq \lambda \alpha. a (\lambda V. V (\lambda e. e b) \alpha)$. This operator is solely motivated by technical reasons, in order to satisfy the equation $[t @ V]_{\mathcal{T}} = [t]_{\mathcal{T}} \# [V]_{\mathcal{V}}$.

Theorem 20. $\mathcal{EF}_{\text{fw}} = (\Phi_{\text{fw}}, E_{\text{fw}}, \cdot \dot{\rightarrow}_{\text{fw}} \cdot)$ defines an evidenced frame and the map $F : V \mapsto [V]_{\mathcal{V}}$ induces a morphism from $\mathcal{EF}_{\mu\bar{\mu}}^{\text{bv}}$ to \mathcal{EF}_{fw} .

Proof. The first part mostly relies on the observation that $[\cdot]_{\mathcal{T}}$ is injective and admits an inverse (see the proof of Prop. 22). The evidences that make the morphism well-behaved w.r.t. the connectives are just obtained as the translation $[\cdot]_{\mathcal{T}}$ of the corresponding evidences in $\mathcal{EF}_{\mu\bar{\mu}}^{\text{bv}}$. \square

As the proof of Theorem 20 shows, not only does the CPS translation induce a morphism, but it does so while preserving evidences. In other words, in this setting, we can actually conclude that the CPS translation does preserve realizers. Observe nonetheless that this is almost a tautology, in that evidences in \mathcal{EF}_{fw} are by definition translated terms. In particular, if \mathcal{EF}_{fw} defines a model for **HA2**, it is by nature very different from the one that $\mathcal{EF}_{\text{HA2}}$ provides.

4.2.2 The backward evidenced frame

We now tackle the other question, that is, we consider a fixed interpretation $\perp\!\!\!\perp_d \in \mathbf{SAT}$ of the return type \mathcal{R} in the realizability interpretation. To define a pole in the source of the

translation, we simply pick the sets $\perp_{\mathfrak{s}} = \{c : [c]_c \in \perp_{\mathfrak{d}}\}$ of commands whose translations belong to $\perp_{\mathfrak{d}}$. This indeed defines a saturated set of commands, since the translation $[\cdot]_c$ preserves computation and $\perp_{\mathfrak{d}}$ is itself a saturated set of terms.

For technical reasons, we now take propositions to be pairs made of the translation of a proposition in $\mathcal{E}\mathcal{F}_{\mu\tilde{\mu}}^{\text{bv}}$ and the translation of its orthogonal set: $\Phi_{\text{bw}} = \{([S]_{\mathfrak{V}}, [S^{\perp}]_{\mathfrak{e}}) : S \in \mathcal{P}(\mathcal{V})\}$. Evidences are again defined as translation of terms $E_{\text{bw}} = [\text{terms}]_{\mathfrak{t}}$ while the evidence relation is given by:

$$(\varphi^V, \varphi^e) \xrightarrow{\text{t}}_{\text{bw}} (\psi^V, \psi^e) \iff \forall V \in \varphi^V. \forall e \in \varphi^e. (t \# V) e \in \perp_{\mathfrak{d}}$$

Theorem 21. *The triple $\mathcal{E}\mathcal{F}_{\text{bw}} = (\Phi_{\text{bw}}, E_{\text{bw}}, \xrightarrow{\text{t}}_{\text{bw}} \cdot)$ defines an evidenced frame and the map $F : V \mapsto ([V]_{\mathfrak{V}}, [V^{\perp}]_{\mathfrak{e}})$ induces a morphism from $\mathcal{E}\mathcal{F}_{\mu\tilde{\mu}}^{\text{bv}}$ to $\mathcal{E}\mathcal{F}_{\text{bw}}$.*

The proof is analogous to the one of Theorem 20 (in particular, realizers are also trivially preserved in this case), and actually unveils that $\mathcal{E}\mathcal{F}_{\text{fw}}$ and $\mathcal{E}\mathcal{F}_{\text{bw}}$ are essentially the same.

Proposition 22. *For any term t , we have: $(\varphi^V, \varphi^e) \xrightarrow{[\text{t}]}_{\text{bw}} (\psi^V, \psi^e) \iff \varphi^V \xrightarrow{[\text{t}]}_{\text{fw}} \psi^V$.*

Proof. We prove that both statement are equivalent to $[\varphi]_{\mathfrak{V}}^{-1} \xrightarrow{\text{t}}_{\text{bv}} [\psi]_{\mathfrak{V}}^{-1}$, where $[\cdot]_{\mathfrak{V}}^{-1}$ is the inverse of the injective map $[\cdot]_{\mathfrak{V}}$. This relies in turn on the definitions of the poles $\perp_{\mathfrak{d}}$, $\perp_{\mathfrak{s}}$ and on the fact that the CPS translation simulates computations (Proposition 16). \square

4.3 A counter-example

We shall now give an example of a pole $\perp_{\mathfrak{s}}$ together with a term that is a realizer in the interpretation based on the $\lambda\mu\tilde{\mu}$ -calculus, but whose translation is not a realizer of the translated formula. We follow here the lines of Oliva-Streicher's presentation of Krivine realizability through a CPS translation, in particular we assume that an interpretation of the return type $\perp_{\mathfrak{d}}$ is given and we define the pole $\perp_{\mathfrak{s}} \triangleq \{c : [c]_c \in \perp_{\mathfrak{d}}\}$ consequently. In the rest of this section, we write $t \Vdash_{\mathfrak{s}} A$ when t is a $\lambda\mu\tilde{\mu}$ -term which realizes A in the source, and $t \Vdash_{\mathfrak{d}} A$ when t is a λ -term realizing A in the destination.

To give a simple example, we will extend thereafter the definitions of the source and destination of the CPS to include a type \mathbb{B} of booleans, whose translation at the level of types will be given by: $[\mathbb{B}]_{\mathfrak{t}} = (\mathbb{B} \rightarrow \mathcal{R}) \rightarrow \mathcal{R}$, and we will present a term t such that $t \Vdash_{\mathfrak{s}} \mathbb{B}$ but $[t]_{\mathfrak{t}} \not\Vdash_{\mathfrak{d}} [\mathbb{B}]_{\mathfrak{t}}$. Let us briefly guide the reader through the rationale of our construction. To prove that $[t]_{\mathfrak{t}} \not\Vdash_{\mathfrak{d}} [\mathbb{B}]_{\mathfrak{t}}$, we shall exhibit a continuation $k \Vdash_{\mathfrak{d}} \mathbb{B} \rightarrow \mathcal{R}$ such that $[t]_{\mathfrak{t}} k \notin \perp_{\mathfrak{d}}$. By definition, if $t \Vdash_{\mathfrak{s}} \mathbb{B}$ in particular for any context $e \in \|\mathbb{B}\|_{\mathfrak{e}}$, the fact that $\langle t \parallel e \rangle \in \perp_{\mathfrak{s}}$ means that $[t]_{\mathfrak{t}} [e]_{\mathfrak{e}} \in \perp_{\mathfrak{d}}$. Therefore we should look for a continuation that is *not* in the image of the CPS translation. Besides, $[t]_{\mathfrak{t}} k$ should *not* reduce to kb where b is a boolean, since otherwise the fact that $k \Vdash_{\mathfrak{d}} \mathbb{B} \rightarrow \mathcal{R}$, $b \Vdash_{\mathfrak{d}} \mathbb{B}$ and antireduction would also entail that $[t]_{\mathfrak{t}} k \in \perp_{\mathfrak{d}}$. For these reasons, we will pick an inert constant κ for t , define the pole $\perp_{\mathfrak{d}}$ to enforce $\kappa \Vdash_{\mathfrak{d}} \mathbb{B}$ and select a continuation k that will be syntactically discriminated for any translated context.

4.3.1 Extension with booleans

We briefly review the extensions of the $\lambda\mu\tilde{\mu}$ -calculus and of the λ -calculus to include a type \mathbb{B} of booleans in the corresponding languages of formulas. We first extend the syntax and reduction rules of the $\lambda\mu\tilde{\mu}$ -calculus to account for boolean values tt and ff , a context to eliminate booleans and an inert term κ :

Terms	$t ::= \dots \mid \kappa$	$\langle \text{tt} \parallel \tilde{\mu}\mathbb{B}.[c_1 \mid c_2] \rangle \triangleright_{\text{bv}} c_1$
Values	$V ::= \dots \mid \text{tt} \mid \text{ff}$	$\langle \text{ff} \parallel \tilde{\mu}\mathbb{B}.[c_1 \mid c_2] \rangle \triangleright_{\text{bv}} c_2$
Contexts	$e ::= \dots \mid \tilde{\mu}\mathbb{B}.[c_1 \mid c_2]$	

These terms can be typed with the following rules:

$$\frac{c_1 : (\Gamma \vdash \Delta) \quad c_2 : (\Gamma \vdash \Delta)}{\Gamma \mid \tilde{\mu}\mathbb{B}.[c_1 \mid c_2] : \mathbb{B} \vdash \Delta} \text{ (}\mathbb{B}_i\text{)} \quad \frac{}{\Gamma \vdash \mathbf{tt} : \mathbb{B} \mid \Delta} \text{ (tt)} \quad \frac{}{\Gamma \vdash \mathbf{ff} : \mathbb{B} \mid \Delta} \text{ (ff)}$$

Similarly, the λ -calculus that serves as a destination of the CPS translation can be extended with booleans and an inert constant, for which we use the same notations:

$$t, u ::= \dots \mid \kappa \mid \mathbf{ff} \mid \mathbf{tt} \mid \text{if } t \text{ then } u \text{ else } v \quad \Bigg| \quad \overline{\text{if } \mathbf{tt} \text{ then } u \text{ else } v}_{\triangleright_\beta} \quad \overline{\text{if } \mathbf{ff} \text{ then } u \text{ else } v}_{\triangleright_\beta}$$

Once again, we can give typing rules to the terms computing with booleans:

$$\frac{\Gamma \vdash b : \mathbb{B} \quad \Gamma \vdash t : A \quad \Gamma \vdash u : A}{\Gamma \vdash \text{if } b \text{ then } t \text{ else } u : A} \text{ (if)} \quad \frac{}{\Gamma \vdash \mathbf{tt} : \mathbb{B}} \text{ (tt)} \quad \frac{}{\Gamma \vdash \mathbf{ff} : \mathbb{B}} \text{ (ff)}$$

The CPS translation is then naturally extended (following what is already defined for values and base types):

$$\llbracket \mathbb{B} \rrbracket_{\mathcal{V}} \triangleq \mathbb{B} \quad \mid \quad \llbracket \mathbf{tt} \rrbracket_{\mathcal{V}} \triangleq \mathbf{tt} \quad \llbracket \mathbf{ff} \rrbracket_{\mathcal{V}} \triangleq \mathbf{ff} \quad \mid \quad \llbracket \tilde{\mu}\mathbb{B}.[c_1 \mid c_2] \rrbracket_{\mathcal{E}} \triangleq \lambda b. \text{if } b \text{ then } [c_1]_{\mathcal{C}} \text{ else } [c_2]_{\mathcal{C}} \quad \mid \quad \llbracket \kappa \rrbracket_{\mathcal{T}} \triangleq \kappa$$

It is straightforward to verify that the different properties of the CPS translation still hold for the extended calculi.

4.3.2 An untranslatable realizer

It now only remains to extend the realizability interpretation, in the $\lambda\mu\tilde{\mu}$ -calculus we simply define the primitive truth value of \mathbb{B} as the set containing the two boolean values

$$\llbracket \mathbb{B} \rrbracket_{\mathcal{V}} \triangleq \{\mathbf{tt}, \mathbf{ff}\}$$

while for the intuitionistic interpretation based on the target calculus, we consider the set of terms reducing to a boolean value:

$$\llbracket \mathbb{B} \rrbracket \triangleq \{t \longrightarrow_\beta \mathbf{tt}\} \cup \{t \longrightarrow_\beta \mathbf{ff}\}$$

These definitions make the typing rules for booleans adequate with the corresponding realizability interpretations.

Finally, we define the pole (and thus the realizability interpretations), which will allow us to complete the construction of the counter-example:

$$\perp\!\!\!\perp_{\mathcal{D}} \triangleq \{u \in \Lambda : u \longrightarrow_\beta [t]_{\mathcal{T}} \lambda x.v \text{ for some term } v\} \quad \perp\!\!\!\perp_{\mathcal{S}} \triangleq \{c : [c]_{\mathcal{C}} \in \perp\!\!\!\perp_{\mathcal{D}}\}$$

Lemma 23. *We have $\kappa \Vdash_{\mathcal{S}} \mathbb{B}$*

Proof. Since any context e translates into a function of the shape $\lambda x.v$, for any context $e \in \llbracket \mathbb{B} \rrbracket_{\mathcal{E}}$, we have in particular $\llbracket \langle \kappa \parallel e \rangle \rrbracket_{\mathcal{C}} = \kappa [e]_{\mathcal{E}} \in \perp\!\!\!\perp_{\mathcal{D}}$, and then $\langle \kappa \parallel e \rangle \in \perp\!\!\!\perp_{\mathcal{S}}$. \square

Lemma 24. *Let us fix $p \in \perp\!\!\!\perp_{\mathcal{D}}$, and define $\mathbf{k} \triangleq (\lambda x.x)\lambda b.p$. Then it holds that $\mathbf{k} \Vdash_{\mathcal{D}} \mathbb{B} \rightarrow \mathcal{R}$.*

Proof. Recall that we define $\llbracket \mathcal{R} \rrbracket = \perp\!\!\!\perp_{\mathcal{D}}$ and $\llbracket \mathbb{B} \rrbracket = \{t : t \longrightarrow_\beta b \text{ with } b \in \mathbb{B}\}$. Let $b \in \mathbb{B}$, and t be such that $t \longrightarrow_\beta b$. We have

$$\mathbf{k}t = ((\lambda x.x)\lambda b.p)t \longrightarrow_\beta (\lambda b.p)t \longrightarrow_\beta p \in \perp\!\!\!\perp_{\mathcal{D}}$$

hence the fact that $\mathbf{k} \Vdash_{\mathcal{D}} \mathbb{B} \rightarrow \mathcal{R}$ by antireduction. \square

We can now check that :

Proposition 25. *We have $[\kappa]_t \not\vdash_d \llbracket \mathbb{B} \rrbracket_t$.*

Proof. We have $[\kappa]_t \mathbf{k} = \kappa \mathbf{k}$. Since this term does not reduce (recall that β is a weak-head reduction relation) and \mathbf{k} is not of the shape $\lambda x.v$, it does not belong to $\perp\!\!\!\perp_d$. The result then follows from Lemma 24. \square

Nonetheless, this example is not entirely satisfying in that the subsequent realizability interpretation is not coherent:

Proposition 26. *The closed term $t_\perp \triangleq \mu\alpha.\langle \kappa \parallel \tilde{\mu}x.\langle x \parallel \alpha \rangle \rangle$ satisfies $t_\perp \Vdash_s \perp$.*

Proof. Indeed, for any context e we have:

$$\langle t_\perp \parallel e \rangle \triangleright_{bv} \langle \kappa \parallel \tilde{\mu}x.\langle x \parallel e \rangle \rangle$$

Since $[\tilde{\mu}x.\langle x \parallel e \rangle]_e = \lambda x.(\lambda k.kx)[e]_e$, in particular we have $\kappa[\tilde{\mu}x.\langle x \parallel e \rangle]_e \in \perp\!\!\!\perp_d$ and thus $\langle t_\perp \parallel e \rangle \in \perp\!\!\!\perp_s$ by antireduction. \square

5 Conclusion

As mentioned in the introduction, the motivation behind this work was twofold. On the one hand, from a methodological perspective, this work also was a pretext to experiment with evidenced frames as a tool to reason on realizability interpretation. As such, the result of this experimentation turns out to be pretty satisfactory, for they have shown to be very helpful during the research process to identify the key ingredients necessary for the different results exposed. In particular, they provided us with a precious algebraic viewpoint to avoid losing ourselves in the implementation details of realizers and translations. In particular, we could easily reproduce Oliva-Streicher's construction while using different calculi.

On the other hand, from a logical perspective, we were actually really interested in the question raised in the title. In that regards, we gave here partial answers, by giving some sufficient conditions, namely the restriction to the backward and forward evidenced frame, for CPS translations to be well-behaved with respect to the realizability interpretations; as well as an example of two realizability interpretations where the translation does not preserve realizers. Many interesting questions remain to be explored in that direction: are there counter-examples that do not require an incoherent pole? What can be said in general of realizers that are compatible with the CPS translation? In particular, which are the terms that are *always* compatible (*i.e.* regardless of the choice of $\perp\!\!\!\perp_d$ and $\perp\!\!\!\perp_s$) with a CPS? In particular, it would be very interesting to wonder whether *universal* realizers (*i.e.* compatible with any pole) are soundly CPS translated. This can be shown for simple data types (for instance boolean or natural numbers) by means of specification techniques [GM16a, GM16b] (but more details on this would drive us out of the scope of this paper), and is still to be investigated for more complex types (functions, etc.).

It is worth mentioning that all these questions go beyond the sole case of CPS translations and also extend to other syntactic translations, and in particular to a wide range of effects accounted for by monads. On a long-term perspective, a lot is yet to be learned on the syntactic translations that are compatible with semantics interpretations, that is, which yield EF morphisms preserving evidences.

References

- [ADH⁺12] Zena M. Ariola, Paul Downen, Hugo Herbelin, Keiko Nakata, and Alexis Saurin. Classical call-by-need sequent calculi: The unity of semantic artifacts. In Tom Schrijvers and Peter Thiemann, editors, *Functional and Logic Programming - 11th International Symposium, FLOPS 2012, Kobe, Japan, May 23-25, 2012. Proceedings*, Lecture Notes in Computer Science, pages 32–46. Springer, 2012.
- [App92] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 1992.
- [Bar92] Henk Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.
- [BPT17] Simon Boulrier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Proceedings of CPP 2017*, pages 182–194, New York, NY, USA, 2017. ACM.
- [CH00] Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of ICFP 2000*, SIGPLAN Notices 35(9), pages 233–243. ACM, 2000.
- [CMT21] Liron Cohen, Étienne Miquey, and Ross Tate. Evidenced frames: A unifying framework broadening realizability models. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13, 2021.
- [DA14] Paul Downen and Zena M. Ariola. The duality of construction. In Zhong Shao, editor, *Programming Languages and Systems*, pages 249–269, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.
- [DMAJ16] Paul Downen, Luke Maurer, Zena M. Ariola, and Simon Peyton Jones. Sequent calculus as a compiler intermediate language. In *ICFP 2016*, 2016.
- [DMMZ10] Olivier Danvy, Kevin Millikin, Johan Munk, and Ian Zerny. *Defunctionalized Interpreters for Call-by-Need Evaluation*, pages 240–256. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.
- [Fre16] Jonas Frey. Classical Realizability in the CPS Target Language. *Electronic Notes in Theoretical Computer Science*, 325(Supplement C):111 – 126, 2016. The Thirty-second Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXII).
- [GM16a] Mauricio Guillermo and Alexandre Miquel. Specifying Peirce’s law in classical realizability. *Mathematical Structures in Computer Science*, 26(7):1269–1303, 2016.
- [GM16b] Mauricio Guillermo and Étienne Miquey. Classical realizability and arithmetical formulæ. *Mathematical Structures in Computer Science*, page 1–40, 2016.
- [Gri90] Timothy Griffin. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’90, pages 47–58, New York, NY, USA, 1990. ACM.
- [Kri03] Jean-Louis Krivine. Dependent choice, ‘quote’ and the clock. *Th. Comp. Sc.*, 308:259–276, 2003.
- [Kri04] Jean-Louis Krivine. A call-by-name lambda-calculus machine. In *Higher Order and Symbolic Computation*, 2004.
- [Kri12] Jean-Louis Krivine. Realizability algebras II : new models of ZF + DC. *Logical Methods in Computer Science*, 8(1):10, February 2012. 28 p.
- [Kri21] Jean-Louis Krivine. A program for the full axiom of choice. *Logical Methods in Computer Science*, Volume 17, Issue 3, September 2021.
- [Lep16] Rodolphe Lepigre. A classical realizability model for a semantical value restriction. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016*,

- Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.
- [Lep17] Rodolphe Lepigre. *Semantics and Implementation of an Extension of ML for Proving Programs. (Sémantique et Implantation d'une Extension de ML pour la Preuve de Programmes)*. PhD thesis, Grenoble Alpes University, France, 2017.
- [Miq11a] Alexandre Miquel. Existential witness extraction in classical realizability and via a negative translation. *Logical Methods in Computer Science*, 7(2):188–202, 2011.
- [Miq11b] Alexandre Miquel. Forcing as a program transformation. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science, LICS '11*, page 197–206, USA, 2011. IEEE Computer Society.
- [Miq17] Étienne Miquey. *Classical realizability and side-effects*. Ph.D. thesis, Université Paris Diderot ; Universidad de la República, Uruguay, November 2017.
- [Miq18] Étienne Miquey. A sequent calculus with dependent types for classical arithmetic. In *LICS 2018*, pages 720–729. ACM, 2018.
- [Miq20] Étienne Miquey. Revisiting the Duality of Computation: An Algebraic Analysis of Classical Realizability Models. In Maribel Fernández and Anca Muscholl, editors, *28th EACSL Annual Conference on Computer Science Logic (CSL 2020)*, volume 152 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:18, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.
- [MM09] Guillaume Munch-Maccagnoni. Focalisation and Classical Realisability. In Erich Grädel and Reinhard Kahle, editors, *Computer Science Logic '09*, volume 5771 of *Lecture Notes in Computer Science*, pages 409–423. Springer, Heidelberg, 2009.
- [MM13] Guillaume Munch-Maccagnoni. *Syntax and Models of a non-Associative Composition of Programs and Proofs*. PhD thesis, Univ. Paris Diderot, 2013.
- [Mur90] Chetan Murthy. *Extracting constructive content from classical proofs*. Ph.D. thesis, Cornell University, 1990.
- [OS08] P. Oliva and T. Streicher. On Krivine’s realizability interpretation of classical second-order arithmetic. *Fundam. Inform.*, 84(2):207–220, 2008.
- [Par97] M. Parigot. Proofs of strong normalisation for second order classical natural deduction. *J. Symb. Log.*, 62(4):1461–1479, 1997.
- [Pit02] Andrew M. Pitts. Tripos theory in retrospect. *Mathematical Structures in Computer Science*, 12(3):265–279, 2002.
- [Plo75] Gordon D. Plotkin. Call-by-name, call-by-value and the lambda-calculus. *Theor. Comput. Sci.*, 1(2):125–159, 1975.
- [Rie14] Lionel Rieg. *On Forcing and Classical Realizability*. Theses, Ecole normale supérieure de lyon - ENS LYON, June 2014.
- [SF93] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3-4):289–360, 1993.
- [SS75] Gerald J. Sussman and Guy L. Steele, Jr. An interpreter for extended lambda calculus. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.
- [vO08] Jaap van Oosten. *Realizability: an introduction to its categorical side*, volume 152 of *Studies in Logic and the Foundations of Mathematics*. Elsevier B. V., Amsterdam, 2008.

Building CFA for λ -calculus from Skeletal Semantics

Thomas Jensen, Vincent Rébiscoul, and Alan Schmitt

INRIA Rennes

Abstract

This paper describes a method to define a correct abstract interpretation from a formal description of the semantics of a programming language. Our approach is based on Skeletal Semantics. We extend it with a notion of program points, in order to differentiate two fragments of the program that are syntactically equivalent but appear at different locations. We introduce a methodology for deriving an abstract interpretation from a Skeletal Semantics that is correct by construction: given a program, abstract states are computed for each program points. We apply our method by defining a Control Flow Analysis for λ -calculus from its Skeletal Semantics.

1 Introduction

As the complexity of software increases, building static analyses becomes more and more important. Analyzers have improved for years and many companies use them to save time by detecting bugs [2], or prove some properties on their code [4]. In both cases, the soundness of the analysis is paramount. Methods to develop analyses from a description of a language are useful because they are systematic, even if done by hand. One of the most famous example is Abstract Interpretation [3], which explains how to define an abstract semantics from an operational semantics and how to prove that the abstract semantics is sound using a Galois connection. Other works used abstract interpretation, for example to build an abstract semantics from a big-step semantics [10].

In order to mechanize the design of sound analyses, one should start from a mechanized semantics. Several tools provide such semantics, such as \mathbb{K} [7], a framework to mechanize semantics using rewriting rules. This makes the formal definition of a semantics easier, and allows to mechanically derive objects from the semantics: for example in \mathbb{K} , one can automatically derive an interpreter from the mechanization of a language. However, \mathbb{K} is not an extensible framework and it is unclear how to derive an analysis from a mechanization in \mathbb{K} .

In this paper we describe the first steps towards the generation of analyses from a Skeletal Semantics. Skeletal Semantics [1] is a recent proposal for machine-representable semantics of programming languages, using a minimalist functional language, Skel. From the Skeletal Semantics of a language, a semantics can be derived by meta-interpretation of Skel, as a big-step semantics [6]. Abstract Interpretation is a powerful framework to build analyses, thus our work focuses on building an abstract interpretation from a Skeletal Semantics. Because both big-step semantics and abstract interpretations stem from the same syntactic object, the proof of soundness of the abstract interpretation is in large part independent from the language considered. Proving the correctness of the abstract interpretation and the big-step semantics given a particular Skeletal Semantics is therefore dependent on small lemmas only.

Our goal is to provide a methodology to easily define several semantics for a language that can be related to one another. Skeletal Semantics [1] is a recent proposal for machine-representable semantics of programming languages. The Skeletal Semantics of a language is a partial description of the language. Several meanings, called interpretations, can be given to this

description like a big-step semantics [6]. However, because the description is partial, some parts are left *unspecified*, and to fully define a semantics, one needs to provide *specifications* to the unspecified parts. There are two benefits to this approach. First, the different interpretations are language independent: an interpretation can be used with any Skeletal Semantics. Second, two interpretations can be related to one another: for example, in this paper we present an abstract interpretation that is a correct approximation of the big-step interpretation.

One benefit to this approach is that the interpretations of Skel can be used for any language with a Skeletal Semantics, the definition of a semantics is done by only specifying the language dependent unspecified parts. Because it is often easy to define relations between interpretations, the different semantics of one language can also be related to by proving small lemmas on the specifications that depend on the interpretations. For example, we present a method in this paper to define an abstract interpretation from a Skeletal Semantics that is a sound approximation of the big-step semantics, provided that some lemmas about the

Contributions We propose a new interpretation of Skel that includes program points in a systematic way. We define an abstract interpretation for Skel which is correct: the set of all the executions of the big-step semantics is safely over-approximated. We provide a modular methodology to prove correctness at the skeletal meta-level. We give an example of how to define a CFA analysis for λ -calculus using the abstract interpretation of Skel. This work has been implemented in a small OCaml program which takes as inputs a Skeletal Semantics and the definitions of unspecified types and terms, and returns an abstract interpreter.

In Section 2, we introduce the syntax of Skeletal Semantics. In Section 3, we present a big-step semantics (or concrete interpretation) of Skel. In Section 4, we show how to modify the big-step semantics to use program points of a given program. In Section 5, we present an abstract interpretation of Skel, used to define abstract interpretation for any language with a Skeletal Semantics, with a theorem of soundness between the Skel abstract interpretation and big-step semantics. Finally, in Section 6, we use the abstract interpretation to define a CFA analysis for the λ -calculus

2 Skeletal Semantics and their Syntax

Skeletal Semantics is a recent approach to mechanize semantics for programming languages [1]. It uses a minimalist, functional, and strongly typed language called Skel [6]. The Necro library [5] is a tool to manipulate Skeletal Semantics. Given Skel code, it can generate Ocaml code, Coq code, step by step debuggers, and more.

The mechanization of a semantics of a language using Skeletal Semantics is done by meta-interpretation of the Skel language, therefore *interpretations* must be provided. In this paper, we will present two interpretations: a big-step semantics and an abstract interpretation.

A Skeletal Semantics is a syntactic object, written in Skel, describing a language. We present the Skeletal Semantics of the call-by-value λ -calculus with environments. We start by defining useful types for our language:

```

1 type ident
2 type env
3
4 type clos =
5 | Clos (ident, lterm, env)
6
```

```

7  type lterm =
8  | Lam (ident, lterm)
9  | Var ident
10 | App (lterm, lterm)

```

It starts with two definitions of *unspecified* types, and two definitions of *specified* types. The types refer, in that order, to identifiers, environments, closures, and λ -terms. Having unspecified types is a unique trait of Skel. Keeping some types unspecified is useful because their instantiation can depend on the interpretation of Skel. Specified types are algebraic data types (ADT). The `clos` type contains only one constructor which parameter is a triplet with an identifier (representing a variable), a λ -term, and an environment, which is a usual definition for a closure. The type `lterm` is the definition of λ -terms: it can be a λ -abstraction, a variable or an application.

Moreover, a Skeletal Semantics contains terms:

```

12 val extEnv : (env, ident, clos) → env
13 val getEnv : (ident, env) → clos

```

The term $getEnv(x, e)$ is a lookup function: it returns the closure associated to x in e . The term $extEnv(e, x, v)$ returns a new environment equals to e but with the new binding where x maps to v . Terms must be explicitly typed. Moreover, both terms are *unspecified*: like types, terms do not need to be completely defined in the Skeletal Semantics. In fact, as `env` is unspecified, functions to access or extend it must be unspecified.

A Skeletal Semantics also contains specified terms:

```

15 val eval (s:env) (l:lterm): clos =
16     branch
17         let Lam (x, t) = l in
18             Clos (x, t, s)
19     or
20         let Var x = l in
21             getEnv (x, s)
22     or
23         let App (t1, t2) = l in
24             let Clos (x, t, s') = eval s t1 in
25             let w = eval s t2 in
26             let s'' = extEnv (s', x, w) in
27             eval s'' t
28     end
29

```

This specified term is the function detailing how a `lterm` should be computed given an environment. For example, the `eval` function starts with a branching with three branches, and each branch is guarded by a `let`-binding doing pattern-matching: this particular branching acts as a case analysis. Indeed, a branching is used to cover several cases: here, there is one case per constructor in the `lterm` type. The first branch evaluates a λ -abstraction by returning a closure, the second branch evaluates a variable by using the `getEnv` function. The final branch evaluates an application: first `t1` is evaluated, which gives a closure $Clos(x, t, s')$ representing the function with parameter x and body t , and where environment s' gives meaning to free variables in t . Then `t2` is evaluated to another closure w , and s' is extended such that x maps

TERM	t	$::=$	$x \mid C t \mid (t, \dots, t) \mid \lambda p : \tau \rightarrow S$
SKELETON	S	$::=$	$t_0 t_1 \dots t_n \mid \mathbf{let} p = S \mathbf{in} S \mid \text{branch } S \text{ or } \dots \text{ or } S \mathbf{end} \mid t$
PATTERN	p	$::=$	$x \mid _ \mid C p \mid (p, \dots, p)$
TYPE	τ	$::=$	$b \mid \tau \rightarrow \tau \mid (\tau, \dots, \tau)$
TERM DECL	r_t	$::=$	$\mathbf{val} x : \tau \mid \mathbf{val} x : \tau = t$
TYPE DECL	r_τ	$::=$	$\mathbf{type} b$ $\mid \mathbf{type} b = \mid C1 \tau_1 \dots \mid Cn \tau_n$

Figure 1: The Syntax of Skeletal Semantics

to w , giving a new environment s'' . Finally, the body of the function t is evaluated with the new environment s'' . This should not be too surprising for people familiar with the semantics of λ -calculus with environments.

Formally, the syntax of Skel is given in Figure 1: it is similar to λ -calculus where the syntax ensures that programs are in A-Normal Form [9]. A specified *term* is either a variable, a constructor applied to a term, a tuple, or a function. Intuitively, a term is a construct that is fully computed. A *skeleton* is either an application, a let-binding, a branching, or a term. Intuitively, a skeleton is a computation. Branching is the most exotic construct of the language, it will be explained in more depth later on. Skel uses *patterns*, notably to perform pattern-matching with let-bindings. A *term declaration* introduces a top-level term. It can either be *unspecified*, in which case only its name and type are given, or *specified*, in which case the specification is a term. A *type declaration* introduces a type that may be unspecified, or it can be the declaration of an algebraic datatype.

A Skeletal Semantics is a set of type definitions and term definitions, that can be specified or unspecified. In the following, for any constructor C of a specified algebraic data type τ , we write $C : (\tau_i, \tau)$ to state that C belongs to type τ and expects an argument of type τ_i .

3 Big-step Semantics of Skel

To give meaning to a Skeletal Semantics, we provide *interpretations* of Skel. In this section, we give the big-step semantics of Skel.

The predicate $\text{na}(\tau)$ is true when τ is not an arrow type. The *arity* of a function f , $\text{arity}(f)$, is n if f has type $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$, where $\text{na}(\tau)$. Let \mathcal{S} be an arbitrary Skeletal Semantics. We write $\text{Funs}(\mathcal{S})$ for the set of pairs $(\Gamma, \lambda p : \tau_1 \rightarrow S_0)$ such that $\lambda p : \tau_1 \rightarrow S_0$ appears in Skeletal Semantics \mathcal{S} . The typing environment Γ gives types to the free variables of $\lambda p : \tau_1 \rightarrow S_0$. Typing rules are given in the Appendix B and $\text{Funs}(\cdot)$ is formally defined in the Appendix C.

3.1 From Types to Concrete Values

We call the interpretation of types the function $V(\tau)$, that given a type τ returns the set of values of that type. These sets are defined with the relation $\vdash \cdot \in V(\cdot)$ in Figure 2. These rules are language independent and are completed with language-dependent rules upon instantiation.

$$\begin{array}{c}
 \frac{\forall 1 \leq i \leq n \quad \vdash v_i \in V(\tau_i)}{\vdash (v_1, \dots, v_n) \in V(\tau_1 \times \dots \times \tau_n)} \text{TUPLE} \qquad \frac{\vdash v \in V(\tau) \quad C : (\tau, \tau_a)}{\vdash C v \in V(\tau_a)} \text{ADT} \\
 \\
 \frac{(\Gamma, \lambda p : \tau_1 \rightarrow S) \in \text{Funs}(\mathcal{S}) \quad \Gamma \vdash E \quad \Gamma + p \leftarrow \tau_1 \vdash S : \tau_2}{\vdash (p, S, E) \in V(\tau_1 \rightarrow \tau_2)} \text{CLOS} \\
 \\
 \frac{\mathbf{val} f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \ [\neq t] \in \mathcal{S} \quad \text{na}(\tau)}{\vdash (f, \text{arity}(f)) \in V(\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau)} \text{DEF} \\
 \\
 \frac{\text{dom } \Gamma = \text{dom } E \quad \forall x \in \text{dom } E, \Gamma \vdash E(x) : \Gamma(x)}{\Gamma \vdash E} \text{IENV}
 \end{array}$$

Figure 2: Rules to Build Values

The **TUPLE** and **ADT** rules build tuples and values of ADTs. The **CLOS** rule builds a closure, which is a triplet (p, S, E) . p is the pattern that contains the parameters of the function, S is the body of the function and E an environment for the free variables in S . Note that this environment must be compatible with the typing of the free variables of the function (Rule **IENV**). The **DEF** rule builds named closures for application of unspecified and specified terms with arrow type. The closure contains only the name and the arity of the function.

An instantiation of a Skeletal Semantics must define the values belonging to unspecified types. We now show how this is done by example. In the case of λ -calculus, the unspecified types are **ident** and **env**, thus we provide rules to build these values.

$$\begin{array}{c}
 \frac{id \in \mathcal{V} = \{x, y, z, \dots\}}{\vdash id \in V(\mathbf{ident})} \text{IDENT} \qquad \frac{}{\vdash [] \in V(\mathbf{env})} \text{ENV-EMPTY} \\
 \\
 \frac{\vdash id \in V(\mathbf{ident}) \quad \vdash c \in V(\mathbf{clos}) \quad \vdash e \in V(\mathbf{env})}{\vdash (id, c) :: e \in V(\mathbf{env})} \text{ENV-CONS}
 \end{array}$$

Identifiers are variables from a countable set of variables \mathcal{V} , and an environment is an association list: it can be empty or built from a new binding and another environment.

3.2 Interpretation of Unspecified Terms

If X is a set, $\mathcal{P}_f(X)$ is the set of finite parts of X .

Now that values are defined, there remains to give definitions to unspecified terms. Take an unspecified term $\mathbf{val} t : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ such that $\text{na}(\tau)$, then an instantiation of t , written $\llbracket t \rrbracket$, is a function such that: $\llbracket t \rrbracket \in (V(\tau_1) \times \dots \times V(\tau_n)) \rightarrow \mathcal{P}_f(V(\tau))$. In particular, if $\mathbf{val} t : \tau$ and $\text{na}(\tau)$, then $\llbracket t \rrbracket \subseteq V(\tau)$. Allowing the specification of a term to be a function returning a set is useful to model non-determinism.

For our λ -calculus, the specifications of unspecified terms are:

$$\begin{aligned} \llbracket \text{getEnv} \rrbracket(x, (x, c) :: e) &= \{ c \} \\ \llbracket \text{getEnv} \rrbracket(x, (y, c) :: e) &= \llbracket \text{getEnv} \rrbracket(x, e) && x \neq y \\ \llbracket \text{getEnv} \rrbracket(x, []) &= \{ \} \\ \llbracket \text{extEnv} \rrbracket(e, x, c) &= (x, c) :: e \end{aligned}$$

The interpretation $\llbracket \text{getEnv} \rrbracket(x, e)$ is defined with three equations: the first equation returns the closure when the correct binding is at the head of the environment, the second equation is a recursive call when the first binding is not the correct one, and in last equation, getEnv returns the empty set because the environment is empty. $\llbracket \text{extEnv} \rrbracket(e, x, c)$ returns a similar environment to e , but with a new binding where x is associated to c .

3.3 Big-step Semantics

We now define the big-step semantics of Skel. $E, S \Downarrow v$ is a relation from a skeletal environment E , mapping skeletal variables to values, and a skeleton S to a value v . The relation is defined in Figure 3a.

There are four rules to evaluate variables, depending on how the variable was defined and its type. If the variable was defined in a 'let' expression, the environment E maps the variable to its value (Rule VAR). Otherwise, assuming the Skeletal Semantics is well typed, the variable must be the name of a specified or unspecified term. If this term has an arrow type, we simply return a named closure, whether the variable is specified or not. A named closure is a pair of the name and arity of the function (Rule TERM_CLOS). If the variable is declared in the Skeletal Semantics, then it is bound to a closed term t , which is evaluated in the empty environment because it only depends on term declarations of the Skeletal Semantics (Rule TERM_SPEC). If the variable is unspecified, a value of the provided instantiation is returned (Rule TERM_UNSPEC). Rules CONST and TUPLE are usual. Rule CLOS returns a skeletal closure. One may observe a similarity between this meta-rule and the corresponding object rule in the λ -calculus; this is because Skel is a meta-language that is an extension of the λ -calculus, but the meta-language and the object language should not be confused.

We now turn to the evaluation of skeletons. Rule LET_IN is usual. Rule BRANCH evaluates a branching by non-deterministically returning the result of the evaluation of a branch. To evaluate an application, we define another relation, defined in Figure 3b with four rules. Rule BASE returns the result when all arguments have been processed. Rule CLOS is the application of a closure. Rules SPEC and UNSPEC are the application of a named closure to a list of arguments. The first evaluates the corresponding definition while the second one uses the user-provided instantiation. Non-specified functions may be non-deterministic since their instantiation may return a set. Note that we only consider full application of named functions: all arguments must be provided. Extending this semantics to partial application is easy, although verbose, but it is not necessary to describe our contributions. Finally, the pattern matching rules for environment extension are given in Figure 4.

$$\begin{array}{c}
 \frac{E(x) = v}{E, x \Downarrow v} \text{VAR} \qquad \frac{\mathbf{val} f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau [= t] \quad \text{na}(\tau) \quad n \geq 1}{E, f \Downarrow (f, n)} \text{TERMCLOS} \\
 \\
 \frac{\mathbf{val} x : \tau = t \in \mathcal{S} \quad \text{na}(\tau) \quad \emptyset, t \Downarrow v}{E, x \Downarrow v} \text{TERMSPEC} \\
 \\
 \frac{\mathbf{val} x : \tau \in \mathcal{S} \quad \text{na}(\tau) \quad v \in \llbracket x \rrbracket}{E, x \Downarrow v} \text{TERMUNSPEC} \qquad \frac{E, t \Downarrow v}{E, (Ct) \Downarrow Cv} \text{CONST} \\
 \\
 \frac{E, t_1 \Downarrow v_1 \quad \dots \quad E, t_n \Downarrow v_n}{E, (t_1, \dots, t_n) \Downarrow (v_1, \dots, v_n)} \text{TUPLE} \qquad \frac{}{E, (\lambda p : \tau \rightarrow S) \Downarrow (p, S, E)} \text{CLOS} \\
 \\
 \frac{E, S_1 \Downarrow v \quad \vdash E + p \leftarrow v \rightsquigarrow E' \quad E', S_2 \Downarrow w}{E, \text{let } p = S_1 \text{ in } S_2 \Downarrow w} \text{LETIN} \qquad \frac{E, S_i \Downarrow v}{E, (S_1, \dots, S_n) \Downarrow v} \text{BRANCH} \\
 \\
 \frac{\forall i \in [0..n]. E, t_i \Downarrow v_i \quad v_0 v_1 \dots v_n \Downarrow_{\text{app}} w}{E, (t_0 t_1 \dots t_n) \Downarrow w} \text{APP}
 \end{array}$$

(a) Rules for the Big-step Semantics of Skeletons and Terms

$$\begin{array}{c}
 \frac{}{v \Downarrow_{\text{app}} v} \text{BASE} \qquad \frac{\vdash E + p \leftarrow v_1 \rightsquigarrow E' \quad E', S \Downarrow v \quad v v_2 \dots v_n \Downarrow_{\text{app}} w}{(p, S, E) v_1 \dots v_n \Downarrow_{\text{app}} w} \text{CLOS} \\
 \\
 \frac{\mathbf{val} f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau = t \in \mathcal{S} \quad \text{na}(\tau) \quad \emptyset, t \Downarrow v \quad v v_1 \dots v_n \Downarrow_{\text{app}} w}{(f, n) v_1 \dots v_n \Downarrow_{\text{app}} w} \text{SPEC} \\
 \\
 \frac{\mathbf{val} f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \in \mathcal{S} \quad \text{na}(\tau) \quad w \in \llbracket f \rrbracket(v_1, \dots, v_n)}{(f, n) v_1 \dots v_n \Downarrow_{\text{app}} w} \text{UNSPEC}
 \end{array}$$

(b) Rules for Application for the Big-step Semantics

Figure 3: Interpretation rules of the Big-step Semantics

$$\begin{array}{c}
 \frac{}{\vdash E + _ \leftarrow v \rightsquigarrow E} \text{ASN-WILDCARD} \qquad \frac{}{\vdash E + x \leftarrow v \rightsquigarrow (x, v) :: E} \text{ASN-VAR} \\
 \\
 \frac{\vdash E + p \leftarrow v \rightsquigarrow E'}{\vdash E + Cp \leftarrow Cv \rightsquigarrow E'} \text{ASN-CONSTR} \\
 \\
 \frac{\vdash E + p_1 \leftarrow v_1 \rightsquigarrow E_2 \quad \dots \quad \vdash E_n + p_n \leftarrow v_n \rightsquigarrow E'}{\vdash E + (p_1, \dots, p_n) \leftarrow (v_1, \dots, v_n) \rightsquigarrow E'} \text{ASN-TUPLE}
 \end{array}$$

Figure 4: Rule of Extension of Environment through Pattern Matching

4 Big-step Semantics with Program Points

4.1 Program Points and New Values

We present our first contribution, which is a method to use *program points* in the big-step semantics of a Skeletal Semantics. This methodology is reused later for the abstract interpretation of Skel. Let \mathcal{S} be a Skeletal Semantics. It usually contains one or more specified algebraic data types that define the syntax of the language, i.e., *programs*. We call these types *program types*. In the case of λ -calculus, `lterm` is the only program type. Let τ a program type in \mathcal{S} and $\mathbf{prg} \in V(\tau)$ a program, we can refer to sub-terms of \mathbf{prg} using program points. A program point is a path from the root of \mathbf{prg} to one of its sub-term. Given a program point pp , $\mathbf{prg}@ \text{pp}$ is the sub-term of \mathbf{prg} obtained by following path pp . A program point pp is an element of $\text{ppt} = \mathbb{N}^*$, the set program points. ϵ is the empty path and $i \cdot \text{pp}$ is a path with first element i , and pp is the rest of the path. The $@$ operator is formally defined as:

$$\begin{aligned} v@ \epsilon &= v \\ C(v_0, \dots, v_{n-1})@i \cdot \text{pp} &= v_i@ \text{pp} && \text{when } 0 \leq i \leq n - 1 \end{aligned}$$

To give an example, for λ -calculus, let $\mathbf{prg} = \text{Lam}(x, \text{Var } x)$. Then, $\mathbf{prg}@0 = x$ and $\mathbf{prg}@1 = \text{Var } x$.

Our approach is to replace values of program types with program points. Each program point refers to a sub-term of the main program: \mathbf{prg} , which is now a parameter of the interpretation. Let \mathcal{T} be the set of program types from \mathcal{S} and \mathbf{prg} a program. Therefore, all values of $\tau \in \mathcal{T}$ are program points and represent sub-terms of \mathbf{prg} . In particular, because \mathbf{prg} is a program, its type should be in \mathcal{T} .

For each type τ , a set of values is built using previously defined rules 2 except for program types that are defined by the equation:

$$\tau \in \mathcal{T} \implies V_{\mathbf{prg}}^{\text{ppt}}(\tau) = \{ \text{pp} \in \text{ppt} \mid \mathbf{prg}@ \text{pp} \in V(\tau) \}$$

The set $V_{\mathbf{prg}}^{\text{ppt}}(\tau)$ is the set of program points that refer to sub-terms of \mathbf{prg} of type τ . This is an important restriction because if $\tau \in \mathcal{T}$, then a value $v \in V_{\mathbf{prg}}^{\text{ppt}}(\tau)$ is a program point and therefore necessarily represent a fragment of \mathbf{prg} and not an arbitrary program.

\mathcal{T} is a set because some languages may have many types for different parts of programs: an imperative language would need at least one type for statements and one type for expressions. Let us give an example with the λ -calculus. We take our program to be $\mathbf{prg} = \text{App}(\text{Lam}(x, \text{Var } x), \text{Var } y)$ and $\mathcal{T} = \{ \text{lterm} \}$. The interpretation of the λ -terms is defined in the next equation. The program points are underlined to differentiate them from natural numbers.

$$V_{\mathbf{prg}}^{\text{ppt}}(\text{lterm}) = \{ \epsilon, \underline{0}, \underline{1}, \underline{01} \}$$

For example, we have $\mathbf{prg}@ \underline{1} = \text{Var } y$

The interpretation of unspecified types does not change and other types are built using previous rules in Figure 2. However, we assume new specifications of unspecified terms. Indeed, some unspecified terms may depend on types in \mathcal{T} , therefore, for all unspecified terms x in \mathcal{S} , we suppose that new specifications are provided: $\llbracket x \rrbracket^{\text{ppt}}$.

4.2 Pattern Matching with Program Points

The interpretation of skeletons with program points does not fundamentally change: only the pattern matching is modified and uses an unfolding mechanism. Unfolding matches a program

$$\begin{aligned} \text{NC}(\tau_1 \rightarrow \tau_2) &= \left\{ (f, n) \mid \begin{array}{l} \mathbf{val} f : \tau_1 \rightarrow \tau_2[= t] \in \mathcal{S} \\ \text{arity}(f) = n \end{array} \right\} \\ \text{C}(\tau_1 \rightarrow \tau_2) &= \left\{ (p, S, E^\#) \mid \begin{array}{l} \exists(\Gamma, \lambda p : \tau_1 \rightarrow S) \in \text{Funs}(\mathcal{S}), \\ \Gamma \vdash E^\# \wedge \Gamma + p \leftarrow \tau_1 \vdash S : \tau_2 \end{array} \right\} \end{aligned}$$

Figure 5: Sets of Abstract Named Closures and Closures

point with a constructor pattern and is described by the following rule.

$$\frac{v_j = \mathbf{if} \tau_j \in \mathcal{T} \mathbf{then} \text{pp} \cdot j \mathbf{else} v'_j \quad \mathcal{T}, \mathbf{prg} \vdash E + p \leftarrow (v_0, \dots, v_{n-1}) \rightsquigarrow E'}{\mathcal{T}, \mathbf{prg} \vdash E + C p \leftarrow \text{pp} \rightsquigarrow E'} \text{ASN-UNFOLD}$$

When a program point pp is matched with a constructor pattern $C p$, $\mathbf{prg}@\text{pp}$ is expected to have the form $C(v'_0, \dots, v'_{n-1})$. Then, for $0 \leq j \leq n-1$, v_j is $\text{pp} \cdot j$ if v'_j has a program type $\tau_j \in \mathcal{T}$, or v'_j otherwise. Then the pattern p is matched with (v_1, \dots, v_n) . The unfolding rule exhibits the constructor and its parameters at program point pp , then continues the pattern matching recursively.

We give an example for λ -calculus, let $\mathbf{prg} = \text{App}(\text{Lam}(x, \text{Var } x), \text{Var } y)$. To compute $\mathcal{T}, \mathbf{prg} \vdash E + \text{Lam}(p) \leftarrow \underline{0} \rightsquigarrow E'$ where p is a pattern, by applying the ASN-UNFOLD rule, it comes that: $\mathcal{T}, \mathbf{prg} \vdash E + p \leftarrow (x, \underline{01}) \rightsquigarrow E'$ Indeed, $\mathbf{prg}@\underline{0} = \text{Lam}(x, \text{Var } x)$ and $\text{Var } x$ has program point $\underline{01}$, therefore p is matched with $(x, \underline{01})$

The big-step semantics of skeletons and the big-step semantics of skeletons with program points are closely related. We formalize this connection through a theorem presented in Appendix D.

5 Abstract Interpretation

We define an abstract interpretation of Skel that is sound with respect to the big-step semantics of Section 4. The abstract interpretation is used in the next section to define a Control Flow Analysis for λ -calculus.

We define the abstract values, comparison functions and upper bounds in Section 5.1. We define the state of the abstract interpretation in Section 5.2. We present the interpretation rules of skeletons for an abstract interpretation in Section 5.3. We present how the abstract values are linked to the concrete values by defining concretization functions in Section 5.4. Finally, we formulate a theorem of soundness in Section 5.5. In the following sections, \mathcal{S} denotes an arbitrary Skeletal Semantics.

5.1 Definition of Abstract Values

The abstract values are defined similarly to the concrete ones. In Figure 5 we define the set of abstract named closures and the set of abstract closures. Abstract named closures are pairs of a name of a function defined in the skeletal semantics and its arity. Abstract closures are a triplet with a pattern, a skeleton and an abstract environment. The abstract environment is defined later in this section.

$$\begin{array}{c}
 \frac{}{\vdash^\# \perp_\tau \in V^\#(\tau)} \text{BOTTOM} \qquad \frac{}{\vdash^\# \top_\tau \in V^\#(\tau)} \text{TOP} \\
 \\
 \frac{\forall (v_1^\#, \dots, v_n^\#) \in t^\# \quad \forall 1 \leq i \leq n, \vdash^\# v_i^\# \in V^\#(\tau_i) \wedge v_i^\# \neq \perp_{\tau_i}}{\vdash^\# t^\# \in V^\#(\tau_1 \times \dots \times \tau_n)} \text{TUPLE} \\
 \\
 \frac{\vdash^\# v^\# \in V^\#(\tau) \quad C : (\tau, \tau_a) \quad v^\# \neq \perp_\tau}{\vdash^\# C v^\# \in V^\#(\tau_a)} \text{ADT} \\
 \\
 \frac{nc \subseteq \text{NC}(\tau_1 \rightarrow \tau_2) \quad c \subseteq \text{C}(\tau_1 \rightarrow \tau_2)}{\vdash^\# c \cup nc \in V^\#(\tau_1 \rightarrow \tau_2)} \text{FUNS} \\
 \\
 \frac{\text{dom } \Gamma = \text{dom } E^\# \quad \forall x \in \text{dom } E^\#, \Gamma \vdash E^\#(x) : \Gamma(x)}{\Gamma \vdash E^\#} \text{IENV}
 \end{array}$$

Figure 6: Rules to Build Abstract Values

The rules to define abstract values are presented on Figure 6. The **BOTTOM** and **TOP** rules define a least and a greatest element for each type. The **TUPLE** rule define the abstract tuples, that are a set of tuples of abstract values. Because one of our target is CFA analysis for λ -calculus, having a set is necessary not to loose too much precision during the analysis. On line 24 of the Skeletal Semantics of the λ -calculus **A**, *eval s t1* returns a closure, $Clos(x, t, s)$ which is essentially a triplet. The relation between identifier, term and environment must be preserved. Values of algebraic types, defined by the rule **ADT**, are constructors applied to abstract values. Abstract functions, defined by the rule **FUNS**, is the union of a subset of abstract named closures, and a subset of abstract closures. For instance for the λ -calculus, the set $\{(\mathbf{eval}, 2), (\mathbf{s}, \lambda \mathbf{l} : \mathbf{lterm}.S_{eval}, \emptyset)\}$ belongs to $V^\#(\mathbf{env} \rightarrow \mathbf{lterm} \rightarrow \mathbf{clos})$, where S_{eval} is the body of the **eval** specified term. Finally, the rule **IENV** defines abstract environments, that are partial functions mapping skeletal variables to abstract values.

To compare abstract values, we define partial orders. For every unspecified type τ_u , we assume comparison function $\sqsubseteq_{\tau_u}^\#$ which is an order and with the constraint that \top_{τ_u} and \perp_{τ_u} are the greatest and smallest elements of $V_{\mathbf{prg}}^\#(\tau_u)$ respectively. For every other types, the comparison function is the smallest order that satisfies the following equations:

$$\begin{aligned}
 C v^\# \sqsubseteq_{\tau_a}^\# C w^\# &\iff v^\# \sqsubseteq_\tau^\# w^\# \text{ with } C : (\tau, \tau_a) \\
 t_1^\# \sqsubseteq_{\tau_1 \times \dots \times \tau_n}^\# t_2^\# &\iff \forall (v_1^\#, \dots, v_n^\#) \in t_1^\#, \exists (w_1^\#, \dots, w_n^\#) \in t_2^\#, \forall i, 1 \leq i \leq n, v_i^\# \sqsubseteq_{\tau_i}^\# w_i^\# \\
 F_1 \sqsubseteq_{\tau_1 \rightarrow \tau_2}^\# F_2 &\iff \begin{cases} (f, n) \in F_1 \implies (f, n) \in F_2 \\ (p, S, E_1^\#) \in F_1 \implies \exists (p, S, E_2^\#) \in F_2, E_1^\# \sqsubseteq_{env}^\# E_2^\# \end{cases} \\
 E_1^\# \sqsubseteq_{env}^\# E_2^\# &\iff \Gamma \vdash E_1^\# \wedge \Gamma \vdash E_2^\# \wedge \forall x \in \text{dom } E_1^\#, E_1^\#(x) \sqsubseteq_{\Gamma(x)}^\# E_2^\#(x) \\
 &\quad v^\# \sqsubseteq_\tau^\# \top_\tau \\
 &\quad \perp_\tau \sqsubseteq_\tau^\# v^\#
 \end{aligned}$$

To compare algebraic values, their parameters are compared. Tuples are compared by checking

that all tuples of abstract values of the left tuple are smaller than a tuple of abstract values in the right tuple. To compare two functions, all named closures of the left function must be in the right function. Moreover, for all closures in the left function, there must be a closure in the right function with the same pattern and skeleton, but with a bigger abstract environment. Abstract environments are compared using point-wise lifting. For each type, top and bottom are respectively the smallest and greatest elements of the type.

For each type, an upper bound (or join) is defined. For every non-specified type τ_u , we assume an upper bound $\sqcup^\#_{\tau_u}$. We define an upper bound $\sqcup^\#_\tau$ for every other types.

$$\begin{aligned}
 (C \ v^\#) \sqcup^\#_{\tau_a} (C \ w^\#) &= C \ (v^\# \sqcup^\#_\tau w^\#) \text{ with } C : (\tau, \tau_a) \\
 (C \ v^\#) \sqcup^\#_{\tau_a} (D \ w^\#) &= \top_{\tau_a} \text{ with } C : (\tau, \tau_a) \wedge D : (\tau', \tau_a) \\
 t_1^\# \sqcup^\#_{\tau_1 \times \dots \times \tau_n} t_2^\# &= t_1^\# \cup t_2^\# \\
 F_1 \sqcup^\#_{\tau_1 \rightarrow \tau_2} F_2 &= F_1 \cup F_2 \\
 E_1^\# \sqcup^\#_{env} E_2^\# &= \left\{ x \in \text{dom } E_1^\# \mapsto E_1^\#(x) \sqcup^\# E_2^\#(x) \right\} \\
 v^\# \sqcup^\#_\tau \top_\tau &= \top_\tau \sqcup^\#_\tau v^\# = \top_\tau \\
 v^\# \sqcup^\#_\tau \perp_\tau &= \perp_\tau \sqcup^\#_\tau v^\# = v^\#
 \end{aligned}$$

Joining two algebraic values with the same constructor is joining their parameters, and joining algebraic values with different constructors yield top. Joining abstract tuples our abstract functions is set union. Joining abstract environments is done by point-wise lifting. For each type, top is an absorbing element, and bottom is the neutral element.

5.2 State of the Abstract Interpretation

The state of the abstract interpretation \mathcal{A} is a machine representable state that contains information collected throughout the abstract interpretation. It is dependent on the analysis and the language, and therefore is non-generic and must be specified. To give an example, when defining a CFA in the next section, the abstract state will contain a mapping from program points to sets of λ -abstraction, that can be viewed as the potential results when evaluating a sub-term at some program point of the analyzed program. We require an order on the abstract states. Intuitively, we should only add information in the states throughout the analysis, and therefore at each step of the analysis, the states should only increase.

A state contains several components: $\mathcal{A} \cdot c$ is the component c of the abstract state \mathcal{A} . The notation $\{ \mathcal{A} \text{ with } c = v \}$ denotes a new state equals to \mathcal{A} , excepts for the c component which is equal to v .

5.3 The Abstract Interpretation of Skel

The abstract interpretation maintains a callstack of specified function. This is used for loop detection and prevent infinite computations: the idea is to inspect the callstack at each call to a specified function to detect identical nested calls. Callstacks are ordered list of frames,

formally defined as:

$$\frac{\overline{\epsilon \in \text{callstack}} \quad \mathcal{A} \text{ an abstract state} \quad \text{val } f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau = t \in \mathcal{S} \quad \text{na}(\tau) \quad v_i \in V_{\text{prg}}^\#(\tau_i) \quad \pi \in \text{callstack}}{(f, \mathcal{A}, [v_1, \dots, v_n]) :: \pi \in \text{callstack}}$$

The abstract interpretation of skeletons is given on Figure 7a. The abstract interpretation of skeletons is similar to the big-step interpretation: the evaluation of terms is almost unchanged except that evaluating a closure or a tuple returns a singleton. When evaluating a skeleton (branch, let-binding, or application), a state of the abstract interpretation is carried through the computations. In the BRANCH rule, all branches are evaluated and joined instead of only one branch being evaluated. The pattern matching now returns sets of environments rather than one, and this will be detailed later, but in consequence the LETIN rule may evaluate S_2 in several abstract environments. The APP rule evaluates all terms and pass a list of values to the application relation. There are separate rules to handle applications on Figure 7b. Because the abstraction of a function is a set of closures and named closures, the APP-SET rule evaluates each one individually. The BASE rule returns the remaining value when all arguments have been processed. Because the extension of environments returns a set of abstract environment, the CLOS rule is modified accordingly and the body of the function S is evaluated in all abstract environments. The SPEC rule evaluates the call to a specified function by doing three things:

- it uses an $\text{update}_f^{\text{in}}(\cdot)$ function which is language dependent and must be specified. It can modify the arguments and the state of the abstract interpretation.
- the call is performed (a frame is added to the callstack at that point).
- an $\text{update}_f^{\text{out}}(\cdot)$ function can modify the state of the interpretation and the returned value.

These update functions are used to maintain invariants and update the state of the abstract interpretation. An example of their use will be presented in the next section. The update functions must respect the following constraints to ensure soundness:

Remark 1.

$$\begin{aligned} \text{update}_f^{\text{in}}(\mathcal{A}, [v_1^\#, \dots, v_k^\#]) = [v_1'^\#, \dots, v_k'^\#], \mathcal{A}' &\implies (v_1^\#, \dots, v_k^\#) \sqsubseteq^\# (v_1'^\#, \dots, v_k'^\#) \wedge \mathcal{A} \sqsubseteq^\# \mathcal{A}' \\ \text{update}_f^{\text{out}}(\mathcal{A}, [v_1^\#, \dots, v_k^\#], v^\#) = v'^\#, \mathcal{A}' &\implies v^\# \sqsubseteq^\# v'^\# \wedge \mathcal{A} \sqsubseteq^\# \mathcal{A}' \end{aligned}$$

The extension of environments, or pattern matching, presented on Figure 8 is modified such that it returns a set of abstract environments. This is necessary because our abstraction of tuples is a finite set of tuples of abstract values. In order not to lose too much precision, we return one abstract environment per tuple of abstract values in our abstract tuple. The rules are similar to the big-step semantics, excepts that there are two rules for tuples: the rule ASN-TUPLE-SINGLETON extends the environment with a tuple of abstract values, the ASN-TUPLE forwards all tuples of abstract values in $t^\#$ to the ASN-TUPLE-SINGLETON rule.

$$\begin{array}{c}
 \frac{E^\sharp(x) = v^\sharp}{E^\sharp, x \Downarrow v^\sharp} \text{VAR} \qquad \frac{\mathbf{val} f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau [= t] \in \mathcal{S} \quad \text{na}(\tau)}{E^\sharp, f \Downarrow \{(f, \text{arity}(f))\}} \text{TERMCLOS} \\
 \\
 \frac{\mathbf{val} x : \tau = t \in \mathcal{S} \quad \emptyset, t \Downarrow v^\sharp}{E^\sharp, x \Downarrow v^\sharp} \text{TERMSPEC} \qquad \frac{\mathbf{val} x : \tau \in \mathcal{S} \quad \text{na}(\tau)}{E^\sharp, x \Downarrow \llbracket x \rrbracket^\sharp} \text{TERMUNSPEC} \\
 \\
 \frac{E^\sharp, t \Downarrow v^\sharp}{E^\sharp, Ct \Downarrow Cv^\sharp} \text{CONST} \qquad \frac{E^\sharp, t_1 \Downarrow v_1^\sharp \quad \dots \quad E^\sharp, t_n \Downarrow v_n^\sharp}{E^\sharp, (t_1, \dots, t_n) \Downarrow \{(v_1^\sharp, \dots, v_n^\sharp)\}} \text{TUPLE} \\
 \\
 \frac{}{\pi, E^\sharp, \lambda p : \tau \cdot S \Downarrow \{(p, S, E^\sharp)\}} \text{CLOS} \qquad \frac{\pi, \mathcal{A}, E^\sharp, S_i \Downarrow v_i^\sharp, \mathcal{A}_i}{\pi, \mathcal{A}, E^\sharp, (S_1 \dots S_n) \Downarrow \sqcup^\sharp v_i^\sharp, \sqcup^\sharp \mathcal{A}_i} \text{BRANCH} \\
 \\
 \frac{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + p \leftarrow v^\sharp \rightsquigarrow \{E_1^\sharp, \dots, E_n^\sharp\} \quad \pi, \mathcal{A}, E^\sharp, S_1 \Downarrow v^\sharp, \mathcal{A}' \quad \pi, \mathcal{A}', E_i^\sharp, S_2 \Downarrow w_i^\sharp, \mathcal{A}_i}{\pi, \mathcal{A}, E^\sharp, \text{let } p = S_1 \text{ in } S_2 \Downarrow \sqcup^\sharp w_i^\sharp, \sqcup^\sharp \mathcal{A}_i} \text{LETIN} \\
 \\
 \frac{E^\sharp, t_i \Downarrow v_i^\sharp \quad \pi, \mathcal{A}, v_0^\sharp v_1^\sharp \dots v_n^\sharp \Downarrow_{\text{app}} v^\sharp, \mathcal{A}'}{\pi, \mathcal{A}, E^\sharp, t_0 t_1 \dots t_n \Downarrow v^\sharp, \mathcal{A}'} \text{APP}
 \end{array}$$

(a) Rules for the Abstract Interpretation of Skeletons and Terms

$$\begin{array}{c}
 \frac{v_i^\sharp = \bigcup_{i=1}^n \{w_i\} \quad \pi, \mathcal{A}, w_i v_1^\sharp \dots v_n^\sharp \Downarrow_{\text{app}} v_{w_i}^\sharp, \mathcal{A}_i}{\pi, \mathcal{A}, v_0^\sharp v_1^\sharp \dots v_n^\sharp \Downarrow_{\text{app}} \sqcup^\sharp v_{w_i}^\sharp, \sqcup^\sharp \mathcal{A}_i} \text{APP-SET} \qquad \frac{}{\pi, \mathcal{A}, v^\sharp \Downarrow_{\text{app}} v^\sharp, \mathcal{A}} \text{BASE} \\
 \\
 \frac{\forall E_i^\sharp \in \{E_1^\sharp, \dots, E_m^\sharp\} \quad \mathcal{T}, \mathbf{prg} \vdash E^\sharp + p \leftarrow^\sharp v_1^\sharp \rightsquigarrow \{E_1^\sharp, \dots, E_m^\sharp\} \quad \pi, \mathcal{A}, E_i^\sharp, S \Downarrow w_i^\sharp, \mathcal{A}_i \quad \pi, \mathcal{A}_i, w_i^\sharp v_2^\sharp \dots v_n^\sharp \Downarrow_{\text{app}} u_i^\sharp, \mathcal{A}'_i}{\pi, \mathcal{A}, (p, S, E^\sharp) v_1^\sharp \dots v_n^\sharp \Downarrow_{\text{app}} \sqcup^\sharp u_i^\sharp, \sqcup^\sharp \mathcal{A}'_i} \text{CLOS} \\
 \\
 \frac{\mathbf{val} f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau = t \in \mathcal{S} \quad \text{na}(\tau) \quad \emptyset, t \Downarrow v^\sharp \quad \text{update}_f^{\text{in}}(\mathcal{A}, [v_1^\sharp, \dots, v_n^\sharp]) = \mathcal{A}', [v_1'^\sharp, \dots, v_n'^\sharp] \quad (f, [v_1'^\sharp, \dots, v_n'^\sharp]) \notin \pi \quad (f, [v_1'^\sharp, \dots, v_n'^\sharp]) :: \pi, \mathcal{A}', v^\sharp v_1'^\sharp \dots v_n'^\sharp \Downarrow_{\text{app}} w^\sharp, \mathcal{A}'' \quad \text{update}_f^{\text{out}}(\mathcal{A}'', [v_1'^\sharp, \dots, v_n'^\sharp], w^\sharp) = w'^\sharp, \mathcal{A}'''}{\pi, \mathcal{A}, (f, n) v_1^\sharp \dots v_n^\sharp \Downarrow_{\text{app}} w'^\sharp, \mathcal{A}'''} \text{SPEC} \\
 \\
 \frac{\mathbf{val} f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau = t \in \mathcal{S} \quad \text{na}(\tau) \quad \emptyset, t \Downarrow v^\sharp \quad \text{update}_f^{\text{in}}(\mathcal{A}, [v_1^\sharp, \dots, v_n^\sharp]) = \mathcal{A}', [v_1'^\sharp, \dots, v_n'^\sharp] \quad (f, [v_1'^\sharp, \dots, v_n'^\sharp]) \in \pi}{\pi, \mathcal{A}, (f, n) v_1^\sharp \dots v_n^\sharp \Downarrow_{\text{app}} \perp, \mathcal{A}'} \text{SPEC-LOOP} \\
 \\
 \frac{\mathbf{val} f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \in \mathcal{S} \quad \text{na}(\tau) \quad \llbracket f \rrbracket^\sharp(\mathcal{A}, v_1^\sharp, \dots, v_n^\sharp) = w^\sharp, \mathcal{A}'}{\pi, \mathcal{A}, (f, n) v_1^\sharp \dots v_n^\sharp \Downarrow_{\text{app}} w^\sharp, \mathcal{A}'} \text{UNSPEC}
 \end{array}$$

$$\begin{array}{c}
 \frac{}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + _ \leftarrow^\sharp v \rightsquigarrow \{E^\sharp\}} \text{ASN-WILDCARD} \\
 \\
 \frac{}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + x \leftarrow^\sharp v \rightsquigarrow \{(x, v^\sharp) :: E^\sharp\}} \text{ASN-VAR} \\
 \\
 \frac{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + p \leftarrow^\sharp v \rightsquigarrow \xi}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + Cp \leftarrow^\sharp Cv \rightsquigarrow \xi} \text{ASN-CONSTR} \\
 \\
 \frac{\begin{array}{c} \mathcal{T}, \mathbf{prg} \vdash E^\sharp + p_1 \leftarrow^\sharp v_1 \rightsquigarrow \xi_1 \\ \dots \\ \mathcal{T}, \mathbf{prg} \vdash \xi_{n-1} + p_n \leftarrow^\sharp v_n \rightsquigarrow \xi_n \end{array}}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + (p_1, \dots, p_n) \leftarrow^\sharp (v_1, \dots, v_n) \rightsquigarrow \xi_n} \text{ASN-TUPLE-SINGLETON} \\
 \\
 \frac{(v_1, \dots, v_n) \in t \quad \mathcal{T}, \mathbf{prg} \vdash E^\sharp + (p_1, \dots, p_n) \leftarrow^\sharp (v_1, \dots, v_n) \rightsquigarrow \xi_{v_1, \dots, v_n}}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + (p_1, \dots, p_n) \leftarrow^\sharp t \rightsquigarrow \bigcup_{(v_1, \dots, v_n) \in t} \xi_{v_1, \dots, v_n}} \text{ASN-TUPLE} \\
 \\
 \frac{\begin{array}{c} \mathbf{prg}@pp = C(v'_1, \dots, v'_n) \quad C : (\tau_1 \times \dots \times \tau_n, \tau) \\ v_j = \mathbf{if} \tau_j \in \mathcal{T} \mathbf{then} pp \cdot j \mathbf{else} v'_j \quad \mathcal{T}, \mathbf{prg} \vdash E^\sharp + p \leftarrow^\sharp \{(v_1, \dots, v_n)\} \rightsquigarrow \xi \end{array}}{\mathcal{T}, \mathbf{prg} \vdash E^\sharp + Cp \leftarrow^\sharp pp \rightsquigarrow \xi} \text{ASN-UNFOLD}
 \end{array}$$

Figure 8: Extension of Environment through Pattern Matching for Abstract Interpretation

5.4 Concretization Function

To define concretization functions for abstract values, we assume monotonic concretization functions for abstract values of non-specified types. Concretization functions are defined for every specified types, and take the state of the abstract interpretation as a parameter:

$\gamma_\tau(\mathcal{A}, \cdot) : V_{\mathbf{prg}}^\sharp(\tau) \rightarrow \mathcal{P}_f(V_{\mathbf{prg}}^{\text{ppt}}(\tau))$. We also define a function of concretization γ_{env} which maps abstract skeletal environments to concrete skeletal environments.

$$\begin{aligned}
 \gamma_{\tau_a}(\mathcal{A}, Cv^\sharp) &= \{Cv \mid C : (\tau, \tau_a), v \in \gamma_\tau(\mathcal{A}, v^\sharp)\} \\
 \gamma_{\tau_1 \times \dots \times \tau_n}(\mathcal{A}, t^\sharp) &= \bigcup_{(v_1^\sharp, \dots, v_n^\sharp) \in t^\sharp} \gamma_{\tau_1}(\mathcal{A}, v_1^\sharp) \times \dots \times \gamma_{\tau_n}(\mathcal{A}, v_n^\sharp) \\
 \gamma_{\tau_1 \rightarrow \tau_2}(\mathcal{A}, F) &= \{(f, n) \mid (f, n) \in F\} \cup \{(p, S, E) \mid (p, S, E^\sharp) \in F \wedge E \in \gamma_{env}(\mathcal{A}, E^\sharp)\} \\
 \gamma_{env}(\mathcal{A}, E^\sharp) &= \left\{ E \mid \begin{array}{l} \Gamma \vdash E \quad \wedge \quad \Gamma \vdash E^\sharp \quad \wedge \quad \Gamma(x) = \tau \\ \text{dom } E = \text{dom } E^\sharp, E(x) \in \gamma_\tau(\mathcal{A}, E^\sharp(x)) \end{array} \right\} \\
 \gamma(\mathcal{A}, \perp_\tau) &= \emptyset \\
 \gamma(\mathcal{A}, \top_\tau) &= V^\sharp(\tau)
 \end{aligned}$$

Lemma 1. If for all unspecified types τ_u , γ_{τ_u} is monotonic on both arguments, then for all τ , γ_τ is also monotonic on both arguments.

5.5 Correctness of the Abstract Interpretation

Definition 1. Let f be an unspecified term of type $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau$ where $\text{na}(\tau)$. We say that $\llbracket f \rrbracket^\sharp$ is a *sound approximation* of $\llbracket f \rrbracket^{\text{ppt}}$ iff $\forall v_i \in V_{\text{prg}}^{\text{ppt}}(\tau_i), \forall v_i^\sharp \in V_{\text{prg}}^\sharp(\tau_i)$, and for all abstract state \mathcal{A} , if

$$\llbracket f \rrbracket^\sharp(\mathcal{A}, v_1^\sharp, \dots, v_n^\sharp) = \mathcal{A}', w^\sharp \implies \llbracket f \rrbracket^{\text{ppt}}(v_1, \dots, v_n) \subseteq \gamma_\tau(\mathcal{A}', w^\sharp)$$

We state the following theorem of correction that states that the abstract interpretation computes a sound approximation of the big-step interpretation.

Theorem 1. Suppose $E, S \Downarrow v$ and $\epsilon, \mathcal{A}_0, E^\sharp, S \Downarrow^\sharp v^\sharp, \mathcal{A}$ and $\Gamma \vdash E$ and $\Gamma \vdash E^\sharp$ and $\Gamma \vdash S : \tau$ and $E \in \gamma(\mathcal{A}, E^\sharp)$.

Suppose $\forall (\text{val } x : \tau) \in \mathcal{S}, \llbracket x \rrbracket^\sharp$ is a sound approximation of $\llbracket x \rrbracket^{\text{ppt}}$.

Then, $v \in \gamma_\tau(\mathcal{A}, v^\sharp)$.

Therefore, to prove the soundness of the analysis, it is sufficient to prove that the abstract specifications of terms are sound approximation of the concrete specifications of terms.

6 Toward a Control Flow Analysis for λ -calculus

In this section we show how the abstract interpretation of Skel can be used to define a Control Flow Analysis (CFA) from the Skeletal Semantics of λ -calculus. A Control Flow Analysis computes an approximation of the Control Flow for higher-order languages. We aim at defining an analysis similar to 0-CFA for λ -calculus. Let t be a λ -term, the result of a 0-CFA for term t is two maps. The first one maps variables of t to an approximation of the values the variable can be bound to. The second one maps sub-terms of t to an approximation of the results of the evaluation of the sub-term.

In this section, we call **prg** the λ -term to be analyzed.

6.1 The State of the Abstract Interpretation

For our analysis, the state of the abstract interpretation contains two maps that we call C and ρ :

$$\begin{aligned} \mathcal{A} \cdot C : \text{ppt} &\rightarrow \mathcal{P}_f(V_{\text{prg}}^\sharp(\text{ident}) \times V_{\text{prg}}^\sharp(\text{lterm})) \\ \mathcal{A} \cdot \rho : \text{ppt} &\rightarrow V_{\text{prg}}^\sharp(\text{ident}) \rightarrow \mathcal{P}_f(V_{\text{prg}}^\sharp(\text{ident}) \times V_{\text{prg}}^\sharp(\text{lterm})) \end{aligned}$$

The C function maps a program point to an approximation of the result of evaluation of the sub-term of **prg** at the program point. For a program point pp , $C(\text{pp})$ is a set of pairs (x, t) , that are really λ -abstractions $\lambda x.t$.

ρ is a function from program points to some abstraction of an environment of λ -calculus. In classic 0-CFA, there is only one global environment, whereas in our analysis, there is one environment per program point. Practically, it means that to evaluate term t , a subterm of the main program **prg** at program point pp , we use the abstract environment $\mathcal{A} \cdot \rho(\text{pp})$.

6.2 Specification of the Unspecified Types and Unspecified Terms

As for the big-step semantics with program points, the λ -terms are program types: $\mathcal{T} = \{\mathbf{lterm}\}$.

We give the specifications of the unspecified types.

$$\frac{id \in \{x, y, z \dots\}}{\vdash^{\#} id \in V_{\mathbf{prg}}^{\#}(ident)} \text{IDENT} \qquad \frac{pp \in \mathbf{ppt}}{\vdash^{\#} pp \in V_{\mathbf{prg}}^{\#}(env)} \text{ENV}$$

$V_{\mathbf{prg}}^{\#}(ident)$ is a set of variables. An abstract environment $pp_e \in V_{\mathbf{prg}}^{\#}(env)$ is a program point and refers to $\mathcal{A} \cdot \rho(pp_e)$ which maps variables to a set of pairs representing λ -abstractions. The definitions of the unspecified terms are:

$$\begin{aligned} \llbracket getEnv \rrbracket^{\#}(\mathcal{A}, x, pp_e) &= \mathcal{A}, Clos(\{(y, pp \cdot 1, pp) \mid (y, pp \cdot 1) \in \mathcal{A} \cdot \rho(pp_e)(x)\}) \\ \llbracket extEnv \rrbracket^{\#}(\mathcal{A}, pp_e, x, c) &= \{ \mathcal{A} \text{ with } \rho = \rho[pp_v \rightarrow \mathcal{A} \cdot \rho(pp_e)[x \rightarrow \mathcal{A} \cdot \rho(pp_e)(x) \cup c]] \}, pp_v \quad pp_v \text{ fresh} \end{aligned}$$

$\mathcal{A} \cdot \rho(pp_e)(x)$ contains pairs of the form $(y, pp \cdot 1)$ representing λ -abstractions, therefore $\mathbf{prg}@pp = Lam(y, t)$. t has a program point that ends with 1 because it is the second element of the Lam constructor. $getEnv$ type constraint the result to be in $V_{\mathbf{prg}}^{\#}(clos)$, so we need to attach an environment to our pairs $(y, pp \cdot 1)$. Because $(y, pp \cdot 1)$ denotes a λ -abstraction defined at program point pp , the associated environment is pp .

$\llbracket extEnv \rrbracket^{\#}(\mathcal{A}, pp_e, x, c)$ does two things: it modifies the state of the interpretation, and returns a *virtual* program point. Indeed, $extEnv$ returns a new environment, therefore a program point, but what program point should be returned? We do not know until we evaluate a new \mathbf{lterm} with this environment. Therefore, we create virtual program point that will be linked to a real program point later. The $\llbracket extEnv \rrbracket^{\#}$ modifies the abstract state such that $\rho(pp_v)$ contains $\rho(pp_e)$ plus the following constraint: $\rho(pp_v)(x) = \rho(pp_e)(x) \cup c$.

There remains to define our update functions:

$$\begin{aligned} \text{update}_{eval}^{in}(\mathcal{A}, [pp_e, pp_t]) &= \{ \mathcal{A} \text{ with } \rho = \mathcal{A} \cdot \rho[pp_t \rightarrow \mathcal{A} \cdot \rho(pp_t) \sqcup^{\#} \mathcal{A} \cdot \rho(pp_e)] \}, [pp_t, pp_t] \\ \text{update}_{eval}^{out}(\mathcal{A}, [pp_e, pp_t], c) &= \{ \mathcal{A} \text{ with } C = C[pp_t \rightarrow \mathcal{A} \cdot C(pp) \cup c] \}, c \end{aligned}$$

The first update function is used before performing the call $eval \ pp_e \ pp_t$, and the second update function is used after the call was done.

pp_e has type env and therefore is a program point which refers to $\mathcal{A} \cdot \rho(pp_e)$. pp_t has type \mathbf{lterm} , and therefore is also a program point denoting the sub-term of \mathbf{prg} at pp_t . Because we want to compute the sub-term at program point pp_t , we should use the abstract environment $\mathcal{A} \cdot \rho(pp_t)$. Therefore, the abstract state needs to be modified such that $\mathcal{A} \cdot \rho(pp_e)$ is included into $\mathcal{A} \cdot \rho(pp_t)$, and this is what the first update function does.

The second update function is called after the call is performed. The only thing does is to record the result into the map $\mathcal{A} \cdot C$.

A derivation gives a CFA:

$$\epsilon, \mathcal{A}_0, \{e \mapsto [], l \mapsto \underline{\epsilon}\}, eval \ e \ l \Downarrow^{\#} \mathcal{A}, v^{\#}$$

The derivation is implicitly parameterized by the λ -term \mathbf{prg} . In the initial environment, l is mapped to the program point $\underline{\epsilon}$, the root of the λ -term \mathbf{prg} . e is mapped to an empty

environment of λ -calculus. \mathcal{A}_0 is the initial state of the abstract interpretation with empty mappings $\mathcal{A}_0 \cdot \rho$ and $\mathcal{A}_0 \cdot C$. The result of the derivation is a value v^\sharp , but most importantly a new state of the interpretation \mathcal{A} , that contains mappings $\mathcal{A} \cdot \rho$ and $\mathcal{A} \cdot C$ that are the results of the CFA: $\mathcal{A} \cdot \rho$ gives an abstraction of what closures the variables in **prg** can be bound to, and $\mathcal{A} \cdot C$ gives an abstraction of the closures that can appear at a given program point.

7 Implementation

This work resulted in an implementation of an Abstract Interpreter Generator [8]: given a skeletal semantics, a specification of unspecified types and terms, it generates an abstract interpreter. We have used it to try our CFA analysis defined in previous sections, and we experimentally compared its output to another CFA program and we found no differences on the examples we tested. So far, our CFA has not been proven incorrect but it remains to do a formal proof or correction.

Moreover, the Abstract Interpreter Generator has been used to generate an abstract interpreter for a small imperative language, with basic integer arithmetic, conditional branchings and loops. For instance, one can do a basic interval analysis. However, we did not do relational analyses.

8 Conclusion

We presented our work to generate a CFA analysis for λ -calculus from a skeletal semantics. We presented what are Skeletal Semantics and how they can be interpreted to define a semantics. The strength of this approach is that one only needs to define the unspecified types and terms to mechanize a big-step semantics for a language that has a Skeletal Semantics. An example of how the big-step semantics of λ -calculus could be mechanized using Skeletal Semantics was given using the big-step semantics of Skel. Then we presented our first contribution which is a big-step semantics of Skel with program points. Our second contribution which is an abstract interpretation of Skel, and we stated a theorem of correction. The proof is a work in progress. Our final contribution is a CFA analysis built using the abstract interpretation of Skel for λ -calculus. We implemented an abstract interpreter generator that produces an analyzer from a Skeletal Semantics. We used this program to generate a CFA analyzer for λ -calculus. Our analyzer was experimentally tested and gives similar results to other 0-CFA analyzers.

However, when mechanizing a semantics using an intermediate language like Skel, we expect to lose precision compared to a language specific semantics. Moreover, there are several possible definitions of a Skeletal Semantics for given a language, the abstract interpretation precision may depend on the definition of the Skeletal Semantics. It is a language-independent approach that generates an abstract interpretation for languages with a Skeletal Semantics.

The proof of correction between the big-step semantics and the abstract interpretation must be mechanized. Moreover, our CFA analysis must be compared to other CFA analyses, in particular 0-CFA analysis and verify if we are systematically at least as precise as 0-CFA. Finally, there are not proofs of termination of the abstract interpretation, and this should be addressed.

References

- [1] Bodin, M., Gardner, P., Jensen, T., Schmitt, A.: Skeletal semantics and their interpretations. *Proceedings of the ACM on Programming Languages* **3**(POPL), 1–31 (2019)
- [2] Calcagno, C., Distefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O’Hearn, P., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: *NASA Formal Methods Symposium*. pp. 3–11. Springer (2015)
- [3] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. pp. 238–252 (1977)
- [4] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astrée analyzer. In: *European Symposium on Programming*. pp. 21–30. Springer (2005)
- [5] Noizet, L.: Necro Library, <https://gitlab.inria.fr/skeletons/necro>, <https://gitlab.inria.fr/skeletons/necro>
- [6] Noizet, L., Schmitt, A.: Semantics in Skel and Necro. In: *ICTCS 2022 - Italian Conference on Theoretical Computer Science*. CEUR Workshop Proceedings, Rome, Italy (Sep 2022)
- [7] Roşu, G., Şerbănuţă, T.F.: An overview of the k semantic framework. *The Journal of Logic and Algebraic Programming* **79**(6), 397–434 (2010)
- [8] Rébiscoul, V.: Abstract Interpreter Generator, <https://gitlab.inria.fr/skeletons/abstract-interpreter-generator>
- [9] Sabry, A., Felleisen, M.: Reasoning about programs in continuation-passing style. In: *LISP AND SYMBOLIC COMPUTATION*. pp. 288–298 (1993)
- [10] Schmidt, D.A.: Natural-semantics-based abstract interpretation (preliminary version). In: *International Static Analysis Symposium*. pp. 1–18. Springer (1995)

A Skeletal Semantics of λ -calculus

```

type ident
type env

type clos =
| Clos (ident, lterm, env)

type lterm =
| Lam (ident, lterm)
| Var ident
| App (lterm, lterm)

val extEnv : (env, ident, clos) → env
val getEnv : (ident, env) → clos

val eval (s:env) (l:lterm): clos =
  branch
    let Lam (x, t) = l in
    Clos (x, t, s)
  or
    let Var x = l in
    getEnv (x, s)

```

```

or
  let App (t1, t2) = 1 in
  let Clos (x, t, s') = eval s t1 in
  let w = eval s t2 in
  let s'' = extEnv (s', x, w) in
  eval s'' t
end
    
```

B Typing Rules of Skeletons and Terms

$$\begin{array}{c}
 \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \text{VAR} \\
 \\
 \frac{\text{val } x : \tau [= t] \in \mathcal{S}}{\Gamma \vdash x : \tau} \text{TERMDEF} \qquad \frac{\Gamma \vdash t : \tau \quad C : (\tau, \tau')}{\Gamma \vdash Ct : \tau'} \text{CONST} \\
 \\
 \frac{\forall i, \Gamma \vdash t_i : \tau_i}{\Gamma \vdash (t_1, \dots, t_n) : (\tau_1, \dots, \tau_n)} \text{TUPLE} \qquad \frac{\Gamma + p \leftarrow \tau \vdash S : \tau'}{\Gamma \vdash (\lambda p : \tau \rightarrow S) : \tau \rightarrow \tau'} \text{FUN} \\
 \\
 \frac{\Gamma \vdash S_1 : \tau \quad \dots \quad \Gamma \vdash S_n : \tau}{\Gamma \vdash (S_1 \dots S_n) : \tau} \text{BRANCH} \qquad \frac{\Gamma \vdash S : \tau \quad \Gamma + p \leftarrow \tau \vdash S' : \tau'}{\Gamma \vdash \text{let } p = S \text{ in } S' : \tau'} \text{LETIN} \\
 \\
 \frac{\Gamma \vdash t_0 : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \quad \forall i \Gamma \vdash t_i : \tau_i}{\Gamma \vdash (t_0 \ t_1 \dots t_n) : \tau} \text{APP}
 \end{array}$$

C The Functions of a Skeletal Semantics \mathcal{S}

$$\begin{aligned}
 \text{Funs}(\Gamma, \text{let } p = S_1 \text{ in } S_2) &= \text{Funs}(\Gamma, S_1) \cup \text{Funs}(\Gamma + p \leftarrow \tau, S_2) \\
 \text{Funs}(\Gamma, \text{branch } S_1 \dots S_n \text{ end}) &= \bigcup_{i=1}^n \text{Funs}(\Gamma, S_i) \qquad \text{Funs}(\Gamma, t_0 \ t_1 \dots t_n) = \bigcup_{i=0}^n \text{Funs}(\Gamma, t_i) \\
 \text{Funs}(\Gamma, \lambda p : \tau \rightarrow S_0) &= \{ \Gamma, \lambda p : \tau \rightarrow S_0 \} \cup \text{Funs}(\Gamma + p \leftarrow \tau, S_0) \\
 \text{Funs}(\Gamma, (t_1, \dots, t_n)) &= \bigcup_{i=1}^n \text{Funs}(\Gamma, t_i) \qquad \text{Funs}(\Gamma, Ct) = \text{Funs}(\Gamma, t) \qquad \text{Funs}(\Gamma, x) = \emptyset
 \end{aligned}$$

The set of λ -abstractions in the Skeletal Semantics \mathcal{S} is:

$$\text{Funs}(\mathcal{S}) \equiv \bigcup_{\text{val } x : \tau = t \in \mathcal{S}} \text{Funs}(\emptyset, t)$$

D Correction of Big-Step Interpretation with Program Points

Definition 2. To relate element with or without program points, we define the following function γ , such that $\forall \tau, \gamma_\tau \in V_{\mathbf{prg}}^{\mathbf{ppt}}(\tau) \rightarrow V(\tau)$. The function γ is parameterized by the global value of type $\tau \in \mathcal{T}$ and satisfies the following constraints.

- $\gamma_\tau(\mathbf{pp}) = \mathbf{prg}@ \mathbf{pp}$ and $\tau \in \mathcal{T}$
- $\gamma_{\tau_1 \times \dots \times \tau_n}((v'_1, \dots, v'_n)) = (\gamma_{\tau_1}(v'_1), \dots, \gamma_{\tau_n}(v'_n))$
- $\gamma_{\tau_a}(C v') = C \gamma_\tau(v')$ with $C : (\tau, \tau_a)$
- Suppose $\Gamma \vdash E$, and $\Gamma \vdash E'$
 $\gamma_{env}(E') = E \iff \text{dom } E' = \text{dom } E \wedge \forall x \in \text{dom } E', \gamma_{\Gamma(x)}(E'(x)) = E(x)$
- $\gamma_{\tau_1 \rightarrow \tau_2}((p, S, E')) = (p, S, \gamma_{env}(E'))$
- Suppose $\mathbf{val } f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau [= t] \in \mathcal{S}$, then
 $\gamma_\tau((f, [v'_1, \dots, v'_n], k)) = (f, [\gamma_{\tau_1}(v'_1), \dots, \gamma_{\tau_n}(v'_n)], k)$

Definition 3. Take f such that $\mathbf{val } f : \tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \tau \in \mathcal{S}$:

$$\begin{aligned} \gamma_{unspec}(\llbracket f \rrbracket^{\mathbf{ppt}}) = \llbracket f \rrbracket &\iff \forall (v'_i, v_i) \in V_{\mathbf{prg}}^{\mathbf{ppt}}(\tau_i) \times V(\tau_i) \text{ such that } \gamma_{\tau_i}(v'_i)v_i \\ &\gamma_\tau(\llbracket f \rrbracket^{\mathbf{ppt}}(v'_1, \dots, v'_n)) = \llbracket f \rrbracket(v_1, \dots, v_n) \end{aligned}$$

Theorem 2. Let $\gamma_{env}(E') = E$, and suppose for all unspecified functions f , $\gamma_{unspec}(\llbracket f \rrbracket^{\mathbf{ppt}}) = \llbracket f \rrbracket$, then:

$$E, S \Downarrow v \implies \exists v', \quad E', S \Downarrow^{PP} v' \text{ and } \gamma_\tau(v') = v$$

Traduire l’univers des mathématiques en DEDUKTI, sans univers

Amélie Ledein^{1*} et Elliot Butte²

¹ Université Paris-Saclay, Inria, CNRS, ENS Paris-Saclay,
Laboratoire Méthodes Formelles, France

`amelie.ledein@inria.fr`

² ENSIIE, France

`elliott.butte@ensiie.fr`

Ces dernières années, le nombre d’assistants à la preuve, de prouveurs automatiques et de vérificateurs de preuve n’a cessé de croître. L’un d’entre eux, le *framework* logique DEDUKTI, a pour objectif de rendre ces outils formels interopérables, c’est-à-dire rendre possible la réutilisation des preuves d’un outil A dans un outil B . Un autre outil formel, METAMATH, a la particularité d’être à la 4e place dans la liste des systèmes possédant le plus grand nombre de preuves parmi la liste des 100 théorèmes à prouver de Freek Wiedijk, tout en étant constitué de très peu de fonctionnalités : pas de type dépendant, pas de polymorphisme, ni de notion d’univers.

Afin d’agréments le nombre de preuves qu’il est possible de traduire dans DEDUKTI, nous présentons Mathiilde, un traducteur automatique de METAMATH vers DEDUKTI. Comme ce traducteur peut utiliser deux encodages différents de METAMATH vers DEDUKTI, nous discutons les avantages et les inconvénients du point de vue de l’interopérabilité.

1 Introduction

DEDUKTI [3], un *framework* logique basé sur le $\lambda\Pi$ -CALCUL MODULO THÉORIE, a pour principal objectif l’interopérabilité en se voulant être un standard dans lequel il est possible de définir n’importe quelle logique, mais également avec lequel la traduction d’une logique vers une autre est plus aisée.

Deux cas d’étude sont basés sur une approche utilisant la théorie \mathcal{U} [5], c’est-à-dire ayant pour objectif de simplifier la preuve pour qu’elle soit exprimable dans une logique plus faible, ici une extension, nommée STTV [15], de la Théorie des Types Simples avec du polymorphisme prénexe et des opérateurs de type, afin de pouvoir la traduire plus facilement vers d’autres assistants de preuve. En effet, F. Thiré a traduit le petit théorème de Fermat [14, 15], écrit en MATITA, tandis que Y. Gérard a traduit le livre I des Éléments d’Euclide [2], écrit en COQ, vers HOL Light, LEAN, MATITA, OpenTheory (donc ISABELLE/HOL et HOL 4), et PVS.

Une autre approche, utilisant l’alignement des concepts, a été suivie dans un autre cas d’étude. A. Assaf, R. Cauderlier et C. Dubois [4, 9] ont reconstruit au sein de DEDUKTI la preuve du Crible d’Eratosthène initialement écrite en COQ, avec les entiers naturels de HOL. Pour ce faire, il a fallu démontrer des relations de morphisme entre les entiers de COQ et les entiers de HOL. Des travaux en cours s’intéressent également à l’alignement des concepts, c’est-à-dire à des procédés permettant d’identifier des concepts équivalents, comme par exemple les entiers naturels de HOL, de PVS ou encore de COQ, et ainsi éviter la redondance de ces concepts lors de la traduction.

*Financée par Digicosme.

METAMATH [13] est un *framework* logique dans lequel de nombreux résultats mathématiques ont été formalisés, comme par exemple 74 problèmes sur les 100 faisant partie de la liste de Freek Wiedijk [1]. Ce résultat est particulièrement surprenant au vu du langage décrit dans le Metamath-book [13], puisque celui-ci ne permet de définir que des constantes, des axiomes et des preuves, sans type dépendant, ni polymorphisme, ni univers. Malheureusement, à notre connaissance, de très rares références existent à propos de METAMATH [13, 10, 6, 7, 8], comme une formalisation en METAMATH de la Matching Logic [10], mais ces références ne sont pas suffisantes pour formaliser ce *framework* logique. La référence principale est le Metamath-book [13] mais, par exemple, celui-ci ne présente pas de grammaire formelle du langage de METAMATH, mais seulement une description en anglais.

Ainsi, la première contribution de cet article est constituée d'une proposition d'une grammaire formelle du langage de METAMATH ainsi que d'une formalisation papier de l'étape de normalisation effectuée par au moins l'implémentation de référence d'un vérificateur de preuve pour METAMATH. En plus de cette formalisation partielle de METAMATH, cet article propose deux encodages de METAMATH vers DEDUKTI, l'un plus proche de la philosophie de METAMATH, l'autre plus proche des fonctionnalités courantes d'un assistant de preuve. La traduction automatique de ces encodages est en cours de développement¹, mais le prototype actuel permet déjà de récupérer une part non-négligeable de la bibliothèque standard de METAMATH à l'aide de notre premier encodage, mais également environ 1 000 preuves écrites dans la logique propositionnelle à la Hilbert à l'aide du deuxième encodage. Ce prototype peut également être vu comme un nouveau vérificateur de METAMATH écrit en OCAML, puisque la construction du λ -terme vérifié par la suite en DEDUKTI est similaire à la méthode de vérification de preuve effectuée par METAMATH.

Après une brève présentation de DEDUKTI (Section 2), nous présentons plus en détails METAMATH, notamment à l'aide d'un exemple d'axiomatisation et de preuve, mais également en expliquant la notion de portée et comment normaliser tout fichier METAMATH (Section 3). Ensuite, nous proposons un premier encodage profond de METAMATH en DEDUKTI, qui se veut être au plus proche de la philosophie même de METAMATH (Section 4). De plus, nous proposons un deuxième encodage en DEDUKTI, plus lisible, qui est cette fois superficiel (Section 5). Ces deux encodages se basent sur une forme normalisée des fichiers METAMATH, comme le montre la figure 1. Enfin, nous présentons les résultats de traduction de la bibliothèque standard de METAMATH pour les différents encodages proposés (Section 6).

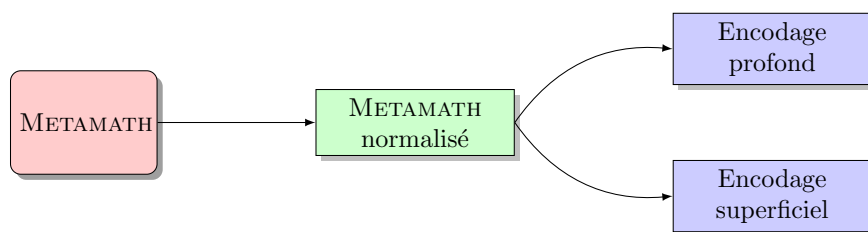


FIGURE 1 – Vue d'ensemble des deux traductions possibles de METAMATH vers DEDUKTI

Dans la suite, les mots-clés d'un langage ou ce qui est natif dans un langage seront distingués par de la couleur. Le langage de DEDUKTI se différenciera par une **couleur bleue**, tandis que le langage de METAMATH sera différencié par une **couleur bordeaux**. Celles-ci facilitent la lecture, mais ne sont pas nécessaires à la compréhension.

1. <https://gitlab.com/semantiko/MM2DK/translator>

2 Dedukti

DEDUKTI [3] est un *framework* logique basé sur le $\lambda\Pi$ -CALCUL MODULO THÉORIE, abrégé $\lambda\Pi\equiv_{\mathcal{T}}$, un λ -calcul avec types dépendants introduit par Cousineau et Dowek [11], et ayant la particularité de posséder une notion primitive de calcul définie à l'aide de règles de réécriture [12]. Plusieurs logiques ont déjà été encodées au sein de DEDUKTI, comme la logique des prédicats ou encore le Calcul des Constructions Inductives. Diverses logiques ont pu être encodées au sein de DEDUKTI, ce qui favorise la possibilité de rendre interopérables les preuves entre différents outils formels comme COQ ou PVS (Figure 2).

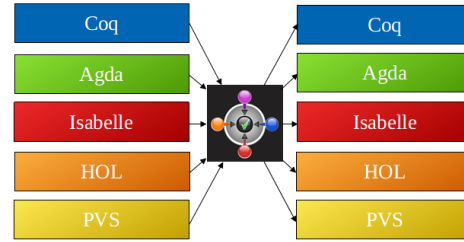


FIGURE 2 – L’interopérabilité des preuves grâce à DEDUKTI.

Dans cette section, nous ne présentons que les fonctionnalités disponibles dans DEDUKTI sur lesquelles nous nous appuyons dans la suite et qui permettront de faciliter, ultérieurement, les traductions vers d’autres formalismes.

Typage et symboles. La syntaxe du $\lambda\Pi\equiv_{\mathcal{T}}$ est directement accessible dans DEDUKTI : **TYPE** (sorte native), λ (abstraction) et Π (produit dépendant). Quand le produit dépendant $\Pi(x : A), B$ est en réalité non-dépendant, c’est-à-dire quand x n’est pas une variable libre de B , nous notons $A \rightarrow B$.

La signature est définie à partir de symboles qui sont composés d’un nom et d’un type exprimable dans le $\lambda\Pi\equiv_{\mathcal{T}}$. Si la déclaration d’un symbole est faite avec le mot-clé **symbol** seul, on dit que le symbole est *défini*, sans aucune propriété particulière, alors qu’avec le mot-clé supplémentaire **constant**, le symbole est dit *constant* et ne peut pas être réduit par une règle de réécriture.

L’exemple suivant définit le type des entiers naturels ainsi que les deux constructeurs usuels.

```
1 constant symbol Nat : TYPE;
2 constant symbol 0 : Nat;
3 constant symbol S : Nat → Nat;
```

Règles de réécriture. Dans DEDUKTI, une règle de réécriture s’écrit **rule** $LHS \leftrightarrow RHS$ dans laquelle les variables libres sont notées $\$x$, $\$y$, etc. Il est possible d’y utiliser un joker ($-$) à gauche (LHS) lorsqu’une variable n’est pas utilisée dans la partie droite (RHS).

L’exemple suivant définit l’addition sur les entiers naturels définis précédemment.

```
4 symbol + : Nat → Nat → Nat;
5 rule $m + 0 ↔ $m;
6 rule $m + (S $n) ↔ S ($m + $n);
```

Les règles de réécriture autorisent l’ordre supérieur, peuvent être non linéaires et ne s’appliquent pas forcément en tête de terme, mais ne sont pas conditionnelles.

3 Metamath

L'objectif de cette section est de présenter METAMATH [13]. Nous commençons par présenter comment définir une axiomatisation dans METAMATH, ainsi que comment effectuer une preuve. Ensuite, nous expliquons qu'il est possible d'alléger la formalisation effectuée en METAMATH à l'aide de la notion de portée, sans étendre l'expressivité de METAMATH. Cette section présente également notre proposition de grammaire pour le langage de METAMATH, ainsi que notre formalisation papier de l'étape de normalisation d'un fichier METAMATH.

3.1 Écrire une axiomatisation

Le langage de METAMATH permet de déclarer des constantes (**\$c**), des variables (**\$v**), le type de ces variables (**\$f**), des contraintes sur ces variables (**\$d**), ainsi que des hypothèses logiques (**\$e**) associées soit à des axiomes (**\$a**), soit à des théorèmes (**\$p**). Les hypothèses commençant par **\$f**, **\$d** ou **\$e** sont nommées respectivement *f-hypothèses*, *d-hypothèses* ou *e-hypothèses*. La figure 3 propose un extrait de la logique propositionnelle à la Hilbert définie en METAMATH, où l'axiome **wi** correspond à la bonne formation de l'implication, et les axiomes **a2** et **mp** correspondent aux règles d'inférence suivantes :

$$\frac{}{\vdash ((\phi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\phi \rightarrow \psi) \rightarrow (\phi \rightarrow \chi)))} \text{a2} \qquad \frac{\vdash \phi \quad \vdash \phi \rightarrow \psi}{\vdash \psi} \text{mp}$$

```

$c ( ) wff -> | - $.

${ $v \phi \psi $.
  wph $f wff \phi $.
  wps $f wff \psi $.
  wi $a wff ( \phi -> \psi ) $. }

${ $v \phi \psi \chi $.
  wph2 $f wff \phi $.
  wps2 $f wff \psi $.
  wch $f wff \chi $.
  a2 $a | - ((\phi -> (\psi -> \chi)) -> ((\phi -> \psi) -> (\phi -> \chi))) $. }

${ $v \phi \psi $.
  wph3 $f wff \phi $.
  wps3 $f wff \psi $.
  min $e | - \phi $.
  maj $e | - ( \phi -> \psi ) $.
  mp $a | - \psi $. }

```

FIGURE 3 – Extrait de la logique propositionnelle à la Hilbert définie en METAMATH

Certaines déclarations possèdent des labels, comme **wi**, **wch** ou encore **mp** : ceux-ci sont utilisés lors de l'élaboration de la preuve et sont donc uniques. D'un point de vue logique, une théorie définie en METAMATH n'est donc composée que de constantes, d'axiomes et de théorèmes. Une constante correspond à un *token*, tandis que l'énoncé d'une hypothèse logique, d'un axiome ou d'un théorème est une liste non vide de *tokens*. Une variable peut être substituée par une liste non vide de *tokens*². Ainsi, un fichier METAMATH peut être vu comme un triplet $(\Sigma, \mathcal{A}, \mathcal{T})$, où la signature Σ correspond à l'ensemble des constantes, \mathcal{A} est l'ensemble des axiomes et \mathcal{T} est l'ensemble des théorèmes.

La section suivante explique comment faire une preuve en METAMATH.

2. Une option existe pour autoriser une variable à être substituée par aucun *token*, mais cette option est considérée *unsafe* par METAMATH.

3.2 Faire une preuve

Le mécanisme de vérification de preuve de METAMATH est basé sur la substitution de variables à l'aide d'une pile. La sous-section 3.2.1 illustre ce mécanisme sur un exemple, tandis que la sous-section 3.2.2 présente plus en détails les contraintes qu'il est possible d'exprimer sur les variables lors de la substitution.

3.2.1 Mécanisme général

Un théorème en METAMATH est composé d'un énoncé, lui-même composé ou non de plusieurs hypothèses, et d'une preuve, c'est-à-dire d'une liste de labels. Un label est associé à une f -hypothèse, une e -hypothèse, un axiome ou un théorème. Par exemple, la preuve du théorème

$$\frac{\vdash (\phi \rightarrow (\psi \rightarrow \chi))}{\vdash ((\phi \rightarrow \psi) \rightarrow (\phi \rightarrow \chi))} \text{ a2i, écrite en METAMATH, est disponible à la figure 4.}$$

```
hyp $e |- (\phi -> (\psi -> \chi)) $.
a2i $p |-( (\phi -> \psi) -> (\phi -> \chi)) $=
  wph ① wps wch ② wi ③ wi ④
  wph wps wi wph wch wi wi ⑤
  hyp ⑥
  wph wps wch a2 ⑦
  mp ⑧ $.
```

FIGURE 4 – Exemple de preuve écrite en METAMATH

Sur le même principe que la notation polonaise inversée, le mécanisme de vérification de preuve de METAMATH utilise une pile évitant toute ambiguïté sans avoir recours à l'utilisation de parenthèses au sein de la preuve. Les figures 5 à 12 illustrent ce mécanisme de vérification sur la preuve du théorème a2i. La pile de la figure 5 contient uniquement **wff** ϕ , qui correspond au contenu associé au label **wph**. La pile de la figure 6 est obtenue en empilant du haut vers le bas les éléments **wff** ψ et **wff** χ , respectivement associés aux labels **wps** et **wch**. Ensuite, comme l'axiome **wi** possède deux variables, deux f -hypothèses lui sont associées, et donc deux éléments sont dépilés du bas vers le haut de la pile. METAMATH cherche ensuite à unifier le premier élément récupéré de la pile avec la f -hypothèse **wps**, et le deuxième élément récupéré de la pile avec la f -hypothèse **wph**. La substitution $\{ \phi \mapsto \psi; \psi \mapsto \chi \}$ est ainsi obtenue, et le nouvel élément **wff** $(\psi \rightarrow \chi)$ est mis sur la pile, comme le montre la figure 7. Le même procédé permet d'obtenir la pile de la figure 8, avec la substitution $\{ \phi \mapsto \phi; \psi \mapsto (\psi \rightarrow \chi) \}$. La suite de labels **wph wps wi wph wch wi wi** permet de construire la pile de la figure 9 en empilant le nouvel élément **wff** $((\phi \rightarrow \psi) \rightarrow (\phi \rightarrow \chi))$. Ensuite, le label **hyp** met le contenu de l'hypothèse **hyp** sur la pile, comme le montre la figure 10. Enfin, l'élément en tête de la pile de la figure 11 est obtenu grâce à la suite de labels **wph wps wch a2**, et le label **mp** permet de finir la preuve, puisque le résultat de l'unification est syntaxiquement égal au contenu associé au label **a2i**, comme le montre la figure 12.

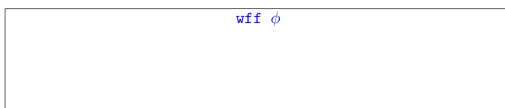


FIGURE 5 – État de la pile en ①

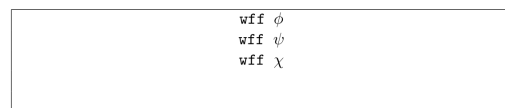


FIGURE 6 – État de la pile en ②

```
wff  $\phi$ 
wff (  $\psi \rightarrow \chi$  )
```

FIGURE 7 – État de la pile en ③

```
wff (  $\phi \rightarrow ( \psi \rightarrow \chi )$  )
```

FIGURE 8 – État de la pile en ④

```
wff (  $\phi \rightarrow ( \psi \rightarrow \chi )$  )
wff ( (  $\phi \rightarrow \psi$  )  $\rightarrow$  (  $\phi \rightarrow \chi$  ) )
```

FIGURE 9 – État de la pile en ⑤

```
wff (  $\phi \rightarrow ( \psi \rightarrow \chi )$  )
wff ( (  $\phi \rightarrow \psi$  )  $\rightarrow$  (  $\phi \rightarrow \chi$  ) )
|- (  $\phi \rightarrow ( \psi \rightarrow \chi )$  )
```

FIGURE 10 – État de la pile en ⑥

```
wff (  $\phi \rightarrow ( \psi \rightarrow \chi )$  )
wff ( (  $\phi \rightarrow \psi$  )  $\rightarrow$  (  $\phi \rightarrow \chi$  ) )
|- (  $\phi \rightarrow ( \psi \rightarrow \chi )$  )
|- ( (  $\phi \rightarrow ( \psi \rightarrow \chi )$  )  $\rightarrow$  (  $\phi \rightarrow \psi$  )  $\rightarrow$  (  $\phi \rightarrow \chi$  ) )
```

FIGURE 11 – État de la pile en ⑦

```
|- ( (  $\phi \rightarrow \psi$  )  $\rightarrow$  (  $\phi \rightarrow \chi$  ) )
```

FIGURE 12 – État de la pile en ⑧

La sous-section suivante explique comment sont utilisées les déclarations imposant des contraintes sur les substitutions de variables (**\$d**), lors du processus d'unification.

3.2.2 Une fonctionnalité un peu particulière

La section 3.1 mentionnait des déclarations imposant des contraintes sur les substitutions de variables qu'il est possible d'effectuer. Ces déclarations commencent par la balise **\$d** et ont la sémantique suivante : **\$d x y \$.** signifie que pour toute substitution $[x \mapsto t_1, y \mapsto t_2, \dots]$:

— t_1 et t_2 n'ont pas de variable en commun

— Pour toutes variable v de t_1 et variable w de t_2 , **\$d v w \$.** est vrai dans le contexte local.

Cette fonctionnalité permet par exemple de modéliser le λ -calcul comme écrit ci-dessous, où **Subst l y v l'** signifie que l' correspond à la substitution de y par v dans l .

```
$c term var lam app Subst $.
$v x y l l' v $.
vx $f var x $.      vy $f var y $.
tl $f term l $.     tb $f term l' $.     tv $f term v $.
$d x y $.           $d x v $.
he $e Subst l y v l' $.
as $a Subst (lam x l) y v (lam x l') $.
```

Aucune indication n'existe dans la preuve pour vérifier ces contraintes. METAMATH vérifie en interne les contraintes imposées par ces déclarations, à chaque fois qu'il effectue une substitution.

3.3 Utiliser la notion de portée

Le langage de METAMATH offre un peu plus de souplesse que ce que nous avons pu présenter à la section précédente. En effet, il est possible d'importer un fichier METAMATH dans un autre fichier à l'aide de la commande `$(filename $)`, mais il est également possible de jouer sur la portée des hypothèses pour éviter des redondances (`$(section $)`). L'exemple de la figure 3 est très verbeux car il nécessite de recopier à de nombreuses reprises des déclarations de variables et d'hypothèses identiques (seul le label qui leur est associé est différent car tous les labels doivent être uniques). Plus concrètement, les fichiers A et B de la figure 13 sont définis sur la même signature $\Sigma \triangleq \{ (;) ; \text{nat} ; 0 ; \text{not} ; = ; + ; * ; / \}$, et sont sémantiquement équivalents. En effet, le fichier A peut se lire ainsi :

Axiome comm-plus : Posons a et b , des entiers naturels. Alors $a + b = b + a$.

Axiome eq-div : Posons a, b, c et d , des entiers naturels.

Si c n'est pas égal à 0 et que $a * b = d * c$, alors $(a * b) / c = d$.

Le fichier B peut se lire ainsi :

Posons a, b, c et d , des entiers naturels.

Axiome comm-plus : $a + b = b + a$.

Axiome eq-div : Si c n'est pas égal à 0 et que $a * b = d * c$, alors $(a * b) / c = d$.

Les fichiers A et B sont donc sémantiquement équivalents.

Fichier A	Fichier B
<pre> \$(() nat 0 not = + * / \$. \${ \$v a b \$. nata-p \$f nat a \$. natb-p \$f nat b \$. comm-plus \$a a + b = b + a \$. }\$ \${ \$v a b c d \$. nata-m \$f nat a \$. natb-m \$f nat b \$. natc-m \$f nat c \$. natd-m \$f nat d \$. notzero \$e not (c = 0) \$. eq-mult \$e a * b = d * c \$. eq-div \$a (a * b) / c = d \$. }\$ </pre>	<pre> \$(() nat 0 not = + * / \$. \$v a b c d \$. nata \$f nat a \$. natb \$f nat b \$. natc \$f nat c \$. natd \$f nat d \$. comm-plus \$a a + b = b + a \$. \${ notzero \$e not (c = 0) \$. eq-mult \$e a * b = d * c \$. eq-div \$a (a * b) / c = d \$. }\$ </pre>

FIGURE 13 – Deux exemples de fichiers METAMATH

Il est également possible d'imbriquer des sections les unes dans les autres. La grammaire complète du langage METAMATH que nous proposons est disponible à la figure 14. A notre connaissance, cette grammaire constitue la première formalisation du langage utilisé par METAMATH. Celle-ci découle de notre lecture du Metamath-book [13], ainsi que de l'observation d'une multitude d'exemples.

Basic terminals	$carac ::= a-z \mid A-Z \mid 0-9$ $special-carac ::= ! \mid " \mid \# \mid \% \mid \& \mid ' \mid (\mid) \mid * \mid + \mid , \mid - \mid . \mid / \mid : \mid ; \mid < \mid = \mid > \mid ? \mid @ \mid [\mid \backslash \mid] \mid ^ \mid _ \mid \{ \mid \} \mid \mid \sim$ $white-space ::= \mid \backslash t \mid \backslash r \mid \backslash n \mid \backslash f$ $text ::= (carac \mid special-carac \mid white-space \mid \$)^+$ $filename ::= ((carac \mid special-carac)^+).mm$ $math-symbol ::= (carac \mid special-carac)^+$ $variable ::= math-symbol \quad (\text{declared in variable symbol declaration})$ $typecode ::= math-symbol \quad (\text{declared in constant symbol declaration})$ $label ::= (carac \mid - \mid \mid .)^+$
Comment	$comment ::= \$(text \$) \quad (\text{not nested})$
Included source file name	$import ::= \#[filename^+ \$]$
Constant symbol declaration	$c-decl ::= \$c math-symbol^+ \$.$
Variable symbol declaration	$v-decl ::= \$v math-symbol^+ \$.$
Block	$b-decl ::= \${ (comment \mid v-decl \mid b-decl \mid hypothesis \mid assertion)^* \$}$
Disjoint-variable restriction	$d-hypo ::= \$d variable variable^+ \$.$
Variable-type hypothesis	$f-hypo ::= label \$f typecode variable \$.$
Logical hypothesis	$e-hypo ::= label \$e typecode math-symbol^* \$.$
Axiomatic assertions	$a-assert ::= label \$a typecode math-symbol^* \$.$
Provable assertions	$p-assert ::= label \$p typecode math-symbol^* \$=(label \mid ?)^* \$.$
File	$declaration ::= c-decl \mid v-decl \mid b-decl$ $hypothesis ::= d-hypo \mid f-hypo \mid e-hypo$ $assertion ::= a-assert \mid p-assert$ $file ::= (comment \mid import \mid declaration \mid hypothesis \mid assertion)^*$

FIGURE 14 – Grammaire de METAMATH

De plus, le langage de METAMATH possède très peu de sucre syntaxique. La déclaration $\$c c_1 \dots c_d \$.$ est équivalente aux déclarations $\$c c_1 \$.$... $\$c c_d \$.$, de même que la déclaration $\$v v_1 \dots v_d \$.$ est équivalente aux déclarations $\$c v_1 \$.$... $\$c v_d \$.$. La déclaration qui impose des contraintes sur les variables $\$d v_1 \dots v_b \$.$ correspond aux déclarations $\$d v_i v_j \$.$ pour tout $(i, j) \in [1; b - 1] \times [i + 1; b]$.

L'étude de l'exemple de la figure 13 illustre que deux fichiers METAMATH peuvent être sémantiquement équivalents mais syntaxiquement très différents. D'après le Metamath-book [13], une étape de normalisation est effectuée au moins par le vérificateur de référence de METAMATH, afin d'en simplifier la vérification. Cette étape de normalisation permet de réécrire le fichier B (Figure 13) et ainsi d'obtenir le fichier A (Figure 13), mais cette étape n'est pas formalisée dans le Metamath-book [13]. La section suivante propose une définition d'un fichier dit *normalisé*, tel que tout fichier METAMATH puisse se réécrire en un fichier normalisé.

3.4 Normaliser un fichier Metamath

Jusqu'à présent, nous avons illustré comment écrire une axiomatisation en METAMATH (Section 3.1) ainsi que comment faire une preuve (Section 3.2), mais également expliqué qu'il est possible d'écrire un fichier METAMATH moins verbeux à l'aide de la notion de portée (Section 3.3). Cette section donne un peu plus de sémantique à la syntaxe vue précédemment, en définissant ce qu'est un fichier METAMATH normalisé, puis en proposant une formalisation papier d'une transformation de tout fichier METAMATH en une version normalisée.

3.4.1 Définition d'un fichier normalisé

Par la suite, nous définissons ce qu'est un axiome normalisé, un théorème normalisé ainsi qu'un fichier METAMATH normalisé. Un *axiome normalisé*, une section constituée, dans l'ordre,

d'une liste de variables, d'une liste de f -hypothèses, d'une liste de contraintes sur ces variables, d'une liste de e -hypothèses et d'un énoncé. Un *théorème normalisé* est une section constituée, dans l'ordre, d'une liste de variables, d'une liste de f -hypothèses, d'une liste de contraintes sur ces variables, d'une liste de e -hypothèses, d'un énoncé et d'une preuve de cet énoncé. Un *fichier normalisé* est donc composé, dans l'ordre, d'une ou de plusieurs déclarations de constantes puis d'une liste d'axiomes normalisés ou de théorèmes normalisés.

Notre formalisation de la transformation d'un fichier METAMATH quelconque en une version normalisée est proposée à la section suivante.

3.4.2 Transformation en un fichier Metamath normalisé

Notre formalisation a besoin d'un contexte global noté $\Gamma = (\Sigma, \mathcal{A}, \mathcal{T})$, où Σ correspond à la signature, \mathcal{A} est l'ensemble des axiomes et \mathcal{T} est l'ensemble des théorèmes. Nous utilisons également un contexte local noté $\Delta = [\Delta_0; \dots; \Delta_{n-1}; \Delta_n]$ avec $\Delta_i = (\mathcal{V}_i, \mathcal{F}_i, \mathcal{D}_i, \mathcal{E}_i)$, où \mathcal{V}_i est l'ensemble des variables à la profondeur i , \mathcal{F}_i est l'ensemble des f -hypothèses à la profondeur i , \mathcal{D}_i est l'ensemble des d -hypothèses à la profondeur i et \mathcal{E}_i est l'ensemble des e -hypothèses à la profondeur i . La notion de profondeur ici modélise la profondeur d'imbrication : une variable définie à la racine du fichier est à la profondeur 0, donc appartient à \mathcal{V}_0 , tandis qu'une variable définie dans une section elle-même définie dans une section, sera à la profondeur 2, donc appartient à \mathcal{V}_2 .

La formalisation que nous proposons est disponible à la figure 15, où $l \geq 0$, $d \geq 1$ et $i \geq 0$. Comme les constantes ne peuvent apparaître qu'à la profondeur 0 d'un fichier, leur traduction n'est valide que pour i valant 0.

$\ s_1 \dots s_l \ _{norm}$	$= \Gamma_{init}, \Delta_{init} \mapsto \ s_1 \ _0 \mapsto \dots \mapsto \ s_l \ _0$
$\ \$c \ c_1 \dots c_d \$ \ \ _0$	$= \Gamma \stackrel{\Sigma}{\leftarrow} \Sigma \cup \{ c_1; \dots; c_d \}, \Delta$
$\ \$v \ v_1 \dots v_d \$ \ \ _i$	$= \Gamma, \Delta \stackrel{\mathcal{V}_i}{\leftarrow} \mathcal{V}_i @ [v_1; \dots; v_d]$
$\ lab \ \$f \ t \ v \ \$ \ \ _i$	$= \Gamma, \Delta \stackrel{\mathcal{F}_i}{\leftarrow} \mathcal{F}_i @ [(lab, t, v)]$
$\ \$d \ d_1 \ d_2 \$ \ \ _i$	$= \Gamma, \Delta \stackrel{\mathcal{D}_i}{\leftarrow} \mathcal{D}_i @ [(d_1, d_2)]$
$\ lab \ \$e \ t \ e_1 \dots e_l \$ \ \ _i$	$= \Gamma, \Delta \stackrel{\mathcal{E}_i}{\leftarrow} \mathcal{E}_i @ [(lab, t, [e_1; \dots; e_l])]$
$\ lab \ \$a \ t \ a_1 \dots a_l \$ \ \ _i$	$= \Gamma \stackrel{\mathcal{A}}{\leftarrow} \mathcal{A} @ [(\overline{\Delta}^{\leq i}, lab, t, [a_1; \dots; a_l])], \Delta$
$\ lab \ \$p \ t \ p_1 \dots p_l \$= \ l_1 \dots l_d \$ \ \ _i$	$= \Gamma \stackrel{\mathcal{T}}{\leftarrow} \mathcal{T} @ [(\overline{\Delta}^{\leq i}, lab, t, [p_1; \dots; p_l], [l_1; \dots; l_d])], \Delta$
$\ \$\{ s_1 \dots s_d \$\} \ _i$	$= \Gamma, [\Delta_0; \dots; \Delta_i; \Delta_{emp}] \mapsto \ s_1 \ _{i+1} \mapsto \dots \mapsto \ s_d \ _{i+1}$

FIGURE 15 – Formalisation de l'étape de normalisation d'un fichier METAMATH

Initialement $\Gamma_{init} = (\emptyset, [], [])$ et $\Delta_{init} = [\Delta_0]$ où $\Delta_0 = \Delta_{emp} \triangleq ([], [], [], [])$. Il faut préserver l'ordre des axiomes et théorèmes car leurs labels peuvent être utilisés dans l'élaboration des preuves des théorèmes, et nous avons vu que la vérification d'une preuve nécessite une pile dont l'ordre des éléments est important car l'inversion de deux hypothèses donnera un résultat tout à fait différent après substitution. Ainsi, nous utilisons rarement la structure mathématique d'ensemble car celle-ci ne préserve pas l'ordre des éléments. Nous optons plutôt pour la structure de liste, et notons $[]$ la liste vide et $@$ l'opération de concaténation de deux listes.

Le résultat de notre traduction est un couple formé d'un contexte global et d'une liste de contextes locaux. La notation $T_1 \mapsto T_2$ indique que la traduction T_2 débute avec le contexte

global et la liste de contextes locaux résultant de la traduction T_1 . La notation $\Delta \xleftarrow{X_i} Y$ indique que nous remplaçons la composante X par Y dans Δ_i , c'est-à-dire dans le i -ème élément de la liste Δ . Enfin, la notation $\overline{\Delta}^{\leq i}$ correspond à la concaténation, composante par composante, des i premiers éléments de la liste Δ . Les variables, ainsi que les f -hypothèses et d -hypothèses associées, ne sont gardées que si les variables apparaissent dans les e -hypothèses ou dans l'énoncé.

Le fichier normalisé correspond au contexte global $\Gamma_{final} \triangleq (\Sigma_{final}, \mathcal{A}_{final}, \mathcal{T}_{final})$ obtenu à la fin de la traduction. Il est presque possible d'obtenir un fichier METAMATH valide en affichant successivement tous les éléments de Σ_{final} , puis de \mathcal{A}_{final} et enfin de \mathcal{T}_{final} . Pour obtenir un fichier complètement valide, il faut assurer que tous les labels sont uniques, et donc procéder à certaines renommages lors de certaines déclarations, mais également dans les preuves. Nous n'avons pas formalisé ce renommage afin de ne pas alourdir notre formalisation.

Les deux encodages proposés par la suite se basent sur cette forme normalisée.

4 Un encodage profond de Metamath en Dedukti

Cette section propose un encodage profond de METAMATH en DEDUKTI, c'est-à-dire un encodage au plus proche de la philosophie actuelle de METAMATH.

4.1 Formalisation de l'encodage profond

La normalisation proposée à la section 3.4 nous indique que METAMATH utilise deux méta-opérateurs afin de *lier* les variables, les différentes hypothèses et l'énoncé entre-eux : une quantification universelle, que nous notons \forall_{MM} , et une implication, que nous nous notons \Rightarrow_{MM} . Ces deux méta-opérateurs manipulent des *tokens*. Nous avons donc besoin de typer chacun des *tokens* :

```
7 symbol token : TYPE;
```

A présent, nous pouvons définir les méta-opérateurs notés \forall_{MM} et \Rightarrow_{MM} :

```
8 symbol  $\forall_{MM}$  : (token  $\rightarrow$  token)  $\rightarrow$  token ;
```

```
9 symbol  $\Rightarrow_{MM}$  : token  $\rightarrow$  token  $\rightarrow$  token ;
```

```
10 notation  $\Rightarrow_{MM}$  infix right 10;
```

En DEDUKTI, il est usuel de définir un symbole permettant de passer d'une formule au type de ses preuves. Ici, ce symbole est noté Prf et est défini à la ligne 11.

```
11 injective symbol Prf : token  $\rightarrow$  TYPE ;
```

Nous pouvons alors définir un *token* ϕ , et une preuve du *token* ϕ , notée ψ , comme ce qui suit :

```
12 symbol  $\phi$  : token ;
```

```
13 symbol  $\psi$  : Prf  $\phi$  ;
```

Il nous est maintenant possible d'expliciter un lien entre les méta-opérateurs de METAMATH, et ceux de DEDUKTI, où $\$ b.[a]$ signifie que a peut apparaître dans b :

```
14 rule Prf ( $\forall_{MM}(\lambda a, \$b.[a])$ )  $\leftrightarrow$   $\Pi a, \text{Prf } \$b.[a]$  ;
```

```
15 rule Prf ( $\$a \Rightarrow_{MM} \$b$ )  $\leftrightarrow$  Prf  $\$a \rightarrow$  Prf  $\$b$  ;
```

Ensuite, nous définissons un opérateur de concaténation de *tokens* afin de pouvoir exprimer, par exemple, le contenu d'une hypothèse ou d'un énoncé.

```
16 symbol ++ : token → token → token ;
17 notation ++ infix right 10;
18 rule ($a ++ $l) ++ $m ↔ $a ++ ($l ++ $m);
```

Afin d'alléger la traduction des preuves, nous déclarons le symbole # comme étant injectif afin de simplifier le processus d'unification effectué par DEDUKTI.

```
19 injective symbol # : token → token → token ;
20 notation # infix right 10;
```

Ce symbole sera intercalé entre un *typecode* (Figure 14) et une liste de *tokens*.

Traduire l'axiomatisation. A l'aide des déclarations précédentes, voyons, sur un exemple, comment traduire une axiomatisation écrite en METAMATH. Nous ne formalisons par cette traduction, par souci de légèreté.

L'axiomatisation disponible à la figure 3 est traduite en DEDUKTI ci-dessous. Les lignes 21 à 25 correspondent à la traduction de la signature, tandis que les lignes 26 à 28 correspondent à l'axiome *wi* et les lignes 29 à 31 à l'axiome du modus ponens.

```
21 constant symbol PL : token ;
22 constant symbol PR : token ;
23 constant symbol wff : token ;
24 constant symbol -> : token ;
25 constant symbol taquet : token ;

26 constant symbol wi :
27   Prf (∀MM(λ φ, ∀MM(λ ψ, ( wff # φ ) ⇒MM( wff # ψ ) ⇒MM
28     ( wff # PL ++ φ ++ -> ++ ψ ++ PR )))) ;
29 constant symbol mp :
30   Prf (∀MM(λ φ, ∀MM(λ ψ, ( wff # φ ) ⇒MM( wff # ψ ) ⇒MM
31     ( taquet # φ ) ⇒MM( taquet # PL ++ φ ++ -> ++ ψ ++ PR ) ⇒MM( taquet # ψ ) )) ) ;
```

Construire un λ-terme. La construction d'un λ-terme suit le mécanisme de vérification des preuves METAMATH. Les labels sont mis au fur et à mesure dans la pile en prenant au préalable un nombre d'éléments dans la pile égal au nombre de *f*-hypothèses et de *e*-hypothèses associées à l'énoncé du label. Nous n'écrivons pas explicitement les valeurs des variables, d'où l'utilisation de jokers, car DEDUKTI est capable de les inférer. Les lignes suivantes correspondent à la traduction de la preuve vue à la section 3.2. Le résultat de cette traduction est très similaire à la liste de labels initialement écrite en METAMATH.

```
33 symbol a2i :
34   Prf (∀MM(λ φ, ∀MM(λ ψ, ∀MM(λ χ, ( wff # φ ) ⇒MM( wff # ψ ) ⇒MM( wff # χ ) ⇒MM
35     ( taquet # PL ++ φ ++ -> ++ PL ++ ψ ++ -> ++ χ ++ PR ++ PR ) ⇒MM
36       ( taquet # PL ++ PL ++ φ ++ -> ++ ψ ++ PR ++ -> ++
37         PL ++ φ ++ -> ++ χ ++ PR ++ PR ) ) ) ) :=
38   λ φ ψ χ wph wps wch hyp,
39   mp - -
40     (wi - - wph (wi - - wps wch))
41     (wi - - (wi - - wph wps) (wi - - wph wch))
42     hyp
43     (a2 - - - wph wps wch);
```

4.2 Premières conclusions liées à cet encodage

Ce premier encodage respecte profondément la philosophie propre de METAMATH. Toutes les constantes jouent le même rôle, et le contenu des énoncés des hypothèses, des axiomes et des théorèmes sont des listes de *tokens*. Certains *tokens* ont du être renommés car certains caractères valides pour METAMATH ne sont pas supportés par DEDUKTI comme `"`, `|`, `.` ou encore `(` et `)`. Finalement, cet encodage préserve la structure de liste initialement donnée par l'utilisateur, mais n'est très pas lisible pour un être humain, comme c'est le cas pour l'énoncé du théorème `a2i`. Ce premier encodage permet cependant de traduire un très grand nombre de preuves de la bibliothèque standard de METAMATH, comme nous le verrons à la section 6.

Afin de gagner en lisibilité, nous proposons un second encodage qui tente de donner un peu plus de sémantique aux *tokens* à l'aide du typage. En DEDUKTI, ainsi que dans de nombreux autres outils formels, les éléments constituant la signature possèdent un nom et un type (et donc une arité). En METAMATH, les éléments constituant la signature possèdent un nom et sont d'arité 0. La section suivante propose un encodage superficiel qui tente de trouver une structure d'arbre appropriée à partir de cette structure de liste. Cela revient à tenter de trouver les types des éléments constituant la signature (et donc leur arité).

5 Vers un encodage superficiel de Metamath en Dedukti

Cette section présente un encodage superficiel de METAMATH en DEDUKTI dont l'objectif principal est de gagner en lisibilité en déterminant le type de chaque symbole de la signature.

5.1 La philosophie de notre encodage superficiel

L'idée générale est de considérer que les éléments de la signature qui sont des *typecodes* de *f*-hypothèses sont en réalité des types. Les axiomes qui ont pour *typecode* un type sont donc des axiomes de bonne formation et sont appelés des *axiomes syntaxiques*. Les autres axiomes sont appelés des *axiomes sémantiques*.

Il nous est donc possible de découper notre signature en trois sous-ensembles : l'ensemble des symboles qui sont des types, comme `wff`, l'ensemble des symboles qui apparaissent en tant que *typecode* mais qui ne sont pas des types, comme `|-`, et l'ensemble des symboles, nommés *opérateurs*, qui n'apparaissent jamais en tant que *typecode*, comme `->`. Ces ensembles forment une partition disjointe de la signature initiale.

A partir de ces trois ensembles, nous donnons le type `TYPE` aux symboles de typage. Par exemple, la ligne suivante correspond à la traduction en DEDUKTI du *token* `wff` :

```
constant symbol wff : TYPE ;
```

De plus, les axiomes syntaxiques nous permettent d'inférer le type des opérateurs. Par exemple, l'axiome `wi` nous permet d'inférer le type du symbole `->`, et ainsi d'obtenir la traduction suivante en DEDUKTI :

```
constant symbol -> : wff -> wff -> wff ;
```

Cependant, il ne nous est pas toujours possible d'inférer toutes les informations voulues, comme par exemple le type du symbole `|-`. Cela fait l'objet de la sous-section suivante.

5.2 Quelques limites

Cette sous-section liste quelques difficultés rencontrées lorsque nous avons voulu traduire des fichiers METAMATH en DEDUKTI avec notre second encodage.

Inférer le type de certains symboles. Certains symboles sont plus difficiles à typer que d'autres. C'est en particulier le cas des symboles qui apparaissent en tant que *typecode* mais qui ne sont pas des types, comme $|-$.

Inférer le parsing des opérateurs. Sous certaines hypothèses, il est possible de se passer des informations de précedence et d'associativité, en connaissant, par exemple, le *token* associé à la parenthèse fermante et le *token* associé à la parenthèse ouvrante. Cependant, il est possible que le nom d'un opérateur soit composé de plusieurs *tokens*, comme par exemple l'opérateur de branchement `if-then-else` qui peut s'écrire avec trois *tokens* : `if _ then _ else _`. Ce cas de figure est très difficile à identifier syntaxiquement.

Détecter le sous-typage. METAMATH permet de définir du sous-typage, comme le montre la déclaration suivante :

```
xnat $f nat x $.
xfloat $a float x $.
```

Cette déclaration indique que si x appartient à l'ensemble des entiers, alors x appartient aussi à l'ensemble des nombres à virgule. Il est possible traduire ce type de définition en DEDUKTI à l'aide d'une injection de l'ensemble de départ vers celui d'arrivée. Cette partie de la traduction n'a pas encore été implémentée à l'heure actuelle.

5.3 Une solution avec des annotations

Pour pallier aux problèmes liés à l'inférence de type et de parsing, il est possible non plus de traduire un fichier METAMATH, mais plutôt de traduire un fichier METAMATH modulo d'autres informations fournies par l'utilisateur, à l'aide d'un fichier JSON par exemple.

À ce stade, nous proposons uniquement à l'utilisateur de préciser un nom de substitution valide pour DEDUKTI (`name_dedukti`), le type du symbole (`type`), la manière dont il faut le parser (`mixfix` et `precedence`) ou si le symbole est à considérer comme une parenthèse ouvrante ou fermante. Un exemple de fichier JSON est disponible à la figure 16 : celui-ci permet de traduire le symbole $|-$ vu précédemment.

```
{ name_metamath : "|-",
  name_dedukti  : "taquet",
  type         : "wff → TYPE",
  mixfix       : "prefix",
  precedence   : 40 }
↓
constant symbol taquet : wff → TYPE ;
```

FIGURE 16 – Exemple de traduction d'un symbole à partir d'un fichier JSON

Ainsi, à l'aide du type précisé dans le fichier JSON, DEDUKTI est capable de vérifier le typage de la traduction de l'axiome du modus ponens : `constant symbol mp : $\Pi (\phi : \text{wff}), \Pi (\psi : \text{wff}), |-\phi \rightarrow |-(\phi \rightarrow \psi) \rightarrow |-\psi ;$`

De plus, la traduction d'une preuve METAMATH en λ -terme est illustrée aux figures 17 à 24. Cette traduction suit le même mécanisme que celui que METAMATH utilise pour vérifier une preuve. La subtilité est que c'est le contenu des f -hypothèses, des e -hypothèses et des axiomes syntaxiques qui est mis sur la pile, tandis que pour les axiomes sémantiques, nous utilisons le label qui lui est associé, en utilisant les arguments qui sont sur la pile. Ainsi, il est possible d'obtenir le λ -terme $\lambda\varphi, \lambda\psi, \lambda\chi, \lambda\text{hyp}, \text{mp } \alpha \beta \text{ hyp (a2 } \varphi \psi \chi)$ avec $\alpha \triangleq (\varphi \rightarrow (\psi \rightarrow \chi))$ et $\beta \triangleq ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$ à partir de la preuve de la figure 4.

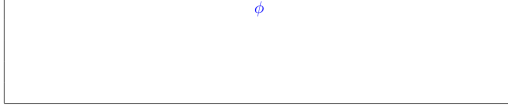


FIGURE 17 – État de la pile en ①

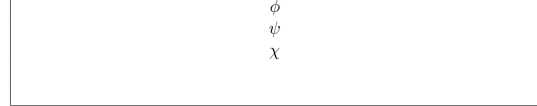


FIGURE 18 – État de la pile en ②

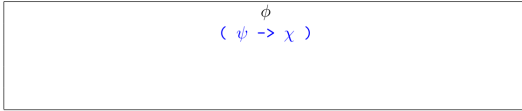


FIGURE 19 – État de la pile en ③

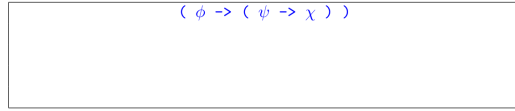


FIGURE 20 – État de la pile en ④

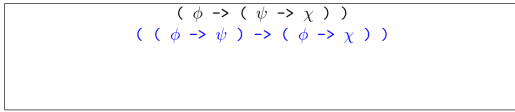


FIGURE 21 – État de la pile en ⑤

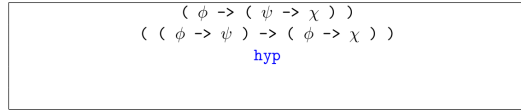


FIGURE 22 – État de la pile en ⑥

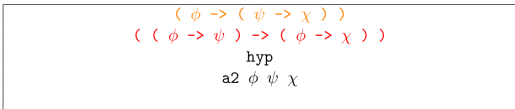


FIGURE 23 – État de la pile en ⑦

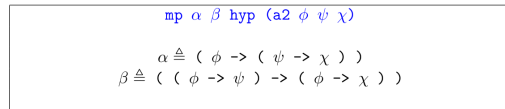


FIGURE 24 – État de la pile en ⑧

L'ajout d'un fichier JSON lors du processus de traduction semble être la seule manière de contourner les différents problèmes que nous rencontrons lorsque nous cherchons à obtenir un encodage superficiel de METAMATH en DEDUKTI. Cependant, cette solution n'est pas complètement satisfaisante puisque la vérification d'un fichier METAMATH ne dépend plus uniquement du fichier lui-même, mais également du fichier JSON fourni par l'utilisateur. Un fichier METAMATH vérifié par DEDUKTI est donc correct modulo les informations fournies à l'aide du fichier JSON donné par l'utilisateur lors de la traduction. Une question ouverte résiste donc : quelle(s) propriété(s) les informations données dans le fichier JSON doivent-elles vérifier pour que la traduction soit pertinente et correcte ?

6 Traduction de la bibliothèque standard de Metamath

Cette section étudie la traduction de la bibliothèque standard de METAMATH avec nos deux encodages, afin de valider les encodages proposés. Après avoir présenté les résultats obtenus, nous proposons une extension commune à l'encodage superficiel et l'encodage profond.

6.1 Résultats de la traduction pour chaque encodage

La bibliothèque standard de METAMATH est composée de neuf fichiers disponibles sur GitHub : <https://github.com/metamath/set.mm>. Les résultats obtenus avec les implémentations actuelles des traductions utilisant nos deux encodages sont disponibles à la figure 25.

	<i>demo0.mm</i>	<i>miu.mm</i>	<i>peano.mm</i>	<i>big-unifier.mm</i>	<i>hol.mm</i>	<i>ql.mm</i>	<i>nf.mm</i>	<i>iset.mm</i>	<i>set.mm</i>
Taille du fichier	1.4K	4.6K	28K	39K	231K	1.8M	8.3M	11M	273M
Nombre de :									
- déclaration(s)	19	27	116	29	2 768	936	24 967	30 813	196 976
- axiome(s)	7	10	48	4	77	71	363	467	2711
- preuve(s)	1	1	0	2	1 138	138	5 966	8 990	38 766
Pourcentage traduit avec l'encodage :									
- profond	100	0	100	100	95,87	100	79,07	80,29	65,26*
- profond étendu	100	0	100	100	100	100	100	100	100*
- superficiel	100	0	0	100	0	0+	0+	0+	2,52+

FIGURE 25 – Pourcentage de traduction de la bibliothèque standard de METAMATH pour les différents encodages proposés

Les pourcentages obtenus indiquent la proportion traduite pour chaque fichier et pour chaque encodage. Les fichiers obtenus ont pu être vérifiés par DEDUKTI, sauf pour les pourcentages ayant une étoile (*). Le fichier *set.mm* étant très volumineux, nous n'avons pas encore réussi à vérifier le typage de la totalité des deux fichiers de traduction obtenus, alors que METAMATH n'a besoin que de quelques secondes pour vérifier le fichier *set.mm*. Cette différence s'explique en partie car notre traducteur ne fonctionne qu'avec des fichiers METAMATH décompressés. Par exemple, la taille du fichier *set.mm* compressé est de 40.8M, et il faut plusieurs heures pour décompresser ce fichier. De plus, nous n'avons pas pu traduire le fichier *miu.mm* car celui-ci autorise la substitution d'une variable par *rien*, fonctionnalité considérée *unsafe* par METAMATH lui-même.

Dans le cas de l'encodage profond présenté à la section 4, la quasi-totalité des axiomatisations et des preuves a pu être traduite. Les preuves non traduites utilisent des variables libres, ce qui est possible car METAMATH fait l'hypothèse que tous les types qu'il manipule sont non vides. Les fichiers *hol.mm*, *nf.mm*, *iset.mm* et *set.mm* possèdent respectivement 47, 1 249, 1 772 et 13 466 preuves nécessitant la traduction de ce cas de figure. Nous avons donc implémenté un encodage profond étendu afin de pouvoir traduire ces preuves : cette extension est suffisante pour traduire

toutes les preuves restantes, comme le montre la figure 25. Ce cas de figure est illustré dans la sous-section suivante, ainsi qu'une explication plus détaillée de l'extension codée pour traduire la totalité des preuves de la bibliothèque standard de METAMATH avec notre encodage profond. Cette extension est également nécessaire pour notre encodage superficiel.

Dans le cas de l'encodage superficiel présenté à la section 5, nous nous sommes principalement concentrés sur la traduction de la logique propositionnelle à la Hilbert (extrait du fichier `set.mm`), dont nous avons réussi à traduire en DEDUKTI environ 1 000 preuves. Le fichier `peano.mm` n'a pas pu être traduit car nous n'avons pas encore implémenté la génération d'injections à partir des axiomes de sous-typage. Le fichier `hol.mm` n'a également pas pu être traduit car il nécessite la traduction d'un opérateur d'application n'ayant pas de *token*, comme le montre la figure 26.

```
$c term ( ) $.
$( Term variables $)
$v F T $.
tf $f term F $.
tt $f term T $.

$( A combination (function application). $)
kc $a term ( F T ) $.
```

FIGURE 26 – Extrait du fichier `hol.mm`

Le pourcentage de traduction des fichiers `q1.mm`, `nf.mm`, `iset.mm` et `set.mm` est certainement plus élevé que celui annoncé car nous n'avons pas cherché à écrire la totalité des fichiers JSON nécessaires. En effet, cette tâche est longue et fastidieuse car il faut réussir à inférer manuellement un parsing et un typage correcte pour tous les *tokens* dont cela est nécessaire. Il nous semble plus judicieux d'essayer de trouver une méthode automatique pour trouver ces informations, vérifier la conformité du fichier traduit en DEDUKTI, puis essayer une nouvelle possibilité de parsing ou de typage en cas d'échec.

6.2 Une extension commune pour chaque encodage

Certaines preuves utilisent des variables qui ne sont pas liées dans l'énoncé du théorème. Cela est possible car, implicitement, METAMATH fait l'hypothèse que tout type est habité : il existe donc au moins une variable pour chaque type. Cependant, ces preuves utilisent des variables libres, qui n'existent plus lors de l'étape de normalisation. Considérons la formalisation en METAMATH suivante afin d'illustrer nos propos :

```
$c ( ) -> <-> wff |- $.
$v ph ps $.
wph $f wff ph $.
wps $f wff ps $.
wi $a wff ( ph -> ps ) $.

id $p |- ( ph -> ph ) $= ... $.

${ mpbir.min $e |- ps $.
  mpbir.maj $e |- ( ph <-> ps ) $.
  mpbir $p |- ph $= ... $. }
```

```
$c A. setvar class = T. $.
${ $v x $.
  vx.wal $f setvar x $.
  wal $a wff A. x ph $. }
${ $v x $.
  vx.cv $f setvar x $.
  cv $a class x $. }
${ $v A B $.
  cA.wceq $f class A $.
  cB.wceq $f class B $.
  wceq $a wff A = B $. }
wtru $a wff T. $.
```

```

 $\{$    $v$   $x$   $.$ 
  vx.tru  $f$  setvar  $x$   $.$ 
  df-tru  $a$  |- ( T. <-> ( A.  $x$   $x$  =  $x$  -> A.  $x$   $x$  =  $x$  ) )  $.$ 
  tru  $p$  |- T.  $=$ 
    wtru vx.tru cv vx.tru cv wceq vx.tru wal vx.tru cv vx.tru cv
    wceq vx.tru wal wi vx.tru cv vx.tru cv
    wceq vx.tru wal id vx.tru df-tru mpbir  $.$   $\}$ 

```

La preuve du théorème `tru` utilise le label `vx.tru` introduisant la variable `x` dans la pile. Lors de la traduction vers DEDUKTI, la variable `x` sera libre car elle n'apparaît pas dans l'énoncé du théorème `tru`. En effet, la forme normalisée de la section précédente est :

```

 $\{$    $v$   $x$   $.$ 
  vx.tru  $f$  setvar  $x$   $.$ 
  df-tru  $a$  |- ( T. <-> ( A.  $x$   $x$  =  $x$  -> A.  $x$   $x$  =  $x$  ) )  $.$   $\}$ 
tru  $p$  |- T.  $=$ 
  wtru vx.tru cv vx.tru cv wceq vx.tru wal vx.tru cv vx.tru cv
  wceq vx.tru wal wi vx.tru cv vx.tru cv
  wceq vx.tru wal id vx.tru df-tru mpbir  $.$ 

```

Ainsi, la preuve du théorème `tru` utilise le label `vx.tru` qui n'est plus dans la portée du théorème `tru`. Nous devons étendre les encodages proposés afin de générer les habitants nécessaires pour chaque type. Dans le cadre de notre exemple, nous devons donc également générer le symbole suivant en DEDUKTI : `constant symbol vxDottru : Prf (setvar # x) ;`.

Cette extension est suffisante pour traduire les preuves restantes de la bibliothèque standard de METAMATH, dans le cas de notre encodage profond.

7 Conclusion

Nous venons de présenter deux encodages de METAMATH vers DEDUKTI, l'un profond, l'autre superficiel.

L'encodage profond de METAMATH en DEDUKTI nous apprend que la logique de METAMATH est faible puisqu'elle n'est constituée que d'un opérateur de quantification et d'un opérateur d'implication. Cette logique semble suffisamment faible pour utiliser les travaux effectués par F. Thiré [14] afin de traduire le résultat obtenu vers COQ, ISABELLE ou encore PVS.

L'encodage superficiel, plus lisible pour un être humain, permet de mettre en lumière les fonctionnalités usuelles modélisées dans une formalisation écrite en METAMATH, comme le sous-typage. Cependant, cet encodage pose de nombreuses questions sur comment utiliser DEDUKTI pour traduire le résultat obtenu vers COQ, ISABELLE ou encore PVS. De plus, il semble difficile de pouvoir complètement automatiser la traduction de fichiers METAMATH dans cet encodage.

La traduction des *d*-hypothèses n'a jamais été présentée, car celles-ci ne sont utilisées que dans un mécanisme de vérification interne à METAMATH dont aucune information n'apparaît dans la preuve écrite en METAMATH. Il semble cependant possible d'étendre notre encodage profond pour également encoder cette vérification.

Cet article constitue donc une étape importante pour permettre l'interopérabilité des preuves écrites en METAMATH à l'aide de DEDUKTI, en utilisant l'approche basée sur la théorie \mathcal{U} [5].

De plus, nous envisageons d'étendre le nombre de fichiers METAMATH traduits, comme par exemple grâce à la formalisation en METAMATH de la MATCHING LOGIC [10].

Enfin, M. Carneiro a proposé des modèles pour les différentes parties de la bibliothèque standard de METAMATH [7]. Nous envisageons une comparaison plus approfondie de ces travaux avec les nôtres. M. Carneiro a également proposé METAMATH-Zero [8], un outil formel inspiré de METAMATH mais qui n'utilise pas de pile de preuves, et qui considère ses expressions comme des arbres et non comme des listes. Nous envisageons de mieux comprendre les liens entre METAMATH et METAMATH-Zero, afin d'évaluer la difficulté à récupérer des preuves de METAMATH-Zero dans DEDUKTI.

Remerciements : Nous remercions Catherine DUBOIS, Chantal KELLER et Andrei PASKEVICH pour les remarques, commentaires et conseils qu'ils nous ont apportés tout au long de ce travail. Nous remercions également les reviewers anonymes pour leurs remarques et suggestions.

Références

- [1] Formalizing 100 Theorems - Freek Wiedijk. <https://www.cs.ru.nl/~freek/100/>.
- [2] Translation of proofs of Euclid's book I into HOL Light, Lean, Matita, OpenTheory and PVS, note = https://github.com/karnaj/sttfa_geoqc_euclid.
- [3] Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing theories in the $\lambda\Pi$ -calculus modulo theory and in the Dedukti system. In *TYPES : Types for Proofs and Programs*, Novi Sad, Serbia, May 2016.
- [4] Ali Assaf and Raphaël Cauderlier. Mixing HOL and Coq in Dedukti (Extended Abstract). In Cezary Kaliszyk and Andrei Paskevich, editors, *Proceedings Fourth Workshop on Proof eXchange for Theorem Proving, PxTP 2015, Berlin, Germany, August 2-3, 2015*, volume 186 of *EPTCS*, pages 89–96, 2015.
- [5] Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, and François Thiré. Some axioms for mathematics. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPICs*, pages 20 :1–20 :19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.
- [6] Mario Carneiro. Conversion of HOL Light proofs into Metamath, 2014.
- [7] Mario Carneiro. Models for Metamath, 2016.
- [8] Mario Carneiro. Metamath Zero : The Cartesian Theorem Prover, 2019.
- [9] Raphaël Cauderlier and Catherine Dubois. FoCaLiZe and Dedukti to the Rescue for Proof Interoperability. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving*, pages 131–147, Cham, 2017. Springer International Publishing.
- [10] Xiaohong Chen, Zhengyao Lin, Minh-Thai Trinh, and Grigore Roşu. Towards a Trustworthy Semantics-Based Language Framework via Proof Generation. In *Proceedings of the 33rd International Conference on Computer-Aided Verification*. ACM, July 2021.
- [11] D. Cousineau and Gilles Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *TLCA*, 2007.
- [12] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics*, pages 243–320, 1990.
- [13] Norman D. Megill. *Metamath : A Computer Language for Mathematical Proofs*. Lulu Press, Morrisville, North Carolina, 2019. <http://us.metamath.org/downloads/metamath.pdf>.
- [14] François Thiré. Sharing a Library between Proof Assistants : Reaching out to the HOL Family. *Electronic Proceedings in Theoretical Computer Science*, 274 :57–71, jul 2018.
- [15] François Thiré. *Interoperability between proof systems using the logical framework Dedukti. (Interopérabilité entre systèmes de preuve en utilisant le cadre logique Dedukti)*. PhD thesis, École normale supérieure Paris-Saclay, Cachan, France, 2020.

Backtracking reference stores

Camille Noûs¹ and Gabriel Scherer²

¹ Laboratoire Cogitamus

² INRIA

Abstract

François Pottier’s `unionFind` library is parameterized over an underlying store of mutable references, and provides the usual references, transactional reference stores (for rolling back some changes in case of higher-level errors), and persistent reference stores. We extend this library with a new implementation of backtracking reference stores, to get a Union-Find implementation that efficiently supports arbitrary backtracking and also subsumes the transactional interface.

Our backtracking reference stores are not specific to `unionFind`, they can be used to build arbitrary backtracking data structures. The natural implementation, using a journal to record all writes, provides amortized-constant-time operations with a space overhead linear in the number of store updates. A refined implementation reduces the memory overhead to be linear in the number of store cells updated, and gives performance that match non-backtracking references in practice.

1 Introduction

Our own use-case for the present work comes from implementing a type-checker for an explicitly-typed (no inference required) extension of System F with Guarded Algebraic Datatypes (GADTs). With GADTs, datatype constructors may witness equalities between types. For example, matching on the constructor `Int` at type α `rtty` may reveal that the (universally quantified) type variable α is in fact equal to `int`, that is, introduce the equality $(\alpha = \text{int})$ in the typing context. Then under this context we have to type-check a sub-expression, the right-hand-side of the pattern matching clause, which in particular involves checking many type equalities $(\Gamma \vdash \tau_1 = \tau_2)$ modulo the equations in Γ . For example, the types $(\alpha \rightarrow \text{int})$ and $(\text{int} \rightarrow \alpha)$ are distinct in general, but they are equal under the assumption $(\alpha = \text{int})$.

Union-Find is the perfect data structure to efficiently check equalities in presence of equality assumptions. Our type-checker carries a global Union-Find graph, we create nodes for bound type variables, equality assumptions add unification edges in the graph, and equality checking can be implemented efficiently – even in presence of cyclic graphs / equi-recursive types. But adding a *local* equality assumption, as required by GADTs, requires a form of *backtracking*: we add unification edges before checking the sub-expressions of the GADT pattern clause, but these edges should not remain present when type-checking the rest of the term outside this clause.

A simple solution to this problem is to use a *persistent* Union-Find; there is a generic way to implement a persistent version of a mutable data-structure, which is to replace all mutable references by indices into a persistent map. When we extend a persistent Union-Find graph G with a new equality, we get a new Union-Find graph G' ; we can type-check the clause right-hand-side using G' and then go back to G to check the rest of the type derivation.

```

let rec typeof env = function (* ... *)
| Use (t, u) ->
  match typeof env t with
  | TyEq (ty1, ty2) ->
    let env = introduce_equalities env ty1 ty2 in
    typeof env u
  | ty -> raise (TypeError (NotAnEquality ty))

```

However, this solution imposes a logarithmic overhead to all accesses and updates, which adds a noticeable constant factor on reasonably-large programs. This is especially frustrating given that GADTs are a relatively rare language feature; the whole type-checker pays for a feature that may not even occur in the program being type-checked.

Another solution is to use a mutable, constant-time-access Union-Find graph, but perform a copy of the graph when entering a GADT equality assumption. This has cost linear in the number of type variables in context, which is unpleasant, but the cost is localized to GADT use sites. This is a reasonable solution that, let's be frank, would probably be fine in practice for our use-case.

```

match typeof env t with
| TyEq (ty1, ty2) ->
  let env = { env with store = StoreVector.copy store } in
  introduce_equalities env ty1 ty2;
  typeof env u

```

We want to do better by implementing a Union-Find that supports both constant-time access/update and (amortized) constant-time backtracking.

```

match typeof env t with
| TyEq (ty1, ty2) ->
  BtRef.branch env.store;
  let finally () = BtRef.terminate_rollback env.store in
  Fun.protect ~finally @@ fun () ->
    introduce_equalities env ty1 ty2;
    typeof env u

```

1.1 A trusted ally: a modular Union-Find library

François Pottier's `unionFind` library provides an efficient implementation of Union-Find parameterized over a notion of *mutable store*, which allows to choose the implementation of mutable references used in the Union-Find graph. At the time of writing, it provides four implementations of this store interface:

StoreRef uses OCaml references directly. This gives the standard UnionFind behavior.

StoreMap uses indices into a persistent map. This gives a mutable API that operates on a persistent graph under the hood, providing a constant-time operation to copy (share) the whole graph. Algorithmically this corresponds exactly to the persistent approach we discussed above.

StoreVector uses indices into a dynamic array. This gives constant-time access and update, but tracks the set of nodes of the graph (and refers them through indices, a portable indirection) which allows copying the whole graph in linear time. This corresponds exactly to the copying approach we discussed above.

StoreTransactionalRef implements a *transactional* reference store, which allow starting a global transaction with the ability to either commit all changes that happened during the transaction or to roll them back. This is important for type-checking algorithms that provide partial operations (typically unification) on top of the Union-Find graph. An operation may perform several unifications in sequence, but it should leave the graph unchanged in case the operation fails – typically to print error messages to the user without showing an inconsistent, in-progress state. Transactions cannot be nested.

1.2 Our contribution

We implement a *backtracking reference store* that allows to backtrack to previous versions of the store in amortized constant time. A store contains many references that are versioned together. There are explicit operations to *branch* the store into a new version, or return a store to its parent version. In particular, if we roll back to a previous version, this rolls back the value of all references in the store that were modified since the last branch. We can instantiate François Pottier’s `unionFind` implementation with our new store, solving our quest for Union-Find with backtracking.

Implementation The implementation of our structure is fairly simple, Intuitively the store maintains a *journal*, or *undo log*, of all reference updates, which allows to roll back to previous versions. In fact we have two implementations:

- In our first, simplest implementation, all updates are logged in the journal. Read/write operations are constant-time, backtracking to the parent version is amortized constant-time. But the journal adds a space overhead linear in the number of updates and branches performed. (This is just fine for Union-Find, where the number of updates per node is less than logarithmic. But our store may have other users without such guarantee.)
- We then propose a *record-eliding* optimization that uses timestamps to avoid recording a given reference twice for the same version of the structure. The space overhead is linear in the number of references updated in each version, rather than the number of writes.

Interface The building blocks of our implementations subsume two different flavors of interfaces that we can express on top of our work:

- We can provide a semi-persistent interface in the sense of [Conchon and Filliâtre \(2008\)](#).
- We provide a transactional interface, subsuming François Pottier’s transactional references, but in addition we support arbitrary nesting of transactions.

Terminology: *backtracking, semi-persistent, transactional* [Conchon and Filliâtre \(2008\)](#) proposes a specific API design for *semi-persistent* structures where backtracking to a previous version is implicit. It happens on-demand when we start again to work on an older version, the first access to this older version implicitly invalidates/backtrack any more recent version. This gives a very declarative programming style, close in spirit to programming with persistent data structures, hence the name *semi-persistent*.

Our core API is different, more imperative. Terminating a version is explicit; this works better with the transactional interface that gives two different ways to terminate a version (`commit` or `abort`), making the implicit choice awkward. But one can easily build the semi-persistent API on top of our existing interface – we do it in Section 4.2. We call our API a *backtracking* API, although it is a bit more expressive than typical backtracking APIs that

only allow to rollback or abort changes, while we can also commit them – as with transactional interfaces.

A more precise name may be *transactional*, but we already use it in the context of François Pottier’s `StoreTransactionalRef` stores that do not support nested transactions. We could call them *one-transactional*, or call ours *nested-transactional*, but we stick with the more common *backtracking*.

Generality We emphasize that while our own use-case is a backtracking Union-Find, our backtracking (and transactional) references are of general interest and could be used to implement basically any backtracking data-structure.

For example, you can rebuild backtracking arrays by simply using arrays of references in a shared backtracking store – at the constant-factor cost of an extra indirection per array element. This would be silly for backtracking or semi-persistent arrays where specialized implementations are already available, or for structures that are naturally built by composing existing backtracking libraries, in the same way [Conchon and Filliâtre \(2008\)](#) builds semi-persistent hashtables out of semi-persistent dynamic arrays and semi-persistent stacks. But you may be interested in backtracking doubly-linked lists, backtracking quad-trees, backtracking skiplists, etc.

1.3 Early challengers: (semi-)persistent dynamic arrays

As we were considering this journey of implementing our own backtracking references, the wizard Jean-Christophe Filliâtre sent us the following remark to test our resolve.

It is possible to implement a Union-Find graph backed by a dynamic array, so you get a semi-persistent Union-Find by using an existing library of semi-persistent dynamic arrays.

This solution has in fact already been presented in [Conchon and Filliâtre \(2007\)](#), which proposes a persistent Union-Find implementation backed by a dynamic array, but tweaks it slightly to be only semi-persistent in its Section 2.3.3.

Our justification for writing our own code is that array-backed Union-Find graphs – including the use of François Pottier’s `unionFind` instantiated with its `StoreVector` module we mentioned above – do not play nicely with garbage collection. All nodes of the graph remain alive as long as the graph lives. In contrast, direct representations where the Union-Find graph is realized by pointers in memory preserve the liveness of individual nodes, which can be collected as soon as they are not used in the program anymore.

This makes a difference in our use-case of using a Union-Find graph on the fly for type-checking: when we traverse our typing derivation, we generate many Union-Find nodes on the fly, each time we check an equality between types. But they are extremely short-lived, we do not use them after we have checked those equalities. We expect those dead fragments of the Union-Find graph to be collected promptly, while they leak when using an array-backed implementation.

Note that transactional or backtracking references also extend the lifetime of values in certain circumstances: to allow rolling back to a previous version we necessarily keep those previous versions alive in memory. But this lifetime-extension ends when the corresponding transaction or version is terminated. In our Union-Find use-case, this means that the garbage-collection behavior of Union-Find nodes is unchanged by these journaling mechanisms.

Finally, (semi-)persistent arrays, in existing OCaml implementations, do not provide the equivalent of our record-eliding optimization – their memory overhead is linear in the total

number of writes, instead of the number of distinct positions written. It may be possible to implement it; we suspect that it is non-obvious for semi-persistent arrays and difficult for persistent arrays. If we want record elision, we need to write new code anyway.

Remark: persistent vs. semi-persistent Semi-persistent data-structures were introduced in [Conchon and Filliâtre \(2008\)](#) as an optimization of fully-persistent data structures for scenarios where backtracking is required, but not more complex reuse scenarios. We mentioned earlier a naive strategy for persistence adding a logarithmic overhead on all access. But note that persistent data-structures can also be fast thanks to *rerooting* as presented in [Conchon and Filliâtre \(2007\)](#), almost as fast as semi-persistent data structures.

Reusing the existing implementation of a Union-Find graph backed by a persistent array would certainly be fine performance-wise; as with any array-backed approach it prevents garbage collection, and we believe that implementing the record-eliding optimization would be more difficult. Our record-eliding optimization relies on the fact that valid versions of a semi-persistent data structure form a linear structure, so they can be denoted by integers, with a fast check of whether a version is an ancestor of another. On the other hand, fully persistent data structures have a tree of versions, so our approach does not work.

2 Specification

In this section we present the set of primitive operations supported by our backtracking store of references, just their specification.

Note: we did not start by writing down a specification, we started by writing an implementation with a bug on nested transactions. We had to work out a clear specification to avoid the bug in the future.

2.1 Specifying backtracking data structures

Let us first recall the vocabulary to specify semi-persistent data structures proposed in [Conchon and Filliâtre \(2008\)](#).

In a semi-persistent implementation, there are several *versions* of the same mutable structures, in our case a store, that is, a set of references. Each version of the data-structure has a corresponding state, independent of the other versions.

There is a *current* (most recent) version.

We can *branch* a new version from the current version: it is a child of the current version and becomes the new current version. This new version starts in exactly the same state as its parent version – the previous current version.

We can *terminate* the current version, if it has a parent version. This parent version becomes the new current version. (Note: [Conchon and Filliâtre \(2008\)](#) does not discuss termination explicitly, so we are deviating slightly here.)

Remark that versions form a linear path, not a tree. We would have a tree if we could branch from any version, not just the current one, in particular we could have a version with several children. This would correspond to a fully persistent implementation.

There is a *root version* that is the current version when the store is just created, and has no parent version.

2.2 Backtracking store of references

Our backtracking data-structure is a *store of references*. We follow François Pottier’s design for stores of references (modulo simplifications for presentation). A store is a set of references and each references belongs to exactly one store.

Usual store operations

```

type store
val new_store : unit -> store

type 'a rref
val make: store -> 'a -> 'a rref
val get:  store -> 'a rref -> 'a
val set:  store -> 'a rref -> 'a -> unit

```

The *state* of a (usual) store of references is just a mapping from each reference to a value of its content type; `get` and `set` operations modify the state as expected.

With our backtracking stores, a store has several versions, and the logical state of each version is exactly this mapping from each reference in the store to a value.

`make` adds a new reference to the store. Note that this operation is independent from the current version: the state of all versions of the store now has a mapping for this reference.

`get` and `set` operate on the current version of the store, as expected.

Creating new versions

```

val branch : store -> unit
val terminate_nodiff : store -> unit

```

`branch` has the usual specification for a semi-persistent data structure: it creates a new version, child of the current version, which becomes the new current version, and initially has the same state as the previous current version.

We provide a `terminate_nodiff` function to terminate the current version; its parent version becomes the new parent version. For reasons that will become apparent in the next paragraph, this function assumes that the state of the current version is equal to the state of its parent. (It fails if the current version is the root version.)

Transactions

```

val commit : store -> unit
val rollback : store -> unit

```

Those functions provide the necessary primitives to implement François Pottier’s transactional interface on top of our backtracking store – and a bit more. They require the current version to have a parent version, and fail if the current version is the root version.

`commit` changes the state of the parent version to become identical to the state of the current version.

`rollback` changes the state of the current version to become identical to the state of the parent version.

Those operations do not terminate the current version, but one can call `terminate_nodiff` after them – it expects the current and parent version to be in the same state, which they both enforce.

Note: it would be natural to provide a higher-level interface that offers only `terminate_commit` and `terminate_rollback`, hiding the harder-to-use `terminate_nodiff` function.

Performance We expect a *fast path* behavior of `get` and `set` when no version has been branched (see below). In that case, `set` should write the content of the reference without any extra book-keeping, so that the performance is very close to standard references. Backtracking must be a *low-cost* abstraction, that only adds noticeable overhead when it is actually

used. (This fast path behavior is also present in the `StoreTransactionalRef` implementation of François Pottier, so it is important that we support it to subsume that implementation.)

3 Implementation(s)

Our references are implemented as record with a mutable field. (Our record-eliding implementation will get an extra metadata field, discussed in the relevant section.)

```
type 'a rref = { mutable current: 'a; }
```

We maintain the invariant that the *global state* of a store, the content of the `current` field of each reference, is equal to the logical state of the current version of the store. This guarantees in particular that `get` can be implemented with a single read of a mutable field.

The general idea of our implementations is that the store maintains a journal of writes to its references, so that those writes can be rolled back when backtracking.

To undo a write/set, we need the reference that was written and the value of the reference before the write:

```
type undo_action =
  | Set : 'a rref * 'a -> undo_action
```

Note that if we supported other types of backtracking mutable objects in the store, for example arrays, we could add more undo actions (for an array write we would store the array, the index and the old value at this index).

We went through three different implementations, from the less to the more sophisticated. They differ by the data-structure used to store undo actions, and whether we record an undo action for all writes or only some of them.

3.1 Stack of stacks

In our first implementation, the difference between a version and its parent version is stored as a stack of undo actions. Our versioned store is just a mutable list of such differences from the current version to the root version.

```
type diff = undo_action Stack.t
type store = diff list ref
```

When the store is empty, the current version is the root version. The root version has no parent, so it does not store a diff. When the store is of the form `d::ds`, `d` is a diff from the current version to its parent, and `ds` a list of diffs from the parent version to the root version. In other words, a store keeps the whole journal of all writes, but segmented in separate stacks, one for each parent-child relationship between versions.

```
let new_store () = ref []
let make (_s : store) (v : 'a) : 'a rref = { current = v; }
let get (_s : store) (x : 'a rref) : 'a = x.current
```

Set The specification of `set` says that it operates on the state of the current version of the store. In our representation, we maintain the invariant that the state of the current version is equal to the global state of the references, so `set` must update this global state. We also maintain the invariant that the store diff list relates the state of the current version to the state of the root version; the current version changes, so `set` must also update the *current diff*, the diff between the current version and its parent.

```

let set (s : store) (x : 'a rref) (v : 'a) : unit =
begin (* Update the diff list. *)
  match !s with
  | [] -> ()
  | current_diff :: _rest ->
    (* Add the write to the diff of the current version *)
    Stack.push (Set (x, x.current)) current_diff;
end;
(* Update the global state. *)
x.current <- v

```

In the case where the current version is the root version, our representation does not maintain an undo log for the root version so we do not record the write. The root version has no parent so `commit` or `rollback` cannot be called, so we cannot observe the presence of an undo log.

Branch and terminate

```

let branch s =
  s := Stack.create () :: !s
let terminate_nodiff s =
  match !s with
  | [] -> invalid_arg "terminate_nodiff_version: root version cannot be terminated"
  | diff :: rest ->
    assert (Stack.is_empty diff);
    s := rest

```

Note: checking that `diff` is empty is slightly stronger than our specification, which allows non-empty diffs of updates that cancel each other. Our implementation enforces the more intentional specification that `terminate_nodiff` is always called immediately after `branch`, `commit` or `rollback`.

Rollback and commit `rollback` mutates the current state to become equal to the parent state, while `commit` mutates the parent state to become equal to the current state.

```

let rollback s =
  match !s with
  | [] ->
    invalid_arg "rollback: the root version cannot be rolled back"
  | current_diff :: _ ->
    (* Current state:
       { current version  $\xrightarrow{\text{current diff}}$  parent version
         in state A                      in state B }
    *)
    while not (Stack.is_empty current_diff) do
      match Stack.pop current_diff with
      | Set (x, old) ->
        x.current <- old
    done;
    (* Final state:
       { current version  $\xrightarrow{\emptyset}$  parent version
         in state B                      in state B }
    *)

```

`commit` relies on an auxiliary function on stacks, `move_stack s1 ~into:s2`. It moves all elements of `s1` into `s2`, in the same order as they were in `s1`, which is now empty. In other words, if before the call the list of elements of `s1` and `s2` were `l1` and `l2` respectively, then after the call it is `[]` and `l1 @ l2`.

```
let commit s =
  match !s with
  | [] ->
    invalid_arg "rollback: the root version cannot be committed"
  | diff :: [] ->
    (* Current state:
       { current version  $\xrightarrow{\text{diff}}$  root version
         in state A           in state B }
    *)
    Stack.clear diff;
    (* Final state:
       { current version  $\xrightarrow{\emptyset}$  root version
         in state A           in state A }
    *)
  | diff :: parent_diff :: _ ->
    (* Current state:
       { current version  $\xrightarrow{\text{diff}}$  parent version  $\xrightarrow{\text{parent diff}}$  parent parent version
         in state A           in state B           in state C }
    *)
    move_stack diff ~into:parent_diff;
    (* Final state:
       { current version  $\xrightarrow{\emptyset}$  parent version  $\xrightarrow{\text{diff} + \text{parent diff}}$  parent parent version
         in state A           in state A           in state C }
    *)
```

Complexity Ideally we want each operation to take constant time, possibly amortized. Amortized reasoning works well with `rollback`. It traverses the undo log of the current version to perform each undo action, removing it from the `diff`. Each undo action is rolled back at most once, and we can consider that its rollback cost was paid in advance by the corresponding `set`.

This reasoning fails for `commit` unfortunately. `move_stack` traverses the current undo log, and this traversal cost can be amortized, but the undo actions are not dropped after traversal, they remain in the `diff` of another version. One would need `set` to pre-pay for each version, which is not a constant cost. Amortized reasoning fails, and in fact we can observe quadratic complexity in the worst case: consider a sequence of N `branch` calls, then N writes, then N (`commit; terminate_nodiff`) sequences, this runs in $O(N^2)$. (This is not a concern if you only use backtracking and never the transactional interface, as in this case you always `rollback` and never `commit`.)

It would be possible to change the `diff` data-structure from stacks to something that provides constant-time concatenation, such as doubly-linked lists. Instead we move to our second implementation, which uses simpler data structures and can thus be expected to have lower constant factors – at the cost of being less abstract.

3.2 Stack and indices

In our second iteration, we use a single stack to store the undo actions of all versions. We remember the boundaries of each version in a separate list.

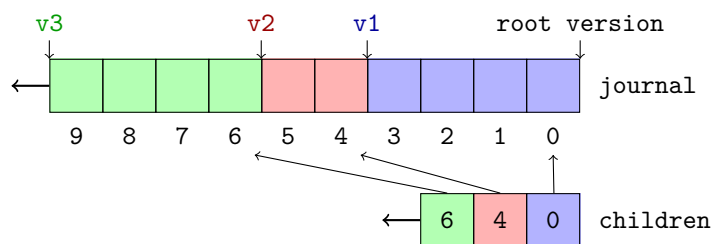
```

type store = { journal: undo_action Stack.t;
               mutable children: int list; }

```

For each child (non-root) version, the list `children` stores the index of this version in the `journal`, that is, the starting position of the diff of this version. The children versions are stored in the list from the most recent (the current version) to the oldest (the immediate child of the root version). Note that the root version keeps no diff / undo log, so its immediate child always starts at position 0.

For example, a store with three children versions in addition to the root version may have the following representation – we draw the stacks and lists as growing from the right to the left.



Easy changes Most functions are trivially adapted to this new representation. In the case of `set`, we have to check the list `s.children` to tell whether we are at the root version (which stores no diff) or not, and only push to `journal` in the non-root case:

```

let set (s : store) (x : 'a rref) (v : 'a) : unit =
  begin match s.children with
  | [] -> ()
  | _ :: _ ->
      Stack.push (Set (x, x.current)) s.journal;
  end;
  x.current <- v

```

`branch` uses the current length of the `journal` as the starting position of the new version:

```

let branch s =
  s.children <- Stack.length s.journal :: s.children

```

Finally, `rollback` performs some length computation to determine how many undo actions to roll back from the `journal`:

```

let rollback s =
  match s.children with
  | [] ->
      invalid_arg "rollback: the root version cannot be rolled back"
  | curr_start :: _ ->
      let current_version_length = Stack.length s.journal - curr_start in
      for _i = 1 to current_version_length do
        match Stack.pop s.journal with
        | Set (x, old) ->
            x.current <- old
      done;
      assert (Stack.length s.journal = curr_start);

```


Commit This function motivated the change of representation. In the case where the parent version is not the root version, `commit` does not touch the journal at all – the undo actions remain in the same position in the stack, we only need to adjust the version boundaries. We move the start of the current version to the end of the stack, so that the parent version adopts the undo actions that were previously in the current version.

```

let commit s =
  match s.children with
  | [] ->
    invalid_arg "rollback: the root version cannot be committed"
  | _curr_start :: parent_start :: rest ->
    (* Current state:
       {
         end of stack      pos. curr_start
         current version ←current diff parent version ←parent diff parent parent version
         state: A           state: B           state: C
       }
       We move the boundary of our current version to the current
       stack length. This moves the content of our diff to our
       parent version. *)
    s.children <- Stack.length s.journal :: parent_start :: rest;
    (* Final state:
       {
         end of stack      pos. Stack.length      current diff      pos. parent_start
         current version = 0 parent version ←+ parent diff parent parent version
         state: A           state: A               + parent diff   state: C
       }
       *)
    | curr_start :: [] ->
      (* Current state:
         {
           end of stack      pos. curr_start = 0
           current version ←current diff root version
           state: A           state: B
         }
         The current version is the immediate child of the root version,
         which has no undo log. We know that the current version starts in
         position [0], and can discard the changes in our undo log. *)
      assert (curr_start = 0);
      Stack.clear s.journal;
      (* Final state:
         {
           end of stack      pos. curr_start = 0
           current version = 0 root version
           state: A           state: A
         }
         *)

```

In the case where the root version is the parent version, a linear cost may remain — depending on the complexity of `Stack.clear`. But there is no complexity issue there, this is amortized constant time. Indeed, we only pay this cost once per undo action, and then the actions are removed from the journal. We can consider that `set` paid for this cost in advance.

3.3 Record elision

Our last iteration is a small modification of our previous version that avoids recording several writes to the same cell in a given version. Recording only the first write suffices to rollback the changes if necessary.

During `set` we could walk back the undo log of the current version, and exit early if we

find another write to the same reference. This would be very slow. Instead we store, in each reference, information on when the last write to this reference was recorded – using positions inside the journal as a natural notion of *timestamp*.

```

type 'a rref = {
  mutable current: 'a;
  (** The value of this reference in the current version. *)
  mutable last_record: int;
  (** [last_record] is the position of the most recent
      record of this reference recorded in the journal,
      or -1 this reference has no record. *)
}

let make (_s : store) (v : 'a) : 'a rref =
  { current = v; last_record = -1; }

```

(Our actual code uses an abstract type `Pos_option.t` instead of `int` to hide the `-1` encoding, but auxiliary abstractions would make the presentation heavier here.)

We need to save this `last_record` field in our undo actions to be able to restore it correctly during rollback:

```

type undo_action =
| Set : {
  ref: 'a rref;
  (** the reference that was written *)
  previous: 'a;
  (** the value of the reference before the write *)
  previous_record: int;
  (** the [last_record] value at write time *)
} -> undo_action

```

Set We can skip recording the write if the last record belongs to the current version. We update the `last_record` field when we extend the journal.

```

let set (s : store) (x : 'a rref) (v : 'a) : unit =
  begin match s.children with
  | [] -> ()
  | current_version_start :: _ ->
    if x.last_record >= current_version_start then ()
    else begin
      let new_record = Stack.length s.journal in
      Stack.push (Set {
        ref = x;
        previous = x.current;
        previous_record = x.last_record;
      }) s.journal;
      x.last_record <- new_record;
    end
  end;
  x.current <- v

```

Note that while the space-overhead complexity reduction may feel nebulous for many ap-

plications, this optimization also makes a qualitative difference in performance. Indeed, the (polymorphic) writes dominate the cost of `set`; with the previous implementations, each `set` would perform two such writes, one to the reference field and the other by pushing into the journal. With the new implementation, in the common case we perform a single write. Outside the root-version fast path, our previous `set` implementations was at least twice slower than with standard references, while the overhead of the new version becomes neglectible for write-heavy workflows. Said otherwise, this implementation is low-cost even in presence of infrequent branching.

Commit There is a subtlety in the case where our parent version is the root version. The previous implementation simply clears the journal in this case, but doing so invalidates the `last_record` fields of the references mentioned in this journal, pointing to journal entries that no longer exist. This turns into a bug later where `set` does record a reference, falsely believing that it is already recorded. To avoid creating invalid `last_record` positions, we must clear this field for the references mentioned in the journal. We highlight the only lines of code that change since the last version:

```
let commit s =
  match s.children with
  | [] -> invalid_arg "rollback: the root version cannot be committed"
  | _curr_start :: parent_start :: rest ->
    s.children <- Stack.length s.journal :: parent_start :: rest;
  | curr_start :: [] ->
    assert (curr_start = 0);
    s.journal |> Stack.iter (function
      | Set {ref; previous = _; previous_record = _} ->
        ref.last_record <- -1;
    );
    Stack.clear s.journal;
```

Note that, when the parent version is not the root version, it may be the case that a single reference was recorded in the current diff and also in the parent diff. In this case it ends up mentioned twice in the final diff of the parent version. We do not preserve the invariant that a reference is recorded at most once in each version. Enforcing this invariant saves no work, and the obvious way to do it is to perform a linear traversal during `commit`, which would create again a quadratic worst case.

4 Interface(s)

```
module BtRef = StoreBacktrackingRef
let rec typeof env = function (*...*) | Use (t, u) ->
  match typeof env t with
  | TyEq (ty1, ty2) ->
    BtRef.branch env.store;
    let finally () =
      BtRef.rollback env.store;
      BtRef.terminate_nodiff env.store; in
    Fun.protect ~finally @@ fun () ->
      introduce_equalities env ty1 ty2;
      typeof env u
```

The primitives defined previously are enough to use our Union-Find with backtracking. In this section we build two higher-level interfaces, to show that our primitives are expressive enough. One expresses the transactional API of François Pottier, the other expresses the original semi-persistent API of [Conchon and Filliâtre \(2008\)](#).

4.1 Transactional API

François Pottier exposes the following function for his transactional reference implementation, `StoreTransactionalRef`:

*(**[tentatively s f] runs the function [f] within a new transaction on the store [s]. If [f] raises an exception, then the transaction is aborted, and all updates performed by [f] on references in the store [s] are rolled back. If [f] terminates normally, then the updates performed by [f] are committed.*

Two transactions on a single store cannot be nested.

*A cell that is created during a transaction still exists after the transaction, even if the transaction is rolled back. In that case, its content should be considered undefined. *)*

```
val tentatively: 'a store -> (unit -> 'b) -> 'b
```

We can implement this API for our backtracking references, with two improvements:

1. Our transactions can be nested at will. (This single change to the `StoreTransactionalRef` specification suffices to express arbitrary backtracking.)
2. Our specification naturally gives a meaning to the content of a reference that was created in a version since rolled back. In our specification, a reference is defined in all versions, and its initial value (in all versions) is the parameter provided to `make`.

As we remarked earlier, `StoreTransactionalRef` also provides the important property of being low-cost in the sense that read and writes outside a `tentatively` call are essentially as fast as raw references. We also preserve this property, thanks to the absence (in our implementations) of an undo log for the root version.

```
let tentatively (s : store) (f : unit -> 'b) : 'b =
  branch s;
  match f () with
  | v ->
    commit s;
    terminate_nodiff s;
    v
  | exception e ->
    let b = Printexc.get_raw_backtrace() in
    rollback s;
    terminate_nodiff s;
    Printexc.raise_with_backtrace e b
```

4.2 Semi-persistent API

The API proposed in [Conchon and Filliâtre \(2008\)](#) has a more declarative, less imperative flavor than ours.

```

module SemiPersistent : sig
  type versioned_store

  val new_store : unit -> versioned_store
  val branch : versioned_store -> versioned_store

  val make : versioned_store -> 'a -> 'a rref
  val get : versioned_store -> 'a rref -> 'a
  val set : versioned_store -> 'a rref -> 'a -> unit
end

```

This API manipulates *versioned stores*, whereas our imperative API acts on a global store. With this versioned API, `branch` returns a new store version. The functions `branch`, `get` or `set` do not need to be called on the current version, but they backtrack to the version they were given, invalidating/aborting any child version. There is no explicit way to terminate a version.

This gives nice, declarative code for our store-backtracking function.

```

module SPRef = StoreBacktrackingRef.SemiPersistent
let rec typeof env = function (*...*) | Use (t, u) ->
  match typeof env t with
  | TyEq (ty1, ty2) ->
    let env = { env with store = SPRef.branch env.store } in
    introduce_equalities env ty1 ty2;
    typeof env u

```

On the other hand, we are not sure how to gracefully integrate the transactional features, the difference between `commit` and `rollback`. This is related to the fact that termination of versions is implicit. We could add a `commit` function, but it might break user expectations by having the user of a child version modify its parent version.

(It is also somewhat awkward that `make` takes a versioned store as input but adds the reference to all versions.)

In fact, we can define a generic functor that takes a data-structure with our imperative API, and implements the declarative API on top.

```

module type Backtracking = sig
  type t
  val branch : t -> unit
  val terminate : t -> unit
end

```

(With our primitives, `terminate` is defined by calling `rollback` then `terminate_nodiff`.)

```

module type SemiPersistent = sig
  type t
  type version

  (* A root version with the given initial state;
     it is initially the current version for this state. *)
  val new_root : t -> version

```

```

(* Branches a new version from a given version
   (any previous transitive child of the given version is invalidated);
   the new version becomes the new current version. *))
val branch : version -> version

(* The given version becomes the current version
   (any transitive child is invalidated);
   the corresponding global state is returned. *))
val access : version -> t
end

module Make (D : Backtracking)
  : SemiPersistent with type t = D.t
= struct ... end

```

The type `version` provided by this functor keeps the global state of the datastructure, along with a mutable `status` field that tracks whether it is currently the current version, is the parent of another version, or has been invalidated by backtracking.

```

type t = D.t
type version = {
  global : D.t;
  mutable status : status;
}
and status =
| Current
| Parent_of of version
| Invalid

let new_root global =
  { global; status = Current; }

```

Branching modifies the status of the branched version to track its child. We first implement `branch_current`, which assumes that it gets the current version.

```

let branch_current version =
  assert (version.status = Current);
  D.branch version.global;
  let child = { global = version.global; status = Current } in
  version.status <- Parent_of child;
  child

```

We then implement a `backtrack` primitive that backtracks the global state to make a given version the current version.

```

let backtrack version =
  (* [collect] accumulates the transitive children of a version. *))
  let rec collect children version =
    match version.status with
    | Current -> children
    | Invalid -> invalid_arg "backtrack: this version is invalid"
    | Parent_of child -> collect (child :: children) child
  in

```

```
(* terminate and invalidate all children versions. *)
collect [] version |> List.iter (fun child ->
  child.status <- Invalid;
  D.terminate version.global;
);
version.status <- Current
```

Finally, `branch` and `access` first backtrack to the given version, then branch a new version or access the underlying global structure.

```
let branch version =
  backtrack version;
  branch_current version

let access version =
  backtrack version;
  version.global
```

5 Related Work

StoreTransactionalRef Within the library ecosystem we are familiar with, our work is fairly similar to François Pottier’s `StoreTransactionalRef` implementation. We provide more features (nested transactions) with no additional costs. We haven’t discussed this with François Pottier yet, but we would expect our proposal to replace his `StoreTransactionalRef` module completely.

In automated solvers Automated solvers, for example SMT solvers, rely heavily on efficient backtracking. When we looked in the `opam` package repository for an implementation of backtrack-able `Union-Find` or backtrack-able references, we could not find anything; but since we worked on this paper every author of a solver we met tells us with “l’air penaud” that they have a version of this, deep in the middle of their own code, that they have never shown to anyone. Boo!

A kind anonymous reviewer also pointed us to the `CVC5` overview paper: [Barbosa, Barrett, Brain, Kremer, Lachnitt, Mann, Mohamed, Mohamed, Niemetz, Nötzli, Ozdemir, Preiner, Reynolds, Sheng, Tinelli, and Zohar \(2022\)](#). In Section 2.4, “context-dependent data structures”, they mention a general idea of backtrackable data structures indexed by a *context* with an imperative push/pop interface – as common in SAT/SMT solvers. `CVC5` supports various data-structures as first-class members of those contexts, which also correspond to our stores. It is not clear to non-experts as we are whether one should try to continue with just backtrackable stores of references as a simple building block, and define elaborate data structures on top of it (`Union-Find`, `hashtables`, etc.), or whether we really get noticeable efficiency benefits by adding more data structures as first-class concepts in the backtrackable store, creating opportunities for specialized implementations or more compact representations.

Hidden OCaml implementations After submitting this article we were pointed to, in particular, (unpublished) implementations of backtracking stores of references in the `FaCiLe` constraint-solving library and in the `Colibri2` constraint solver, which more generally implements the context interface of `CVC5` for various data structures. Note that SMT implementations typically support only backtracking, that is an `abort` operation, but not the `commit` operation of François Pottier’s transactional references. `FaCiLe` does support a `commit` operation which is described as a `cut` from logic programming.

Specialized implementations Finally, in the late 80s there was apparently a lot of work on specialized implementations of Union-Find with various forms of built-in backtracking support, see for example [Apostolico, Italiano, Gambosi, and Talamo \(1994\)](#). We don't know of released implementation of these algorithms. They may be marginally faster than building on top of stores of references, but cannot be reused for other data structures.

6 Benchmarks

We wrote some benchmarks to confirm experimentally our overall claim: the implementation we present in this work (the latest iteration, with the record-elision optimization) provides references with backtracking support at low cost. They are suitable to build backtracking data structures.

You can find our detailed performance results at <https://gitlab.inria.fr/gscherer/unionfind/-/blob/jfla-benchmarks/bench/README.Store.JFLA.md> A subset of these results are mentioned in [Appendix A](#).

In our performance tests, the `Facile` implementation is around 15% slower than ours, while `Colibri2` is around 60% slower than our code. (These overheads are small and become negligible for most workflows that are not dominated by reference performance; for comparison, using a persistent Map is 13x slower than our implementation, and our attempt to use `CCHashTrie` with transient updates fared even worse.)

We believe that the `Colibri2` slowdown comes from an "on-demand" design where the backtracking work is done the next time a reference is accessed, instead of backtracking all references eagerly at abort/rollback time. This on-demand logic may save effort in some workloads, but it adds an extra `rewind` function call in the hot paths of `get` and `set`. In the common case the `rewind` function returns immediately because no backtracking is going on, or because it has already been performed, but a function call that returns immediately is enough in a very fast path for a noticeable performance difference.

References

- Alberto Apostolico, Giuseppe F. Italiano, Giorgio Gambosi, and Maurizio Talamo. The set union problem with unlimited backtracking. *SIAM Journal on Computing*, 23(1):50–70, 1994.
- Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. `cvc5`: A versatile and industrial-strength smt solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99524-9.
- Sylvain Conchon and Jean-Christophe Filliâtre. [A Persistent Union-Find Data Structure](#). In *ACM SIGPLAN Workshop on ML*, pages 37–45, Freiburg, Germany, October 2007. ACM Press.
- Sylvain Conchon and Jean-Christophe Filliâtre. [Semi-Persistent Data Structures](#). In *17th European Symposium on Programming (ESOP'08)*, April 2008.

A Boring Benchmarks

We benchmarked several implementations of stores of references, some backtracking, some supporting only non-nested transactions, some supporting no transaction at all:

Ref is just using raw references. No transactions or backtracking. transactional or backtracking workflows.

Map is François Pottier’s `StoreMap` implementation. It supports constant-time backtracking, but the logarithmic overhead on each access is prohibitive.

Vector is François Pottier’s implementation storing all references in a dynamic array. The array can be copied, which gives us a way to implement backtracking.

BacktrackingRefv1 was our first implementation of backtracking reference store in Section 3.1. It contains a complexity bug for the `commit` operation.

BacktrackingRefv2 was our second implementation of backtracking reference store in Section 3.2, fixing the complexity issue for `commit`.

BacktrackingRef is our final, third implementation of backtracking reference store in Section 3.3. It contains a record-elision optimization that elides redundant writes to the journal at the same version, improving space usage and also time constant factors (only one polymorphic write in the common case).

TransactionalRef is François Pottier’s implementation of “transactional” references. It contains its own sort of record-eliding optimization, and should behave similarly well in write-heavy workflows.

Facile is an implementation of our interface on top of FaCiLe’s backtracking references: https://github.com/Emmanuel-PLF/facile/blob/master/lib/fcl_stak.mli FaCiLe also implements a record-elision optimization.

Colibri2 is an implementation of our interface on top of Colibri2’s references in backtracking contexts. <https://git.frama-c.com/pub/colibrics/-/blob/bobot/abs/colibri2/stdlib/context.mli> Colibri2 also implements a record-elision optimization.

(For full scripts and even more results, see <https://gitlab.inria.fr/gscherer/unionfind/-/blob/jfla-benchmarks/bench/README.Store.JFLA.md>)

For most applications of references, reads dominate writes. The benchmarks that follow perform 2^{15} writes and 2^{20} reads (so 32x more reads) to an array of 2^{10} references. (This workload is then iterated a certain number of times.)

A.1 Raw benchmarks

These benchmarks only test read and write, no transactional or backtracking workflows. (No transaction is ever started or no backtracking snapshot taken.)

impl.	time	memory
Ref	1.37s	2.4Mio
TransactionalRef	1.64s	2.4Mio
BacktrackingRefv1	1.55s	2.4Mio
BacktrackingRefv2	1.52s	2.5Mio
BacktrackingRef	1.53s	2.4Mio
Facile	1.74s	2.4Mio
Colibri2	2.45s	2.4Mio
Vector	1.62s	2.4Mio
Map	20.00s	4.5Mio

We see that there is a very small performance difference between simple references and the transactional or backtracking version – mostly due to code size / inlining factors, simple references have shorter code but they do the same thing in the hot path.

Facile and Colibri2 are slower than our implementations. Facile is 15% slower, Colibri2 is 60% slower. (See our Related Work section 5 for our understanding of the performance disadvantage of Colibri2.)

Persistent maps have much worse performance than all other due to their logarithmic access overhead.

A.2 Transactional benchmarks

This benchmark repeatedly performs the raw alloc-read-write workflow under one transaction, which is either committed or aborted.

impl.	time	memory
TransactionalRef	1.14s	2.4Mio
BacktrackingRefv1	1.42s	12.2Mio
BacktrackingRefv2	1.52s	12.3Mio
BacktrackingRef	1.13s	2.5Mio
Facile	1.20s	2.5Mio
Colibri2	1.63s	2.5Mio
Vector	1.66s	4.1Mio
Map	12.48s	4.5Mio

TransactionalRef, BacktrackingRef, Facile and Colibri2 perform well under this benchmark, with similar relative differences between them.

On the other hand, previous versions v1 and v2 of BacktrackingRef pay a 2x time overhead due to running under a transaction. Their memory usage is also noticeably higher, due to the absence of record-elision logic: they add each write to the log. (All further benchmarks show the same memory advantage to our final version of BacktrackingRef, corresponding to the better space-complexity claims.)

B Backtracking benchmarks

impl.	time	memory
BacktrackingRefv1	18.96s	458.0Mio
BacktrackingRefv2	1.63s	231.1Mio
BacktrackingRef	0.89s	2.5Mio
Facile	1.25s	2.5Mio
Colibri2	1.77s	2.6Mio
Vector	1.87s	4.1Mio
Map	17.12s	4.6Mio

We can observe the complexity bug in `commit` for `BacktrackingRefv1`.

In our previous backtracking benchmark, the nesting depth is 100, with 2^{20} reads and 2^{15} writes at each level. We could observe better results with `Map` by having a longer nesting depth, and fewer reads and writes.

If we use a nesting depth of 30000, and only 2^5 reads and 2^5 writes at each level, we observe the following results:

impl.	time	memory
BacktrackingRefv1	0.51s	160.0Mio
BacktrackingRefv2	0.50s	161.5Mio
BacktrackingRef	0.09s	32.2Mio
Facile	0.10s	40.9Mio
Colibri2	0.13s	33.4Mio
Vector	0.78s	542.6Mio
Map	0.80s	42.0Mio

`Map` looks much better, but the record-eliding versions (`BacktrackingRef`, `Facile`, `Colibri2`) perform even better. `Vector` consumes a lot of memory due to frequent copies; copies corresponding to previous versions of the structure cannot be reclaimed by garbage collection, as we may still restore them on backtracking.

We are not sure whether this high-nesting-low-work scenario is relevant to any real-world use case. Automated theorem provers have very deep nestings of backtracking; but even SMT solvers tend to do a lot of propagation work in-between two backtracking points.

Analyse statique de valeurs par interprétation abstraite de programmes fonctionnels manipulant des types algébriques récurrents

Milla Valnet¹, Raphaël Monat² et Antoine Miné³

¹ École Normale Supérieure, Université PSL, Paris
`milla.valnet@ens.psl.eu`

² Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRISTAL, F-59000 Lille
`raphael.monat@inria.fr`

³ Sorbonne Université, CNRS, LIP6, F-75005 Paris
`antoine.mine@lip6.fr`

Résumé

Afin de prévenir les erreurs de programmation, des analyseurs statiques ont été développés pour de nombreux langages ; cependant, aucun analyseur mature ne cible l’analyse de valeurs pour un langage fonctionnel à la ML. Des outils de vérification pour ces langages existent, tels les systèmes de types classiques ou les méthodes déductives, mais le raisonnement automatique sur des programmes numériques a jusqu’alors été peu exploré. Cet article décrit un analyseur statique de valeurs par interprétation abstraite pour un langage fonctionnel typé du premier ordre, approche sûre et automatique pour garantir l’absence d’erreurs à l’exécution. En se basant sur des domaines abstraits relationnels et en réalisant des résumés des champs récurrents des types algébriques, cette approche permet d’analyser des fonctions récurrentes manipulant des types algébriques récurrents et d’inférer dans un domaine abstrait leur relation entrée-sortie. Une implémentation est en cours sur la plateforme d’analyse multilangage MOPSA et analyse avec succès de courts programmes de quelques lignes. Ce travail ouvre ainsi la voie vers une analyse de valeurs précise et relationnelle basée sur l’interprétation abstraite pour les langages fonctionnels d’ordre supérieur à la ML.

1 Introduction

Même lorsqu’il est fortement et statiquement typé, un programme fonctionnel est toujours susceptible de signaler des erreurs lors de l’exécution — accès à un tableau en dehors de ses bornes, assertions fausses, divisions par zero, échec de filtrage, etc. — ou d’avoir un comportement indésirable tels les dépassements d’entier. Notre objectif est de développer une analyse capable de détecter statiquement de telles erreurs.

L’obtention de garanties sur les langages fonctionnels s’est jusqu’ici très largement reposée sur les systèmes de types, certains trop imprécis pour prouver l’absence d’erreurs, comme l’inférence à la ML, d’autres très précis comme les types de raffinements [31, 32] mais utilisant des solveurs lourds comme boîtes noires. D’autres techniques se basant sur des prouveurs de théorèmes [13, 30] supportent les fonctions récurrentes sur les types récurrents, mais demandent à l’utilisateur de spécifier les pré- et post-

conditions des fonctions récursives (équivalent des invariants de boucles) pour assister le prouveur. Le choix d'une analyse statique comporte ainsi l'avantage d'une méthode automatique, sans annotation.

Cependant, les analyses statiques de programmes fonctionnels se sont beaucoup concentrées sur l'analyse de flot de contrôle (CFA), [24, 25] un type d'analyse visant à déterminer à quel point du programme une fonction peut être appelée, ou encore sur des analyses de *strictness* ou de terminaison [10]. Ces méthodes ne retiennent aucune information concernant les valeurs des variables numériques et ne permettent donc pas de garantir l'absence de certaines erreurs à l'exécution. Plus récemment, Montagu et al. [23] se sont intéressés aux relations entre les champs des types algébriques et Bautista et al. [2, 3] à celles entre leurs valeurs, mais aucun ne supporte les types récurifs.

Enfin, comparés aux langages impératifs et orientés objets, sur lesquels sont développés la plupart des analyseurs abstraits [4, 6, 12, 29] à ce jour, les langages fonctionnels introduisent des défis nouveaux, rassemblés sur l'exemple suivant :

```

type list = Cons of int * list | Nil
let rec mult2 l =
  match l with
  | Cons(h, q) -> Cons(2*h, mult2 q)
  | Nil -> Nil
in
let hd x = match x with Cons(h,q) -> h | Nil -> assert false in
let x = Cons(0, Cons(1, Cons(2, Nil))) in
assert(hd (mult2 x) <= 4)

```

La fonction `mult2` multiplie par deux tous les éléments de la liste — définie par un type utilisateur — passée en argument, tandis que la fonction `hd` renvoie son premier élément (sa tête). L'objectif de notre analyse est d'être en mesure de prouver l'assertion et donc l'absence d'erreurs à l'exécution. Cela implique d'être capable d'analyser les types algébriques récurifs, les fonctions récursives et le filtrage par motifs. Ce travail fournit donc les contributions suivantes :

- Un domaine abstrait relationnel pour les objets algébriques récurifs, modulaire en le domaine numérique choisi, effectuant des résumés par champ.
- Une analyse de valeurs par interprétation abstraite pour un langage fonctionnel monomorphe du premier ordre capable d'analyser des fonctions récursives manipulant des objets récurifs et d'inférer leur relation entrée-sortie.
- Une implémentation en OCaml dans la plateforme MOPSA, en cours de développement et analysant avec succès de courts programmes écrits en OCaml.

Plan. La section 2 décrit le langage fonctionnel sur lequel nous travaillerons. Les méthodes d'analyse sont présentées et formalisées en section 3. La section 4 s'attarde

sur leur implémentation dans la plateforme MOPSA, tandis que la section 5 détaille l'état de l'art. La section 6 présente les extensions possibles et conclut.

2 Un langage fonctionnel

Dans cette partie, nous décrivons un langage fonctionnel jouet que nous chercherons à analyser dans les parties suivantes.

Alors que les langages impératifs manipulent majoritairement des données numériques, des structures ou encore des pointeurs, les langages fonctionnels utilisent quant à eux largement la notion de *type algébrique* (ou *Algebraic Data Types*, ADTs). Il s'agit d'un type formé par combinaison (sommées ou produits) d'autres types. Nous considérerons ici des types algébriques de la forme suivante :

$$\begin{aligned} \mathcal{T}_0 &::= \text{int} \mid (C_1 \text{ of } \tau_{1,1} * \dots * \tau_{1,m_1} \mid \dots \mid C_n \text{ of } \tau_{n,1} * \dots * \tau_{n,m_n}) \quad \text{avec } \tau_{i,j} \in \mathcal{T}_0 \\ \mathcal{T} &::= d \in \mathcal{T}_0 \mid d_1 \rightarrow \dots \rightarrow d_n \rightarrow d_{n+1}, \quad \text{avec } \forall i, d_i \in \mathcal{T}_0 \end{aligned}$$

Ainsi, un type est dans \mathcal{T}_0 s'il est le type *int* (type des entiers), ou s'il est un type algébriquement construit à partir de lui-même et des types de \mathcal{T}_0 , c'est-à-dire sous la forme d'une somme de constructions $C_i \text{ of } \tau_{i,1} * \dots * \tau_{i,m_i}$, où les C_i sont appelés constructeurs de type. Un type t sous cette forme représente ainsi les objets s'écrivant sous la forme $C_i(e_{i,1}, \dots, e_{i,m_i})$, où $e_{i,j}$ est une expression de type $\tau_{i,j}$.

Ce type peut être vu comme la somme des types $C_i \text{ of } \tau_{i,1} * \dots * \tau_{i,m_i}$, eux-mêmes réalisant le produit de types $\tau_{i,1} * \dots * \tau_{i,m_i}$. Cette définition des types algébriques, proche de celle de OCaml, n'est pas une définition générale. En effet, une construction unique réunit ici deux constructeurs différents, la somme et le produit, de sorte à obtenir une syntaxe et une sémantique plus compacte. Par souci de simplification, nous ne considérerons pas les types mutuellement récursifs dans ce formalisme, mais tous les raisonnements menés s'y étendraient sans difficulté. De plus, notre analyse se limitera à un langage monomorphe du premier ordre, aussi les types $\tau_{i,j}$ ne peuvent pas être des fonctions, et les types algébriques polymorphes ne sont pas supportés.

L'ensemble \mathcal{T} des types possibles est donc l'ensemble des types $d \in \mathcal{T}_0$, ainsi que le type des fonctions de d_1, \dots, d_n dans d_{n+1} avec $d_i \in \mathcal{T}_0$. On considérera désormais un programme préfacé par une liste de déclarations de types sous la forme **type** $t = C_1 \text{ of } \tau_{1,1} * \dots * \tau_{1,m_1} \mid \dots \mid C_n \text{ of } \tau_{n,1} * \dots * \tau_{n,m_n}$. Un type peut être algébriquement construit uniquement à base des types précédemment déclarés, du type *int* et de lui-même. On note \mathbb{C} l'ensemble (fini) des constructeurs de type C_i utilisés dans le programme. En posant $e_1, \dots, e_n \in \mathcal{E}, p_1, \dots, p_n \in \Pi, C \in \mathbb{C}$, nous nous munirons ici du langage « à la ML » dont la syntaxe est détaillée en figure 1.

Ici, \mathbb{V} représente l'ensemble des variables, le symbole \oplus les opérateurs du langage ($+$, $-$, $*$, $/$, $=$, etc.). $fv(e)$ correspond aux variables libres de l'expression e . Π , l'ensemble des motifs (*patterns*), contient ainsi le joker $_$, les variables, les entiers, les clauses **when**, et les constructeurs de types dont les paramètres sont des motifs aux variables libres disjointes. \perp_X symbolise la non-terminaison des expressions à valeur dans X . Les

$$\begin{aligned}
\mathcal{E} ::= & x \in \mathbb{V} \\
& \begin{array}{l|l}
n \in \mathbb{Z} & e_1 \oplus e_2 \\
e_1 e_2 \dots e_n & \text{fun } x_1 \dots x_n \rightarrow e \\
\text{let } x = e_1 \text{ in } e_2 & \text{let rec } x = e_1 \text{ in } e_2 \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 & \text{assert } e \\
C(e_1, \dots, e_n) & (\text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m)
\end{array} \\
\Pi ::= & _ \mid x \in \mathbb{V} \mid n \in \mathbb{Z} \mid p_1 \text{ when } e_1 \mid C(p_1, \dots, p_n), \text{ où } \forall i \neq j, fv(p_i) \cap fv(p_j) = \emptyset \\
\mathcal{V}_0 ::= & \mathbb{Z}^\perp \cup \mathbb{C}(\mathcal{V}_0)^\perp \cup \{\omega\}^\perp, \text{ avec } \begin{cases} X^\perp = X \cup \{\perp_X\} \\ \mathbb{C}(X) = \{ C(x_1, \dots, x_n) \mid C \in \mathbb{C}, x_i \in X \} \end{cases} \\
\mathcal{V} ::= & \mathcal{V}_0 \cup \Lambda^\perp \text{ avec } \Lambda = [\mathcal{V}_0^n \rightarrow \mathcal{V}_0] \\
\Sigma = & \mathbb{V} \rightarrow \mathcal{V}
\end{aligned}$$

Figure 1 – Syntaxe du langage fonctionnel.

valeurs de premier ordre du langage \mathcal{V}_0 sont les entiers, les constructeurs de types dont les arguments sont des valeurs de premier ordre et les erreurs ω auxquelles on adjoint \perp . L'ensemble des valeurs \mathcal{V} se constitue ainsi de l'ensemble des valeurs du premier ordre ainsi que des fonctions continues des valeurs d'ordre 1 dans les valeurs d'ordre 1 (\perp en cas de non-terminaison). Enfin, Σ représente l'environnement, associant une valeur à chaque variable. La sémantique concrète du langage définie en Annexe A, figure 4.

Nous avons mentionné les types et leur déclaration au début de cette sous-section. Dans la suite, nous travaillerons sous l'hypothèse que nos programmes sont bien typés. L'inférence de type ne sera pas détaillée ici, mais fonctionne de la manière habituelle. Au cours de l'analyse, les informations sur les déclarations de types comme sur les types des variables seront utilisées pour sélectionner le domaine d'abstraction adéquat, en particulier pour les objets algébriques. De plus, cette inférence permet de détecter de manière statique certaines situations dans laquelle une expression s'évalue en ω . Elle échoue cependant à détecter cette erreur lorsque celle-ci est déclenchée par un échec de filtrage (i.e. lorsque l'élément ne correspond à aucun motif), d'assertions ou de division par zéro. Ainsi, c'est sur de telles propriétés, portant sur la valeur des variables, que notre analyse se focalisera.

On notera que le type `int` correspond ici aux entiers mathématiques; cependant, travailler sur des entiers machines dans le cadre de l'analyse d'OCaml ne représente pas de difficulté supplémentaire majeure. Enfin, le langage défini n'encode pas toutes les caractéristiques des langages fonctionnels. L'ordre supérieur et le polymorphisme ne sont pas supportés, et ce langage est fonctionnel *pur*, ce qui signifie qu'il ne permet aucun effet de bord — ceci exclut en particulier l'utilisation des références et des tableaux. Ce langage contient ainsi les caractéristiques fonctionnelles sur lesquelles nous travaillerons, les fonctions et types algébriques récursifs. Cela permettra en particulier d'analyser les

fonctions récurives manipulant des structures de données récurives.

3 Formalisation des domaines

3.1 L'interprétation abstraite

Le domaine des valeurs et les opérateurs définissant la sémantique concrète du langage de la section 2 sont trop complexes à manipuler : les propriétés intéressantes sont incalculables. On utilise la théorie de l'interprétation abstraite [9], qui consiste à remplacer un domaine concret C par un domaine abstrait plus simple, A , aux éléments représentables en mémoire, munis respectivement d'ordre partiels \leq et \sqsubseteq correspondant au niveau d'information contenu dans les éléments.

Une fonction de *concrétisation* $\gamma : A \rightarrow C$ donne alors le sens d'un élément abstrait, en expliquant quel élément concret il peut représenter, et doit être compatible avec l'ordre d'information — croissante, donc. Pour $a \in A, c \in C, \gamma(a)$ est alors la concrétisation de a dans C , et si $\gamma(a) \geq c$, alors a est une approximation correcte de c .

Par exemple, dans un programme manipulant des valeurs numériques, chaque variable peut être représentée par l'ensemble de ses valeurs possibles, dans $\mathcal{P}(\mathbb{Z})$, ordonné par l'inclusion. Cependant, cet ensemble étant de taille non bornée, son stockage et sa manipulation sont coûteux. En choisissant comme ensemble abstrait des intervalles de valeurs [8], on sur-approxime $\mathcal{P}(\mathbb{Z})$ en un ensemble plus léger à manipuler. Par exemple, on peut abstraire $\{1, 3, 18, 37\}$ par $[1, 37]$.

Il s'agit ensuite d'être capable d'abstraire les opérateurs du concret, telle l'addition des entiers, en des opérateurs sur les éléments abstraits. Une *abstraction sûre* de $f : C \rightarrow C$ est alors une fonction $f^\# : A \rightarrow A$ telle que $\forall a \in A, f(\gamma(a)) \leq \gamma(f^\#(a))$.

Enfin, pour interpréter des boucles, ou dans notre cas, des fonctions récurives, analyser chaque itération puis effectuer l'union des résultats possibles demande un nombre d'opérations arbitrairement grand, voire infini. On doit donc se munir d'un opérateur capable d'assurer la convergence de telles itérations, l'élargissement (*widening*) $\nabla : A \times A \rightarrow A$, qui sur-approxime l'union et force la convergence dans A en temps fini.

À partir de cette théorie, on trouve dans la littérature de nombreux domaines abstraits numériques abstrayant $\mathcal{P}(\mathbb{Z}^n)$, avec différents compromis entre coût et précision (intervalles, octogones, polyèdres, etc.), dont certains sont relationnels — i.e. ils permettent de garder trace des relations entre les variables plutôt que de les abstraire indépendamment. Nous les exploiterons par la suite pour construire nos domaines pour les types algébriques.

3.2 Abstractions relationnelles avec des variables symboliques

Nous présentons ensuite les domaines abstraits utilisés pour définir une sémantique abstraite calculable $\mathbb{E}^\#$. Traditionnellement, une telle sémantique prend en paramètre une expression et renvoie une valeur abstraite dans le domaine pertinent. Cependant,

$$\begin{aligned}
\mathcal{E}_{\mathbb{Z}} &::= n \in \mathbb{Z} \mid v \in \mathbb{V}_{\mathbb{Z}} \mid e_1 \oplus e_2, \text{ où } e_1, e_2 \in \mathcal{E}_{\mathbb{Z}} \\
\mathcal{V}^{\sharp} &= \mathcal{D} \cup \mathbb{V}_{\mathbb{Z}} \\
\mathcal{E}^{\sharp} &= \mathcal{D} \cup \mathcal{E}_{\mathbb{Z}} \\
\Sigma^{\sharp} &= (\mathbb{V} \rightarrow \mathcal{V}^{\sharp}) \times \mathcal{D}_{\mathbb{Z}}
\end{aligned}$$

Figure 2 – Formalisation de la sémantique abstraite.

cette méthode ne garde pas trace des relations entre variables numériques.

```

let x = random 1 10 in
let y = x + 1 in y

```

En effet, une telle analyse interprétera sur le programme ci-contre x et y comme des valeurs abstraites et pourra inférer, dans le domaine des intervalles, $x \rightarrow [1, 10]$ et $y \rightarrow [2, 11]$, mais pas l'information relationnelle $y = x + 1$. Pour pallier cela, on choisit d'abstraire une expression numérique par une expression symbolique, qui peut contenir des variables, et un environnement abstrait adjoint à un domaine numérique, qui garde la relation entre ces variables. Ce principe est utilisé dans l'analyseur MOPSA [19], pour l'analyse de C et de Python, et permet aux expressions non-numériques d'exploiter la relationnalité de domaines abstraits tels les domaines numériques relationnel — relation entre tailles de chaînes de caractère, par exemple.

La figure 2 définit nos ensembles abstraits. En notant $\mathbb{V}_{\mathbb{Z}}$ l'ensemble des variables représentant des entiers, $\mathcal{E}_{\mathbb{Z}}$ est l'ensemble des expressions numériques, et contient donc des variables symboliques. On note ensuite \mathcal{D} l'union des domaines des objets algébriques et des fonctions, définis par la suite, et $\mathcal{D}_{\mathbb{Z}}$ un domaine abstrait numérique existant sur $\mathbb{V}_{\mathbb{Z}}$ — tels les intervalles ou les polyèdres. En particulier, il peut être relationnel. \mathcal{V}^{\sharp} correspond alors à l'ensemble des valeurs abstraites, où les valeurs numériques sont représentées par des variables, et \mathcal{E}^{\sharp} à l'ensemble des expressions abstraites. On note enfin Σ^{\sharp} l'environnement abstrait : le premier composant est une fonction des variables dans les valeurs abstraites, le second composant donne une valeur aux variables numériques, et notamment aux variables symboliques. On cherche dans cette partie à définir la sémantique abstraite $\mathbb{E}^{\sharp}[\cdot] : \mathcal{E} \times \Sigma^{\sharp} \rightarrow \mathcal{E}^{\sharp} \times \Sigma^{\sharp}$.

Enfin, on note $s[x \rightarrow v]$ l'association de la valeur $v \in \mathcal{V}^{\sharp}$ à la variable $x \in \mathbb{V}$ dans un environnement $s \in (\mathbb{V} \rightarrow \mathcal{V}^{\sharp})$, et avec $\mathbb{E}_{\mathbb{Z}}$ la fonction de transfert, supposée donnée, du domaine numérique, $\mathbb{E}_{\mathbb{Z}}[v_x \leftarrow v]$ est l'association de la variable v_x à une valeur ou une expression symbolique v dans le domaine numérique. Elle renvoie un environnement abstrait dont la deuxième composante d est mise à jour. On note $\sqcup_{\Sigma^{\sharp}}$ l'union sur les environnements. On définit alors l'association de $v \in \mathcal{V}^{\sharp}$ à $x \in \mathbb{V}$ dans $(s, d) \in \Sigma^{\sharp}$:

$$\forall (s, d) \in \Sigma^{\sharp}, (s, d)[x \rightarrow v] = \begin{cases} \mathbb{E}_{\mathbb{Z}}[v_x \leftarrow v]((s[x \rightarrow v_x], d)) & \text{si } x : \mathbf{int} \\ (s[x \rightarrow v], d) & \text{sinon} \end{cases}$$

Lorsque x est entière, on la représente par la variable v_x dans $\mathbb{V} \rightarrow \mathcal{V}^{\sharp}$, avant d'ef-

fectuer l'assignation de v_x à v dans le domaine numérique. Sinon, on associe x à v dans s . Dans ce contexte, la sémantique abstraite de notre programme peut renvoyer : $(v_y, ([x \rightarrow v_x][y \rightarrow v_y], [1 \leq v_x \leq 10][v_y = v_x + 1]))$. La variable numérique v_y correspondant à y est alors bien en relation avec la variable numérique v_x correspondant à x . On différencie ici la variable numérique v_x sur la quantité numérique et la variable de programme x . Notons enfin que $\forall e \in \mathcal{E}, \mathbb{E}^\#[\llbracket e \rrbracket] \emptyset = (\perp^\#, \emptyset)$. Par souci de simplification, on supposera par la suite que les erreurs sont propagées implicitement : si l'abstraction d'une sous-expression contient ω , l'expression aussi.

3.3 Une abstraction pour les types algébriques

Les types algébriques sont fréquemment manipulés en programmation fonctionnelle — on peut citer le type des listes d'entiers simplement chaînées :

```
type list = Cons of int * list | Nil
```

On souhaite pouvoir analyser des programmes manipulant des objets de ce type. Plus généralement, on cherche à disposer d'un domaine spécialisé pour des objets dont le type est un type algébrique (potentiellement récursif) de la forme :

```
type  $\tau = C_1$  of  $\tau_{1,1} * \dots * \tau_{1,m_1} \mid \dots \mid C_n$  of  $\tau_{n,1} * \dots * \tau_{n,m_n}$ 
```

Bautista et al. [2, 3] s'intéressaient déjà aux types algébriques lorsque ceux-ci étaient non récursifs. À notre connaissance, aucun domaine visant à abstraire les valeurs prises par un type algébrique *récursif* n'a pour l'heure été créé. On note $\mathbb{C}_\tau = \{C_i \mid 1 \leq i \leq m\}$ l'ensemble des constructeurs du type τ et \mathcal{T}_τ l'ensemble des objets concrets *finis* de type τ . Remarquons que ces objets sont cependant de taille non bornée, et \mathcal{T}_τ peut être infini. On cherche à définir, pour le type $\tau_{i,j}$, un domaine abstrait $\mathcal{D}_{i,j}^\#$ pour représenter tous les objets de type $\tau_{i,j}$ contenus dans le champ j des constructeurs C_i accessibles dans un objet de type τ .

$$\mathcal{D}_{i,j}^\# = \begin{cases} \mathcal{P}(\mathbb{C}_\tau) & \text{si } \tau_{i,j} = \tau \\ \mathbb{V}_\mathbb{Z} & \text{si } \tau_{i,j} = \text{int} \\ \mathcal{D}_{\tau_{i,j}}^\# & \text{le domaine du type } \tau_{i,j} \text{ déjà défini sinon} \end{cases}$$

On cherche désormais à définir un ensemble abstrait pour l'ensemble concret \mathcal{T}_τ :

$$\mathcal{D}_\tau^\# = \prod_{\substack{1 \leq i \leq m \\ 1 \leq j \leq n_i}} (\mathcal{D}_{i,j}^\#)^\perp \times \mathcal{P}(\mathbb{C}_\tau)$$

- Intuitivement, $\mathcal{D}_\tau^\#$ abstrait un objet x de type τ en un objet abstrait (g, \mathcal{C}) , où :
- $g \in \prod (\mathcal{D}_{i,j}^\#)^\perp$ est tel qu'en notant $g_{i,j}$ l'élément de $(\mathcal{D}_{i,j}^\#)^\perp$ dans le tuple g , celui-ci abstrait toutes les valeurs du j -ième élément des constructeurs C_i de x .
 - $\mathcal{C} \in \mathcal{P}(\mathbb{C}_\tau)$ est l'ensemble des constructeurs C_i possibles pour x

On crée alors $x.i.j$ une variable numérique symbolique, qui représente toujours le champ j des constructeurs C_i accessibles dans x lorsque celui-ci est entier. Ainsi, le domaine numérique peut inférer des relations entre ces variables. Remarquons alors que $\mathcal{D}_{i,j}^\# \neq \mathcal{D}_{\tau_{i,j}}^\#$ lorsque $\tau_{i,j} = \tau$ ou int .

Exemple 3.1. Dans le cas des listes d'entiers, en choisissant pour $\mathcal{D}_{\mathbb{Z}}$ l'ensemble des intervalles \mathbb{I} , on a $\mathbb{C}_{\text{list}} = \{\mathbf{Cons}, \mathbf{Nil}\}$, $\mathcal{D}_{\text{list}}^{\#} = (\mathbb{V}_{\mathbb{Z}}^{\perp} \times \mathcal{P}(\mathbb{C}_{\text{list}})^{\perp}) \times \mathcal{P}(\mathbb{C}_{\text{list}})$. En effet, le constructeur \mathbf{Cons} a deux champs : le premier contient des entiers, représenté par les variables numériques $\mathbb{V}_{\mathbb{Z}}$ — associées dans un environnement σ à des éléments de \mathbb{I} — et le second, récursif, des listes, donc représenté par $\mathcal{P}(\mathbb{C}_{\text{list}})$. Pour un objet x , $x.1.1$ représente alors le champ 1 du premier constructeur, donc le champ `int` de \mathbf{Cons} . Un objet de $\mathcal{D}_{\text{list}}^{\#}$ serait $((r.1.1, \{\mathbf{Cons}, \mathbf{Nil}\}), \{\mathbf{Cons}\}) \in \mathcal{D}_{\text{list}}^{\#}$, avec $[r.1.1 \rightarrow [1, 10]] \in \sigma$.

L'étape suivante est de définir la concrétisation γ_{τ} qui à un objet abstrait de $\mathcal{D}_{\tau}^{\#}$ dans un environnement abstrait associera un ensemble d'objets concrets de \mathcal{T}_{τ} pouvant lui correspondre. On note $\gamma_{\tau_{i,j}}$ la concrétisation associée au domaine de $\tau_{i,j}$ lorsque $\tau_{i,j} \neq \tau$.

Définition 3.1. Pour $(g, \mathcal{C}) \in \mathcal{D}_{\tau}^{\#}$, $\sigma^{\#}$ un environnement abstrait, on a la concrétisation :

$$\gamma_{\tau}((g, \mathcal{C}), \sigma^{\#}) = \{ x : \tau \mid x = C_i(x_{i,1}, \dots, x_{i,n_i}) \wedge C_i \in \mathcal{C} \wedge \\ \forall j, x_{i,j} \in \begin{cases} \gamma_{\tau}((g, g_{i,j}), \sigma^{\#}) & \text{si } \tau_{i,j} = \tau \\ \gamma_{\tau_{i,j}}(g_{i,j}, \sigma^{\#}) & \text{sinon} \end{cases} \}$$

Lorsque le type τ est fini, la récursion est bien fondée ($x_{i,j}$ contient strictement moins de constructeurs que x) et cette définition est correcte.

Intuitivement, un objet x de type τ peut alors être abstrait par (g, \mathcal{C}) si :

- Il s'écrit à partir d'un constructeur C_i de \mathcal{C} .
- Chaque j -ième champ de constructeur C_i accessible dans x a :
 - son constructeur dans $g_{i,j}$, quand il est récursif ($\tau_{i,j} = \tau$);
 - ou peut être abstrait par $g_{i,j}$ s'il est non récursif.

Ainsi, $g_{i,j}$ représente donc un résumé de tous les champs j de constructeur C_i accessibles dans x . L'environnement en paramètre permet de garder trace des variables numériques.

Un objet d'un type récursif étant de taille non bornée, il s'agit ici d'obtenir un domaine abstrait permettant de le représenter de manière finie. Pour ce faire, on « replie » chaque champ de constructeurs en un résumé, de sorte à ne mémoriser qu'une variable *résumé* pour chacun d'eux. Cette approche comporte des similitudes avec l'idée de Gopan et al. [14] qui proposait d'utiliser un nombre fixé de dimensions pour sur-approximer les valeurs d'une collection non bornée d'objets numériques, dans le cadre d'analyses de tableaux dynamiques ou de structures allouées sur le tas. Remarquons que le « résumé » ou « repliement » est effectué au site d'allocation de la variable de type algébrique — l'abstraction est dite *object-sensitive* [28].

Exemple 3.2. Dans le cas des listes, en notant $(g, \mathcal{C}) = ((r, \{\mathbf{Cons}, \mathbf{Nil}\}), \{\mathbf{Cons}\})$ et $\sigma^{\#} = [r \rightarrow [1, 10]]$, la concrétisation $\gamma_{\text{list}}((g, \mathcal{C}), \sigma^{\#})$ vaut :

$$\{ x : \text{list} \mid x = \mathbf{Cons}(h, q), q \in \gamma_{\text{list}}(((r, \{\mathbf{Cons}, \mathbf{Nil}\}), \{\mathbf{Cons}, \mathbf{Nil}\}), \sigma^{\#}) \wedge h \in [1, 10] \}$$

En particulier, $\mathbf{Cons}(1, \mathbf{Cons}(4, \mathbf{Cons}(10, \mathbf{Nil}))) \in \gamma_{\text{list}}(((r, \{\mathbf{Cons}, \mathbf{Nil}\}), \{\mathbf{Cons}\}), \sigma^{\#})$, ce qui signifie que $((r, \{\mathbf{Cons}, \mathbf{Nil}\}), \{\mathbf{Cons}\})$ est une sur-approximation correcte de

$\text{Cons}(1, \text{Cons}(4, \text{Cons}(10, \text{Nil})))$ dans σ^\sharp . De manière générale, c'est une sur-approximation correcte pour toutes les listes non vides dont les éléments sont dans $[1, 10]$.

Pour définir un domaine abstrait, on doit également se munir d'une relation d'ordre entre les éléments de \mathcal{D}_τ^\sharp .

Définition 3.2. On note $\sqsubseteq_{i,j}^\perp$ l'ordre plat sur $\mathcal{D}_{i,j}^{\sharp,\perp}$. On définit ensuite l'opérateur \sqsubseteq :

$$(g^1, \mathcal{C}_1) \sqsubseteq (g^2, \mathcal{C}_2) \iff \mathcal{C}_1 \subseteq \mathcal{C}_2 \wedge (\forall 1 \leq i \leq n, C_i \in \mathcal{C}_1 \implies \forall 1 \leq j \leq m_i, g_{i,j}^1 \sqsubseteq_{i,j}^\perp g_{i,j}^2)$$

Exemple 3.3. Pour les listes, $((x.1.1, \mathcal{C}_1), \mathcal{C}'_1) \sqsubseteq ((y.1.1, \mathcal{C}_2), \mathcal{C}'_2) \iff \mathcal{C}'_1 \subseteq \mathcal{C}'_2 \wedge \mathcal{C}_1 \subseteq \mathcal{C}_2 \wedge x.1.1 \sqsubseteq_{\mathbb{I}} y.1.1$. Par exemple, $((x.1.1, \{\text{Nil}\}), \{\text{Cons}\}) \sqsubseteq ((y.1.1, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}\})$ si $x.1.1 \rightarrow [1, 4]$ et $y.1.1 \rightarrow [1, 10]$. $x.1.1$ et $y.1.1$ représentent les valeurs numériques dans les listes d'entiers x et y .

On construit maintenant une abstraction \cup_τ^\sharp sûre pour \cup :

$$(g^1, \mathcal{C}_1) \cup_\tau^\sharp (g^2, \mathcal{C}_2) = (g, \mathcal{C}_1 \cup \mathcal{C}_2) \quad \text{avec } \forall i \leq n, \forall j \leq m_j, g_{i,j} = g_{i,j}^1 \cup_{\mathcal{D}_{i,j}} g_{i,j}^2$$

On définit l'élargissement de la même manière, en remplaçant \cup_τ^\sharp et $\cup_{\mathcal{D}_{i,j}}$ par ∇_τ et $\nabla_{\mathcal{D}_{i,j}}$. On définit de même l'intersection abstraite. Les preuves de sûreté sont fournies en annexe D.1 et D.2. Il s'agit ensuite de définir les fonctions de transfert pour la construction d'objets de type algébrique, c'est-à-dire, étant donné un environnement abstrait σ^\sharp et une expression concrète $C_k(e_1, \dots, e_n)$, en calculer une abstraction (g, \mathcal{C}) dans un nouvel environnement abstrait. Pour ce faire, on utilise un label syntaxique frais r pour l'expression et les $r.i.j$ sont alors des variables fraîches résumant le champ entier du j -ème élément du constructeur i , dont la valeur $h_{i,j}$ est stockée dans l'environnement.

$$\mathbb{E}^\sharp \llbracket C_k(e_1, \dots, e_n)^r \rrbracket \sigma^\sharp = (g, \{C_k\}), \sigma_0^\sharp[\forall i \leq n, j \leq m_i, r.i.j \rightarrow h_{i,j}] \quad \text{avec}$$

$$\begin{cases} v_i, \sigma_i^\sharp = \mathbb{E}^\sharp \llbracket e_i \rrbracket \sigma^\sharp \\ (g^0, \mathcal{C}) = \bigcup_{e_i:\tau}^\sharp v_i \\ \sigma_0^\sharp = \bigsqcup_i \sigma_i^\sharp \end{cases} \quad h_{i,j} = \begin{cases} g_{k,j}^0 \cup_{\mathcal{D}_{k,j}} v_j & \text{si } i = k \\ \tau_{k,j} \neq \tau & \\ g_{i,j}^0 & \text{sinon} \end{cases} \quad g_{i,j} = \begin{cases} r.i.j & \text{si } \tau_{i,j} = \text{int} \\ h_{i,j} & \text{sinon} \end{cases}$$

L'idée est la suivante : la sémantique abstraite de $C_k(e_1, \dots, e_n)$ est un objet x du domaine \mathcal{D}_τ^\sharp . Le seul constructeur possible pour cet objet est C_k . On abstrait ensuite ses champs de la manière suivante, dans la variable g : on note (g^0, \mathcal{C}) l'union abstraite des évaluations de tous ses champs récurifs — les e_i de type τ . De la sorte, le champ $g_{i,j}^0$ représente l'union de tous les contenus des champs j des C_i accessibles depuis l'objet x .

On définit g en définissant ses composantes $g_{i,j}$, censées représenter tous les objets dans un champ j de constructeur C_i accessibles dans x :

- Si $g_{k,j}$ correspond à un champ non récurif de C_k : il vaut l'union de tous les champs j des C_k accessibles, c'est-à-dire l'union de la sémantique abstraite de e_j
- un champ j de C_k — avec $g_{k,j}^0$ — résumé de tous les autres champs j des C_k .

— Sinon : tous les objets dans un champ j de constructeur C_i sont accessibles uniquement via les champs récurifs, et donc résumés dans $g_{i,j}^0$. Les champs $g_{i,j}$ de type `int` sont alors effectivement assignés à la variable $r.i.j$, de valeur $h_{i,j}$ dans l’environnement. Cette indirection permet, dans le cas où $\mathcal{D}_{\mathbb{Z}}$ est relationnel, d’obtenir des relations entre les $r.i.j$, comme illustré dans l’exemple 3.6. On remarque ainsi que ce domaine s’appuie largement sur les domaines numériques. Sa preuve de sûreté est fournie en annexe D.3.

Exemple 3.4. Ainsi, on peut écrire

$$\begin{aligned}\mathbb{E}^{\sharp}[\llbracket \text{Nil} \rrbracket] \sigma^{\sharp} &= ((r.1.1, \perp), \{\text{Nil}\}), \sigma^{\sharp}[r.1.1 \rightarrow \perp] \\ \mathbb{E}^{\sharp}[\llbracket \text{Cons}(10, \text{Nil}) \rrbracket] \sigma^{\sharp} &= ((r.1.1, \{\text{Nil}\}), \{\text{Cons}\}), \sigma^{\sharp}[r.1.1 \rightarrow \perp \cup_{\mathbb{I}} [10, 10]] \\ \mathbb{E}^{\sharp}[\llbracket \text{Cons}(1, \text{Cons}(10, \text{Nil})) \rrbracket] \sigma^{\sharp} &= (r.1.1, \{\text{Nil}\} \cup \{\text{Cons}\}), \{\text{Cons}\}), \sigma^{\sharp}[x \rightarrow [1, 10]]\end{aligned}$$

On remarque ici qu’on utilise largement les informations de type du programme. En effet, le type des champs i, j (`t` ou `int`, par exemple) guide l’abstraction en renvoyant la gestion des champs vers les domaines pertinents ; pour ce faire, il est nécessaire de connaître la définition du type `t`. On repose donc très fortement sur l’hypothèse que le programme analysé est bien typé.

Exemple 3.5. On peut illustrer l’utilisation de ce domaine abstrait :

```
type tree = Node of tree * int * tree | Leaf of int
let x = Node(Node(Leaf(250), 100, Leaf(251)), 1, Leaf(252))
```

Dans ce contexte, $\mathbb{E}^{\sharp}[\llbracket \text{Node}(\text{Node}(\text{Leaf}(250), 100, \text{Leaf}(251)), 1, \text{Leaf}(252)) \rrbracket] []$ vaut donc $(g, \mathcal{C}) = ((\{\text{Node}, \text{Leaf}\}, x.1.2, \{\text{Leaf}\}, x.2.1), \{\text{Node}\})$ dans l’environnement $\sigma^{\sharp} = ([v_{x.1.2} \rightarrow x.1.2][v_{x.2.1} \rightarrow x.2.1], [v_{x.1.2} \rightarrow [1, 100]][v_{x.2.1} \rightarrow [250, 252]])$.

$$\begin{aligned}\gamma((g, \mathcal{C}), \sigma^{\sharp}) &= \{x \mid x = \text{Node}(t_1, n, t_2) \wedge n \in [1, 100] \wedge t_1 \in \gamma((g, \{\text{Node}, \text{Leaf}\}) \\ &\quad t_2 \in \gamma((g, \{\text{Leaf}\}))\end{aligned}$$

Cela correspond à l’ensemble des arbres construits à partir de `Node` et poussant uniquement vers la gauche, dont les entiers des constructeurs `Leaf` sont entre 250 et 252 et ceux de `Node` entre 1 et 100.

Cet exemple permet de constater qu’une telle analyse perd beaucoup en précision dans des programmes simples. On pourrait par exemple souhaiter pouvoir garder trace du contenu exact de `x` dans un tel exemple, et ne pas effectuer de résumé des champs. Une proposition pour améliorer la précision de l’analyse décrite ci-dessus serait de n’effectuer un résumé d’un champ que lorsque nécessaire pour la convergence, i.e. lors d’une boucle ou d’un appel récurif, ou encore uniquement à partir d’une certaine profondeur n de l’objet. De telles améliorations n’ont pas encore été implémentées.

Cet exemple permet également de voir que bien qu'ils soient de mêmes types, le deuxième champ du constructeur `Node` est résumé dans une variable différente que le champ du constructeur `Leaf`. Cela permet de gagner en précision, notamment lorsque ces champs revêtent des significations différentes.

Exemple 3.6. L'analyse est également capable d'inférer des propriétés relationnelles :

```
let z = Cons(a, Nil) in
let x = if y <= 2*a + 1 then Cons(y, z) else Cons(2 * a + 1, z)
```

En choisissant comme domaine numérique le domaine des polyèdres [7], on est ici en mesure de prouver : $x_{1,1} \leq 2a + 1$. Cela signifie que toutes les éléments de la liste sont inférieurs à $2a + 1$. Cet aspect relationnel est crucial dans l'inférence des relations entrées-sorties des fonctions (section 3.5), effectuée sans information sur les arguments.

3.4 Abstraction du filtrage par motifs

On s'intéressera ici à la fonction de transfert du filtrage par motifs. Celles des autres constructions du langage sont très standard et sont détaillées en Annexe B. Pour ce faire, on se munit de la fonction abstraite $\text{match}^\sharp : \mathcal{V}^\sharp \times \Sigma^\sharp \times \Pi \rightarrow \Sigma^\sharp \times \Sigma^\sharp$, détaillée en Annexe C.3. Elle prend en paramètre une valeur abstraite e , un environnement abstrait σ^\sharp , et un motif p , et renvoie un environnement abstrait restreint dans lequel p et e correspondent, et un dans lequel ils ne correspondent pas. On utilise pour ce faire la fonction de filtre \mathcal{F}^\sharp , fournie par le domaine numérique : étant donnée une expression numérique et un environnement abstrait, elle restreint cet environnement abstrait de sorte que l'expression numérique en paramètre soit non nulle. Par exemple, cela permet d'écrire lorsque le motif s'écrit p when e_1 :

$$\text{match}^\sharp(\sigma^\sharp, v^\sharp, p \text{ when } e_1) = \mathcal{F}^\sharp[\![e_1^n = 0]\!] \sigma_1^\sharp, \sigma_{-,0}^\sharp \text{ avec } \begin{cases} \text{match}^\sharp(\sigma^\sharp, v^\sharp, p) = \sigma_0^\sharp, \sigma_{-,0}^\sharp \\ \mathbb{E}^\sharp[\![e_1]\!] \sigma_0^\sharp = e_1^n, \sigma_1^\sharp \end{cases}$$

Ainsi, v^\sharp et p when e_1 correspondent dans l'environnement où v^\sharp et p correspondent, restreint au cas où e_1^n est non nul.

$$\mathbb{E}^\sharp[\![\text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m]\!] \sigma^\sharp = \begin{cases} \mathbb{E}^\sharp[\![e_1]\!] \sigma_1^\sharp \cup^\sharp \mathbb{E}^\sharp[\![e'_1]\!] \sigma_{-,1}^\sharp & \text{si } m \geq 1 \\ \{\omega\}, \sigma^\sharp & \text{si } m = 0 \end{cases}$$

$$\text{avec } \begin{cases} \sigma_1^\sharp, \sigma_{-,1}^\sharp = \text{match}^\sharp(\mathbb{E}^\sharp[\![e_0]\!] \sigma^\sharp, p_1) \\ e'_1 = \text{match } e_0 \text{ with } p_2 \rightarrow e_2 \mid \dots \mid p_m \rightarrow e_m \end{cases}$$

L'interprétation du filtrage de motif est l'union abstraite des interprétations e_i dans un environnement dans lequel e_i correspond au motif p_i mais pas aux motifs p_j avec $j < i$, $\{\omega\}$ si e_0 ne correspond à aucun motif. Cela correspond à l'intuition que l'expression e_0

est comparée successivement aux différents motifs, jusqu'à la première correspondance. On remarque qu'on renvoie une erreur lorsqu'après lecture du dernier motif, on analyse un filtrage sans motif dans un environnement non vide : cela correspond au cas où il reste des environnements possibles n'ayant été filtrés par aucun motif. Ainsi, notre analyse cible les échecs de filtrage, d'assertions, de division par zero et de dépassements d'entier — si le domaine numérique les traite. Les autres erreurs sont déjà évitées par le typage. La preuve de sûreté pour ces fonctions de transfert est fournie en Annexe D.3.

Exemple 3.7. Sur un exemple manipulant la structure de filtrage de motifs :

```
let x = match Cons(1, Nil) with
| Cons(h, q) -> h
| Nil -> 0
in assert (x = 1)
```

Informellement, on obtient comme abstraction $\mathbb{E}^\#[\text{Cons}(h, q)]\sigma_1^\# \cup^\# \mathbb{E}^\#[\text{Nil}]\sigma_{-,1}^\#$ avec $\sigma_1^\# = [h \rightarrow 1]$ et $\sigma_{-,1}^\# = \emptyset$. Le résultat sera donc $h \cup^\# \perp^\# = h$ dans l'environnement $[h \rightarrow 1]$.

Ainsi, on est en mesure d'inférer que $x = 1$ et l'assertion est donc prouvée correcte.

3.5 Fonctions récursives

Les fonctions récursives sont un incontournable des langages à la ML tel OCaml. Nous avons choisi ici de représenter les fonctions du premier ordre par des relations entre les variables d'entrées et la sortie, en se reposant donc sur un domaine relationnel pour inférer les relations d'entrée-sortie. Le langage étant pur, celles-ci correspondent simplement à l'état relationnel à la fin de la fonction, réduit aux arguments et à la valeur de retour. On s'intéresse aux fonctions de type $\tau = \tau_1 \rightarrow \dots \rightarrow \tau_{n+1}$ avec $\tau_1, \dots, \tau_{n+1} \in \mathcal{T}_0$. On note $\mathcal{D}_i^\#$ le domaine choisi pour représenter τ_i .

Définition 3.3. On note $\mathcal{D}_\tau^f = \mathcal{P}(\mathbb{V}) \times \mathcal{E}^\# \times \Sigma^\#$ le domaine choisi pour représenter les fonctions de type τ . On considère alors la concrétisation :

$$\gamma_f(x_1, \dots, x_n, e^\#, \sigma^\#) = \{ f : \tau \mid \forall v_1 : \tau_1, \dots, v_n : \tau_n, \forall a_1 \in \mathcal{D}_1^\#, \dots, a_n \in \mathcal{D}_n^\#, \forall i, \\ v_i \in \gamma(a_i, \sigma^\#) \wedge f(v_1, \dots, v_n) \in \gamma(e^\#, \sigma^\#[x_1 = a_1, \dots, x_n = a_n]) \}$$

Ainsi, une fonction est abstraite par : le nom x_1, \dots, x_n de tous ses arguments formels, la valeur abstraite de son résultat $e^\#$, et l'environnement $\sigma^\#$ dans lequel ce résultat est valide. Lorsqu'elle est appliquée à des arguments réels de valeurs v_i abstraites par a_i , on ajoute dans l'environnement les contraintes d'égalité entre les arguments x_i et les v_i . On définit ensuite les fonctions de transfert :

$$\mathbb{E}^\#[\text{fun } x_1 \dots x_n \rightarrow e_1]\sigma^\# = (x_1, \dots, x_n, \mathbb{E}^\#[e_1]\sigma^\#[x_1 \rightarrow \top, \dots, x_n \rightarrow \top]), \sigma^\# \\ \mathbb{E}^\#[\text{let rec } f = e_1 \text{ in } e]\sigma^\# = \mathbb{E}^\#[e]\sigma^\#[f \rightarrow \nabla_{n \in \mathbb{N}} F^n((x_1, \dots, x_k, \perp, \sigma^\#[\forall i \leq k, x_i \rightarrow \top]))]$$

$$\text{avec } F(v^\#) = \text{fst}(\mathbb{E}^\#[e_1]\sigma^\#[f \rightarrow v^\#]), \quad f : \tau, \quad \tau = \tau_1 \rightarrow \dots \rightarrow \tau_k \rightarrow \tau_{k+1}$$

$$\mathbb{E}^\#[e_0 \dots e_n]\sigma^\# = e^\#, \sigma_n^\#[x_1 = e_1^\#, \dots, x_n = e_n^\#], \quad \text{avec } \begin{cases} \mathbb{E}^\#[e_0]\sigma^\# = (x_1, \dots, x_n, e^\#, \sigma_0^\#), \\ \mathbb{E}^\#[e_{i+1}]\sigma_i^\# = e_{i+1}^\#, \sigma_{i+1}^\# \end{cases}$$

Pour les fonctions définies récursivement, l'intuition est la suivante. On souhaite réaliser une analyse modulaire de la fonction. Pour analyser le corps de la fonction, on considère donc les valeurs des arguments initialisées à \top , le maximum du domaine. En effet, pour obtenir la relation entrée-sortie la plus générale possible, on doit supposer les arguments quelconques. De la sorte, on peut appliquer le résumé obtenu à chaque appel, sans contrainte sur la valeur des arguments. Lorsque la fonction est récursive, on cherche à sur-approximer le plus petit point fixe de la sémantique concrète. Par le théorème de Kleene, calculer ce point fixe correspond à itérer l'analyse en débutant à $f \rightarrow \perp$. Dans l'abstrait, on procède de même, mais en réalisant un élargissement, ou *widening*, noté ∇_τ , pour assurer la convergence. Enfin, l'application $e_0 \dots e_n$ de la fonction e_0 aux arguments e_1, \dots, e_n ajoute les contraintes d'égalités entre les arguments x_i par l'interprétation abstraite des arguments e_i , dans l'environnement du résultat de la fonction enrichie par les informations obtenues par abstraction des e_j précédents.

On choisit ici d'effectuer l'analyse d'une fonction dès sa déclaration. En effet, cela permet de ne l'analyser qu'une fois, et d'en instancier le résultat à chaque site d'appel selon les valeurs des paramètres. Cela permet une analyse plus efficace. Dans le cas de fonctions mutuellement récursives, non mentionnées ici, notre raisonnement s'étend sans difficulté en itérant sur un vecteur d'abstractions de fonctions.

En notant \mathbb{A} l'ensemble des types algébriques, \mathcal{D} l'union des domaines d'objets algébriques et des domaines de fonctions est alors $\mathcal{D} = \bigcup_{\tau \in \mathbb{A}} \mathcal{D}_\tau^\# \cup \bigcup_{\tau = \tau_1 \rightarrow \dots \rightarrow \tau_n} \mathcal{D}_\tau^f$. Pour obtenir notre domaine abstrait global, on définira l'union $\cup^\#$ sur $\mathcal{E}^\# \times \Sigma^\#$ en Annexe C.2 et de la concrétisation $\gamma : \mathcal{E}^\# \times \Sigma^\# \rightarrow \mathcal{V}$. Sous certaines conditions, il existe une correspondance de Galois (annexe E).

Exemple 3.8. Cependant, cette méthode peut induire une perte de précision :

```
let rec binary = fun x -> if x > 0 then 10 else -10 in binary 1
```

Ici, notre analyse est en mesure de déduire, en analysant `binary` à sa déclaration, qu'étant donnée une entrée quelconque, elle renvoie une valeur dans l'intervalle $[-10, 10]$. Ainsi, on sait que la valeur de `binary 1` est entre -10 et 10. On aurait pu obtenir un résultat plus précis en analysant le corps de `binary` sachant que l'argument `x` valait 1, ou qu'il était strictement supérieur à 0.

On pourrait également choisir de ne pas être modulaire et d'effectuer cette analyse pour des arguments connus. Ainsi, il suffit de les initialiser non pas à \top mais à leur valeur abstraite. On réalise une analyse identique par élargissement, en se donnant cette fois la possibilité de réaliser des élargissements sur les arguments si les appels récursifs élargissent leur domaine. On pourrait même adapter la méthode de partitionnement, développée par Bourdoncle [5] pour un langage récursif impératif : la fonction est analysée pour plusieurs valeurs abstraites couvrant les arguments d'entrée. Cette méthode permet de gagner en précision sans pour autant analyser ou spécialiser la fonction à chaque appel. Cette implémentation serait l'objet d'un travail futur.

Enfin, l'hypothèse d'un fragment pur est essentielle ici. En effet, une fonction est abstraite comme une relation entre les variables d'entrée et la variable de sortie : cette abstraction est permise puisque la fonction ne réalise aucune modification de l'état mémoire. Dans un fragment impur, contenant tableaux, références ou champs mutables, il serait nécessaire de modifier cette abstraction de sorte à garder trace des modifications de l'état mémoire. Des pistes pour travaux futurs seront résumées en section 6.

3.6 Exemple d'application

Détaillons l'analyse en calculant la sémantique abstraite de l'exemple introductif :

```
let rec mult2 = fun l -> match l with
  | Cons(h, q) -> Cons(2*h, mult2 q) | Nil -> Nil in
let hd = fun y -> match y with Cons(h,q) -> h | Nil -> 0 in
let x = Cons(0, Cons(1, Cons(2, Nil))) in assert(hd (mult2 x) <= 4)
```

On note $body = \text{match } l \text{ with } | \text{Cons}(h, q) \rightarrow \text{Cons}(2 * h, \text{mult2 } q) | \text{Nil} \rightarrow \text{Nil}$. On calcule alors $\nabla_{n \in \mathbb{N}} F^n((l, \perp, \sigma^\#))$, où $\sigma^\# = [l \rightarrow \top]$ puisque l est quelconque. Enfin, on note r l'identifiant unique pour le résultat de $body$. Comme l'analyse est générique en le domaine $\mathbb{D}_{\mathbb{Z}}$ choisi pour représenter les entiers, les opérateurs $+$, $-$, $*$ le sont aussi.

$$\begin{aligned}
F((l, \perp, \sigma^\#)) &= \mathbb{E}^\#[\llbracket \text{fun } l \rightarrow body \rrbracket \sigma^\# [mult2 \rightarrow (l, \perp, \sigma^\#)]] \\
&= (l, ((r.1.1, \perp), \{\text{Cons}\}, \sigma^\#[r.1.1 \rightarrow 2 * l.1.1]) \\
&\quad \cup_r^\# ((r.1.1, \perp), \{\text{Nil}\}, \sigma^\#[r.1.1 \rightarrow \perp])) \\
&= (l, (r.1.1, \perp), \{\text{Cons}, \text{Nil}\}, \sigma^\#[r.1.1 \rightarrow 2 * l.1.1]) \\
F^2((l, \perp, \sigma^\#)) &= F((l, (r.1.1, \perp), \{\text{Cons}, \text{Nil}\}, \sigma^\#[r.1.1 \rightarrow 2 * l.1.1])) \\
&= (l, (r.1.1, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\}, \sigma^\#[r.1.1 \rightarrow 2 * l.1.1]) \\
F^3((l, \perp, \sigma^\#)) &= F^2((l, \perp, \sigma^\#)) \\
\nabla_{n \in \mathbb{N}} F^n((l, \perp, \sigma^\#)) &= (l, (r.1.1, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}, \text{Nil}\}, \sigma^\#[r.1.1 \rightarrow 2 * l.1.1]) = V_1
\end{aligned}$$

Par raffinement successif du résumé de la fonction, on est capable d'inférer que la variable résumé des éléments de la liste de retour ($r.1.1$) vaut deux fois la variable résumé des éléments de la liste d'entrée ($l.1.1$). Ainsi, si $\mathcal{D}_{\mathbb{Z}}$ est le domaine des intervalles et que les éléments de la liste d'entrée sont dans $[a, b]$, alors ceux de la liste de retour sont dans $[2a, 2b]$. On associe ensuite ce résultat à $mult2$ et on poursuit l'analyse de e :

$$\mathbb{E}^\#[\llbracket \text{let rec } mult2 = body \text{ in } e \rrbracket \sigma^\#] = \mathbb{E}^\#[\llbracket e \rrbracket \sigma^\# \sigma_1^\#] \quad \text{avec } \sigma_1^\# = \sigma^\#[mult2 \rightarrow V_1]$$

Puis on évalue $body_2 = \text{match } x \text{ with } | \text{Cons}(h, q) \rightarrow h \mid \text{Nil} \rightarrow 0$, qu'on associe à hd .

$$\begin{aligned} \mathbb{E}^\sharp[\text{fun } y \rightarrow body_2]\sigma_1^\sharp &= (y, \mathbb{E}^\sharp[body_2]\sigma_1^\sharp[y \rightarrow \top]), \sigma_1^\sharp \\ &= (y, z, \sigma_1^\sharp[z \rightarrow y.1.1 \cup_{\mathbb{Z}} 0]), \sigma_1^\sharp = V_2, _ \\ \mathbb{E}^\sharp[\text{let } hd = body_2 \text{ in } e_1]\sigma_1^\sharp &= V_2, \sigma_2^\sharp \quad \text{où } \sigma_2^\sharp = \sigma_1^\sharp[hd \rightarrow V_2] \end{aligned}$$

Ainsi, hd renvoie z , dont la valeur est le résumé du premier champ de Cons pour l ($l.1.1$) joint à 0. Enfin, on calcule l'abstraction de x .

$$\begin{aligned} \mathbb{E}^\sharp[\text{Cons}(0, \text{Cons}(1, \text{Cons}(2, \text{Nil})))]\sigma_2^\sharp &= ((r_2.1.1, \{\text{Cons}, \text{Nil}\}), \{\text{Cons}\}), \sigma_3^\sharp \\ &= V_3, \sigma_3^\sharp \quad \text{où } \sigma_3^\sharp = \sigma_2^\sharp[r_2.1.1 \rightarrow 0 \cup_{\mathbb{Z}} 1 \cup_{\mathbb{Z}} 2] \end{aligned}$$

$$\mathbb{E}^\sharp[\text{let } x = \text{Cons}(0, \text{Cons}(1, \text{Cons}(2, \text{Nil}))) \text{ in } e_2]\sigma_2^\sharp = \mathbb{E}^\sharp[e_2]\sigma_2^\sharp, \sigma_4^\sharp \quad \text{où } \sigma_4^\sharp = \sigma_3^\sharp[x \rightarrow V_3]$$

Puis on analyse l'assertion en appliquant successivement $mult2$ et hd à x :

$$\begin{aligned} \mathbb{E}^\sharp[mult2 \ x]\sigma_4^\sharp &= V_1, \sigma_4^\sharp[l \rightarrow \sigma_4^\sharp(x)] \\ \mathbb{E}^\sharp[hd \ (mult2 \ x)]\sigma_4^\sharp &= z_1, \sigma_4^\sharp[l \rightarrow \sigma_4^\sharp(x)][y \rightarrow V_1][z_1 \rightarrow z] \end{aligned}$$

z est la variable résultat de $body_2$, et z_1 est l'interprétation abstraite de l'expression testée par l'assertion. Avec le domaine des polyèdres, on a alors :

$$0 \leq x.1.1 \leq 2 \wedge l.1.1 = x.1.1 \wedge V_1.1.1 \leq 2 * l.1.1 \wedge y.1.1 = V_1.1.1 \wedge 0 \leq z \leq y.1.1 \wedge z_1 = z$$

On déduit $0 \leq V_1.1.1 \leq 4$, puis $0 \leq z_1 \leq 4$. Ainsi, le programme est prouvé correct.

4 Implémentation et expérimentation

4.1 La plateforme MOPSA

L'implémentation s'est effectuée dans MOPSA (Modular Open Platform for Static Analysis) [19], une plateforme en logiciel libre et multi-langage visant à faciliter le développement d'analyseurs statiques par interprétation abstraite.

Elle se différencie des analyseurs statiques existants, tels Astrée [4], Frama-C [6], Infer [12] ou Julia [29] en ce qu'elle tente de combiner des analyses relationnelles basées sur la coopération de domaines abstraits indépendants les uns des autres avec une plateforme modulaire non spécifique à un langage. Elle implémente des analyses ciblant du code écrit en C, en Python, ou mêlant du C et du Python. Elle vérifie l'absence d'erreurs à l'exécution et d'exceptions non rattrapées, et possède un langage de modélisation pour les bibliothèques C, appliqué, par exemple, à la bibliothèque standard. Pour améliorer la précision, les domaines communiquent et coopèrent via produit réduit, composition ou réécriture, ce qui est facilité par une signature commune.

La plateforme est développée en OCaml; l'environnement de travail consiste en 19 000 lignes de code. Le support pour OCaml, en cours de développement, correspond à

Programme	Lignes	Temps (ms)
<code>list.ml</code>	3	3
<code>tree.ml</code>	2	5
<code>match.ml</code>	4	4
<code>match_alarm.ml</code>	4	5
<code>match_error.ml</code>	4	4
<code>add.ml</code>	3	4

Figure 3 – Temps d’exécution de l’analyse de quelques programmes jouets.

quelques 2000 lignes de code¹. Celui-ci récupère l’arbre de syntaxe abstraite typé dans le fichier `.cmt` résultant de la compilation d’OCaml, de sorte à bénéficier des parseurs et typeurs natifs et à faciliter la transition vers un support complet du langage. Il supporte actuellement les filtrages de motifs, les types algébriques récur­sifs et les fonctions non-récur­sives. Le domaine des fonctions récur­sives est encore en cours d’implémentation.

4.2 Résultats expérimentaux

Une évaluation expérimentale a été réalisée sur des programmes OCaml écrits à la main. Dans la figure 3, on s’intéresse au temps d’exécution de l’analyse sur un ordinateur portable standard pour de courts programmes jouets déclarant des listes ou des arbres et les manipulant via filtrage de motifs, et dont le code peut être trouvé en Annexe F. Sur des exemples simples, l’analyse trouve les résultats attendus de manière rapide. Pour l’heure, nous nous sommes limités à des programmes jouets de quelques lignes. Une extension serait de s’intéresser à des programmes à taille réelle, provenant de projets ou de la bibliothèque standard.

5 État de l’art

Pour établir des propriétés sur des programmes fonctionnels, les systèmes de types ont été largement utilisés. Les plus simples permettent de prévenir un certain nombre d’erreurs, telle l’addition d’un entier à une fonction. D’autres, tels les types de raffinements [31, 32], permettent d’exprimer des propriétés complexes sur des valeurs algébriques ou numériques mais délèguent pour cela des vérifications à des solveurs SMTs. Les méthodes dites déductives, telles Why3 [13] ou F* [30], peuvent quant à elles établir des propriétés plus fines sur les programmes, telle leur correction par rapport à une spécification. Elles reposent cependant sur des annotations utilisateurs et des solveurs SMT. Notre travail cherche un nouveau compromis, une méthode automatique et sans solveur paramétrée par un choix d’abstraction, pour inférer des propriétés numériques.

Si de nombreuses recherches traitent de l’analyse statique des langages fonctionnels, elles se sont beaucoup concentrées sur l’analyse de flots de contrôles (CFA) [24], ce qui, comme précisé dans [21] ne permet pas de garder trace des valeurs traversant les flots.

1. disponible sur gitlab au lien <https://gitlab.com/mvalnet/mopsa-analyzer/-/tree/ocaml>

Cousot et Cousot [10], eux, se concentrent sur les fonctions comme objets de première classe mais ont pour but d’exprimer par interprétation abstraite des analyses classiques telles la *strictness* ou la terminaison et ne proposent aucune analyse de valeurs.

Dans [23], Montagu et Jensen cherchent à inférer une forme de *frame condition* pour les langages fonctionnels purs, c’est-à-dire identifier des relations d’égalité entre des parties de valeurs algébriques. Une telle approche, si elle se confronte à des problématiques proches, ne vise pas à garder trace des valeurs des variables, non plus que des relations *numériques* entre celles-ci. Elle cherche plutôt à prouver des propriétés de sorte à en décharger les assistants de preuves et à effectuer des optimisations à la compilation. De plus, elle ne permet pas de manipuler des types algébriques récur­sifs.

Bautista et al. [2] proposent quant à eux une méthode d’abstraction relationnelle pour des valeurs algébriques contenant des données numériques, qu’ils enrichissent ensuite [3] en inférant des égalités structurelles entre champs non numériques. Cependant, cette approche, quoique plus précise que la nôtre dans des cas non récur­sifs, ne permet pas l’analyse des structures de données récur­sives tels les listes ou les arbres.

À notre connaissance, le seul analyseur statique de valeurs par interprétation abstraite pour un langage fonctionnel est [16], écrit en 2011. Il génère des contraintes avec sous-typage dont la vérification implique la sûreté du programme, puis les transforme en langage impératif de sorte à réutiliser des analyseurs existants. Cependant, dans cette approche, on perd très tôt la particularité fonctionnelle de notre langage source, ce qui risque d’impliquer une perte de précision. L’article mentionne une implémentation, mais celle-ci n’est pas disponible en ligne, aussi n’avons-nous pu comparer nos résultats.

6 Limitations et prolongements

Nos travaux permettent de mettre en évidence de nombreuses pistes vers une analyse statique de valeurs pour des langages fonctionnels d’ordre supérieur réalistes.

Pour l’heure, nous n’avons pas développé de méthode générique pour le polymorphisme. Pour cette raison, notre approche pour une fonction polymorphe diffère de celle d’une fonction monomorphe : on effectue l’analyse non plus lors de sa définition, mais lors de son application à des arguments de type connu. Ainsi, l’inférence du résumé de la fonction est réalisée dans une spécialisation du type de la fonction, et les méthodes pour les fonctions monomorphes s’appliquent. Cependant, travailler sur une analyse polymorphe permettrait d’améliorer les performances en ne réalisant qu’une seule analyse par variable, mais également d’obtenir des propriétés générales sur les variables polymorphes telles des relations d’égalité et d’inégalité, ainsi que des relations d’ordre dans un langage comme OCaml où tout type est muni d’une relation d’ordre total. Lever cette limitation serait un premier pas pour parvenir à l’analyse d’une bibliothèque.

De même, une méthode possible pour supporter l’ordre supérieur serait de procéder par injection (*inlining*) du corps des fonctions lorsque celles-ci sont appliquées. Ainsi, lorsqu’un argument d’une fonction est d’ordre supérieur, on diffère l’analyse de cette fonction jusqu’à ce qu’il puisse être spécialisé par une fonction connue, se ramenant ainsi

à une analyse du premier ordre. Cependant, notre analyse semble pouvoir s'adapter à l'ordre supérieur : en effet, en modélisant comme des relations, donc comme des points du domaine des polyèdres, des fonctions manipulant des entiers, fonctions comme valeurs sont réduites à un ensemble de points dans un espace vectoriel. La complexité et la précision de cette méthode serait à évaluer, tandis que l'étendre ou créer de nouveaux domaines pour les fonctions manipulant des objets algébriques sont des travaux futurs.

De plus, notre analyse se concentre pour l'heure sur un fragment pur de langage fonctionnel. Cela exclut de nombreuses fonctionnalités — tableaux, références, champs mutables — mais permet de simplifier l'analyse des relations entrées-sorties. Diverses pistes ont été envisagées pour étendre aux fonctionnalités impures. L'impureté d'une variable pouvant être identifiée par son type — `'a array`, `'a ref`, `mutable` — on peut choisir d'abstraire toute variable impure par la valeur maximale du domaine abstrait pour signifier qu'on ne connaît aucune information à son sujet, ou de les passer en paramètre de toute fonction susceptible de les manipuler. Pour améliorer la performance, on pourrait également identifier les fonctions dont les variables impures locales n'échappent pas à leur corps et ainsi utiliser les méthodes précédemment élaborées à moindre surcoût.

Un objectif à long terme serait enfin de s'appuyer sur cette analyse pour créer des domaines plus sophistiqués, capable d'établir automatiquement des propriétés plus fines sur les programmes. De telles utilisations de l'interprétation abstraite ont déjà été réalisées pour des langages impératifs, par Cousot et al. dans [7] pour des analyses de contenu de tableau, par Journault et Miné dans [17] pour prouver la correction de programme manipulant des matrices, ou encore avec la *shape analysis* [27], pour prouver des propriétés sophistiquées sur les listes et les arbres. Nous aimerions nous en inspirer pour créer des domaines capables d'exprimer des propriétés sur la taille des listes, la profondeur des structures de données, ou l'équilibre d'un arbre.

Conclusion

Ainsi, nous avons élaboré des méthodes d'analyse statique de valeur par interprétation abstraite dédiées aux langages fonctionnels. Les domaines abstraits sous-jacents sont relationnels, et permettent de traiter un fragment de langage fonctionnel du premier ordre supportant les types algébriques récurifs ainsi que les fonctions récursives les manipulant. Une partie de ces méthodes a été implémentées au sein de la plateforme MOPSA pour un fragment de OCaml et ont montré des résultats concluants sur un petit ensemble de programmes écrits à la main. Ce travail dégage également de nombreuses pistes vers l'obtention d'une analyse de valeurs précise et efficace pour un langage fonctionnel d'ordre supérieur.

Références

- [1] Bagnara, R., Hill, P.M., Ricci, E. and Zaffanella, E., 2005. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1-2), pp.28-56.
- [2] Bautista, S., Jensen, T. and Montagu, B., 2020, November. Numeric domains meet algebraic

- data types. In Proceedings of the 9th ACM SIGPLAN International Workshop on Numerical and Symbolic Abstract Domains (pp. 12-16).
- [3] Bautista, S., Jensen, T. and Montagu, B., 2022, December. Lifting Numeric Relational Domains to Algebraic Data Types. In *Static Analysis : 29th International Symposium, SAS 2022*, Auckland, New Zealand, December 5–7, 2022, Proceedings (pp. 104-134). Cham : Springer Nature Switzerland.
 - [4] Bertrane, J., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A. and Rival, X., 2015. Static analysis and verification of aerospace software by abstract interpretation. *Foundations and Trends in Programming Languages*, 2(2-3), pp.71-190.
 - [5] Bourdoncle, F., 1992. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4), pp.407-435.
 - [6] Bühler, D., 2017. Structuring an abstract interpreter through value and state abstractions : eva, an evolved value analysis for frama-c (Doctoral dissertation, Université de Rennes 1).
 - [7] P. Cousot & N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL'78*, 84–96, ACM, 1978.
 - [8] Cousot, P. and Cousot, R., 1977. Static determination of dynamic properties of generalized type unions. *ACM SIGOPS Operating Systems Review*, 11(2), pp.77-94.
 - [9] Cousot, P. and Cousot, R., 1977, January. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (pp. 238-252).
 - [10] Cousot, P. and Cousot, R., 1994, May. Higher-order abstract interpretation (and application to compartment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of 1994 IEEE International Conference on Computer Languages (ICCL'94)* (pp. 95-112). IEEE.
 - [11] Cousot, P., Cousot, R. and Logozzo, F., 2011. A parametric segmentation functor for fully automatic and scalable array content analysis. *ACM SIGPLAN Notices*, 46(1), pp.105-118.
 - [12] Distefano, D., Fähndrich, M., Logozzo, F. and O'Hearn, P.W., 2019. Scaling static analyses at Facebook. *Communications of the ACM*, 62(8), pp.62-70.
 - [13] Filiâtre, J.C. and Paskevich, A., 2013, March. Why3—where programs meet provers. In *European symposium on programming* (pp. 125-128). Springer, Berlin, Heidelberg.
 - [14] Gopan, D., DiMaio, F., Dor, N., Reps, T., Sagiv, M., 2004. Numeric Domains with Summarized Dimensions, in : Jensen, K., Podelski, A. (Eds.), *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 512–529. https://doi.org/10.1007/978-3-540-24730-2_38
 - [15] C. Gunter & D. Scott. *Semantic domains*. Vol. B of *Handbook of Theoretical Computer Science*, pp. 633–674. Elsevier, 1990.
 - [16] Jhala, R., Majumdar, R., & Rybalchenko, A. (2011, July). HMC : Verifying functional programs using abstract interpreters. In *International Conference on Computer Aided Verification* (pp. 470-485). Springer, Berlin, Heidelberg.
 - [17] Journault, M. and Miné, A., 2016, September. Static analysis by abstract interpretation of the functional correctness of matrix manipulating programs. In *International Static Analysis Symposium* (pp. 257-277). Springer, Berlin, Heidelberg.
 - [18] Journault, M., 2019. Analyse statique modulaire précise par interprétation abstraite pour la preuve automatique de correction de programmes et pour l'inférence de contrats (Doctoral

- dissertation, Sorbonne université).
- [19] Journault, M., Miné, A., Monat, R. and Ouadjaout, A., 2020. Combinations of reusable abstract domains for a multilingual static analyzer. In Working Conference on Verified Software : Theories, Tools, and Experiments (pp. 1-18). Springer, Cham.
 - [20] Krishnaswami, N., 2006. Separation logic for a higher-order typed language. In Workshop on Semantics, Program Analysis and Computing Environments for Memory Management, SPACE (Vol. 6, pp. 73-82).
 - [21] Liang, S. and Might, M., 2013, November. Entangled abstract domains for higher-order programs. In Proceedings of the 2013 Workshop on Scheme and Functional Programming, Washington, DC.
 - [22] Monat, R., 2021. Static type and value analysis by abstract interpretation of Python programs with native C libraries (Doctoral dissertation, Sorbonne Université).
 - [23] Montagu, B. and Jensen, T., 2020. Stable relations and abstract interpretation of higher-order programs. Proceedings of the ACM on Programming Languages, 4(ICFP), pp.1-30.
 - [24] Montagu, B. and Jensen, T., 2021, June. Trace-based control-flow analysis. In Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (pp. 482-496).
 - [25] Nielson, F. and Nielson, H.R., 1997, January. Infinitary control flow analysis : a collecting semantics for closure analysis. In Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 332-345).
 - [26] Rice, H. G., 1953, Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical society, 74(2), 358-366.
 - [27] Sagiv, M., Reps, T. and Wilhelm, R., 2002. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems (TOPLAS), 24(3), pp.217-298.
 - [28] Smaragdakis, Y., Bravenboer, M. and Lhoták, O., 2011, January. Pick your contexts well : understanding object-sensitivity. In Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (pp. 17-30).
 - [29] Spoto, F., 2016, September. The Julia static analyzer for Java. In International Static Analysis Symposium (pp. 39-57). Springer, Berlin, Heidelberg.
 - [30] Swamy, N., Hrițcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M. and Zinzindohoue, J.K., 2016, January. Dependent types and multi-monadic effects in F. In Proceedings of the 43rd annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (pp. 256-270).
 - [31] Swamy, N., Weinberger, J., Schlesinger, C., Chen, J. and Livshits, B., 2013. Verifying higher-order programs with the Dijkstra monad. ACM SIGPLAN Notices, 48(6), pp.387-398.
 - [32] Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D. and Peyton-Jones, S., 2014, August. Refinement types for Haskell. In Proceedings of the 19th ACM SIGPLAN international conference on Functional programming (pp. 269-282).

Annexe A Sémantique du langage fonctionnel

On définit en Fig 4 la sémantique concrète collectrice de notre langage fonctionnel.

Ainsi, la valeur d'une variable x dans un environnement σ est $\sigma(x)$, celle d'un entier n est ce même entier. L'expression $\text{fun } x_1 \dots x_n \rightarrow e$ est interprétée comme une fonction

$$\begin{aligned}
\mathbb{E}[\cdot] : \Sigma &\rightarrow \mathcal{V} \\
\mathbb{E}[x \in \mathbb{V}]\sigma &= \sigma(x) \\
\mathbb{E}[n \in \mathbb{Z}]\sigma &= n \\
\mathbb{E}[\text{fun } x_1 \dots x_n \rightarrow e]\sigma &= \lambda a_1 \dots a_n. \mathbb{E}[e]\sigma[x_1 \rightarrow a_1, \dots, x_n \rightarrow a_n] \\
\mathbb{E}[e_1 e_2 \dots e_{n+1}]\sigma &= \begin{cases} f[x_i \rightarrow \mathbb{E}[e_i]\sigma] & \text{si } \mathbb{E}[e_1]\sigma = \lambda x_1 \dots x_n. f \wedge \forall 1 \leq i \leq n, \mathbb{E}[e_i]\sigma \in \mathcal{V}_0 \setminus \{\omega\} \\ \omega & \text{sinon} \end{cases} \\
\mathbb{E}[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]\sigma &= \begin{cases} \mathbb{E}[e_2]\sigma & \text{si } \mathbb{E}[e_1]\sigma = z \in \mathbb{Z} \wedge z \neq 0 \\ \mathbb{E}[e_3]\sigma & \text{si } \mathbb{E}[e_1]\sigma = 0 \\ \omega & \text{sinon} \end{cases} \\
\mathbb{E}[\text{assert } e]\sigma &= \begin{cases} 1 & \text{si } \mathbb{E}[e]\sigma = z \in \mathbb{Z} \wedge z \neq 0 \\ \omega & \text{sinon} \end{cases} \\
\mathbb{E}[\text{let } x = e_1 \text{ in } e_2]\sigma &= \begin{cases} \mathbb{E}[e_2]\sigma[x \rightarrow \mathbb{E}[e_1]\sigma] & \text{si } \mathbb{E}[e_1]\sigma \neq \omega \\ \omega & \text{sinon} \end{cases} \\
\mathbb{E}[\text{let rec } x_f = e_1 \text{ in } e_2]\sigma &= \begin{cases} \mathbb{E}[e_2]\sigma[x_f \rightarrow \text{lfp}(F_{x_f, x, g})] & \text{si } \mathbb{E}[e_1]\sigma = \lambda x_1 \dots x_n. g \\ \omega & \text{sinon} \end{cases} \\
\text{avec } F_{x_f, x, g}(\phi) &= \lambda y_1 \dots y_n. \mathbb{E}[g]\sigma[x_i \rightarrow y_i, x_f \rightarrow \phi] \\
\mathbb{E}[e_1 \oplus e_2]\sigma &= \begin{cases} n_1 \oplus n_2 & \text{si } \mathbb{E}[e_1]\sigma = n_1 \in \mathbb{Z} \text{ et } \mathbb{E}[e_2]\sigma = n_2 \in \mathbb{Z} \\ \omega & \text{sinon} \end{cases} \\
\mathbb{E}[C(e_1, \dots, e_n)]\sigma &= \begin{cases} C(\mathbb{E}[e_1]\sigma, \dots, \mathbb{E}[e_n]\sigma) & \text{si } \forall 1 \leq i \leq n, \mathbb{E}[e_i]\sigma \in \mathcal{V}_0 \setminus \{\omega\} \\ \omega & \text{sinon} \end{cases} \\
\mathbb{E}[\text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m]\sigma &= \begin{cases} \mathbb{E}[e_1]\sigma' & \text{si } b_{guard} = \mathbf{true} \\ \mathbb{E}[e'_1]\sigma & \text{si } b_{guard} = \mathbf{false} \wedge m > 1 \\ \omega & \text{sinon} \end{cases} \\
\text{avec } \begin{cases} \text{match}(\sigma, \mathbb{E}[e_0]\sigma, p_1) = (\sigma', b_{guard}), & \text{cf. Fig. 5} \\ e'_1 = \text{match } e_0 \text{ with } p_2 \rightarrow e_2 \mid \dots \mid p_m \rightarrow e_m \end{cases}
\end{aligned}$$

Figure 4 – Sémantique fonctionnelle du langage.

mathématique, qui à des paramètres $a_1 \dots a_n$ renvoie l'objet mathématique correspondant à la sémantique de e dans un contexte où les variables x_i valent a_i . L'application s'interprète de la manière suivante : si e_1 s'interprète en une fonction, on évalue cette dernière avec pour arguments les interprétations des e_i si ceux-ci ne sont pas des fonctions. Sinon, ω est renvoyée. La sémantique du if est naturelle : si e_1 s'interprète en un entier non nul, la valeur est celle de l'interprétation de e_2 , celle de e_3 s'il s'interprète en

$$\begin{aligned}
& \text{match}(\cdot, \cdot, \cdot) : \Sigma \times \mathcal{V} \times \Pi \rightarrow \Sigma \times \mathcal{B} \cup \{\omega\} \\
& \text{match}(\sigma, \omega, p) = \sigma, \omega \\
& \text{match}(\sigma, v, _) = \sigma, \mathbf{true} \\
& \text{match}(\sigma, v, x \in \mathcal{V}) = \sigma[x \rightarrow v], \mathbf{true} \\
& \text{match}(\sigma, v, n_2 \in \mathbb{Z}) = \sigma, \begin{cases} n_1 = n_2 & \text{si } v = n_1 \in \mathbb{Z} \\ \mathbf{false} & \text{sinon} \end{cases} \\
& \text{match}(\sigma, C(v_1, \dots, v_n), C(p_1, \dots, p_n)) = \sigma_n, \bigwedge_{i \in \llbracket 1, n \rrbracket} b_i \\
& \text{avec } \begin{cases} \sigma_0 = \sigma \\ \forall i \in \llbracket 1, n \rrbracket, \sigma_{i+1}, b_{i+1} = \text{match}(\sigma_i, v_i, p_i) \end{cases} \\
& \text{match}(\sigma, v, C(p_1, \dots, p_n)) = \sigma, \mathbf{false} \quad \text{avec } v \neq C(v_1, \dots, v_n) \\
& \text{match}(\sigma, v, p_1 \text{ when } e_1) = \sigma', b \wedge \mathbb{E}[e_1] \sigma' \neq 0 \quad \text{avec } \text{match}(\sigma, v, p) = \sigma', b
\end{aligned}$$

Figure 5 – Définition de `match`.

0, une erreur sinon. L'assert s'interprète en 1 si e s'interprète en un entier non nul, une erreur sinon. Le let correspond à l'interprétation de e_2 dans un environnement où x a comme valeur l'interprétation de e_1 si celle-ci n'est pas une erreur, ω sinon.

Le let rec quant à lui associe à x_f l'interprétation suivante : si e_1 est une fonction, on considère la fonction $F_{x_f, x, g}$, qui à une fonction ϕ associe la fonction qui à y associe l'interprétation du corps g . Cette fonction, $F_{x_f, x, g}(\phi)$, est ainsi une meilleure approximation de x_f que ϕ . On associe alors à x_f le plus petit point fixe de $F_{x_f, x, g}$, défini sur les fonctions continues. Ce point fixe est bien défini [15].

Enfin, $e_1 \oplus e_2$ a pour valeur l'opération mathématique correspondant à \oplus appliquée aux interprétations de e_1 et e_2 si toutes deux sont des entiers, ω sinon. $C(e_1, \dots, e_n)$ s'interprète en interprétant chacune des expressions en paramètres du constructeur de type si celles-ci ne sont pas des fonctions et ne renvoient pas d'erreurs, ω sinon. La sémantique du `match` est plus complexe. On définit pour ce faire la fonction `match` en Fig. 5.

La fonction `match` prend en paramètre un environnement, une valeur et un motif. Elle renvoie un booléen, vrai si et seulement si la valeur peut correspondre au motif, ainsi que l'environnement dans lequel les variables du motif sont associées aux valeurs correspondantes de l'expression.

Ainsi, si le v est un joker `_`, la valeur peut correspondre au motif : le booléen est vrai et l'environnement est inchangé car aucune variable n'a été liée. Si le motif est une variable x , la valeur v peut également correspondre au motif dans un environnement où x est lié à v . Si le motif est un entier $n_2 \in \mathbb{Z}$, alors v correspond à n_2 si et seulement si n_1 et n_2 vus comme des entiers mathématiques sont égaux. Sinon, le booléen est faux.

Enfin, si le motif est sous la forme $C(p_1, \dots, p_n)$ et la valeur de la forme $C(v_1, \dots, v_n)$, la valeur correspond au motif si et seulement si tous les bv_i peuvent correspondre aux p_i . L'environnement est alors l'environnement σ_n dans lequel si la conjonction des b_i est vrai, les v_i correspondent aux p_i . Comme $\forall i, j, fv(p_i) \neq fv(p_j)$, une variable n'a pas été liée plusieurs fois. Quant au motif p_1 when e_1 , la valeur correspond au motif si et seulement si elle correspond à p_1 et e_1 s'interprète en une valeur non nulle.

Pour revenir à la sémantique du filtrage de motif, $\text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m$ est interprété de la manière suivante : si le booléen renvoyé par $\text{match}(\sigma, e_0, p_1)$ est vrai, c'est-à-dire si e_0 et p_1 peuvent correspondre, on évalue e_1 dans l'environnement σ' dans lequel e_0 et p_1 correspondent. Sinon, si $m > 1$, on ré-interprète le match en retirant le cas $p_1 \rightarrow e_1$. Enfin, si $m = 1$ et b_{guard} est faux, l'expression ne correspond à aucun motif et une erreur est renvoyée.

Intuitivement donc, l'expression est séquentiellement comparée aux différents motifs ; sitôt que celle-ci correspond à un motif p_i , l'expression e_i correspondante est évaluée dans le contexte résultant.

Annexe B Fonctions de transfert

On détaille ici les fonctions de transfert pour les constructions du langage.

$$\begin{aligned}
\mathbb{E}^\# \llbracket x \rrbracket (\epsilon, d) &= \epsilon(x), (\epsilon, d) \\
\mathbb{E}^\# \llbracket n \in \mathbb{Z} \rrbracket \sigma^\# &= n, \sigma^\# \\
\mathbb{E}^\# \llbracket \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rrbracket \sigma^\# &= \mathbb{E}^\# \llbracket e_2 \rrbracket (\mathcal{F}^\# \llbracket e_1^n \rrbracket \sigma_1^\#) \cup^\# \mathbb{E}^\# \llbracket e_3 \rrbracket (\mathcal{F}^\# \llbracket \neg e_1^n \rrbracket \sigma_1^\#) \\
&\quad \text{où } \mathbb{E}^\# \llbracket e_1 \rrbracket \sigma^\# = e_1^n, \sigma_1^\# \\
\mathbb{E}^\# \llbracket \text{let } x = e_1 \text{ in } e_2 \rrbracket \sigma^\# &= \mathbb{E}^\# \llbracket e_2 \rrbracket \sigma_1^\# [x \rightarrow e^\#] \quad \text{où } \mathbb{E}^\# \llbracket e_1 \rrbracket \sigma^\# = e^\#, \sigma_1^\# \\
\mathbb{E}^\# \llbracket e_1 \oplus e_2 \rrbracket \sigma^\# &= x \oplus y, \sigma_2^\# [x \rightarrow e_1^n] [y \rightarrow e_2^n], \quad \text{où } \mathbb{E}^\# \llbracket e_1 \rrbracket \sigma^\# = e_1^n, \sigma_1^\#, \mathbb{E}^\# \llbracket e_2 \rrbracket \sigma_1^\# = e_2^n, \sigma_2^\# \\
\mathbb{E}^\# \llbracket \text{assert } e \rrbracket \sigma^\# &= \begin{cases} \{\omega\}, \sigma^\# & \text{si } \mathcal{F}^\# \llbracket e^n \rrbracket \sigma_0^\# \neq \perp, \mathbb{E}^\# \llbracket e \rrbracket \sigma^\# = e^n, \sigma_0^\# \\ \mathbb{E}^\# \llbracket 1 \rrbracket \sigma^\# & \text{sinon} \end{cases}
\end{aligned}$$

Figure 6 – Fonctions de transfert pour les constructions du langage.

Ainsi, l'interprétation abstraite d'une variable renvoie sa valeur dans l'environnement abstrait ainsi que l'environnement abstrait dans lequel ceci est vrai - en effet, si x est un entier, $\epsilon(x)$ est une variable numérique dont les valeurs possibles sont définies par d . L'interprétation de n renvoie l'entier n dans l'environnement inchangé. L'interprétation de $\text{if } e_1 \text{ then } e_2 \text{ else } e_3$ est l'union abstraite de l'interprétation de e_2 dans un environnement restreint dans lequel l'expression numérique associée à e_1 est nécessairement non nulle et de celles de e_3 dans un environnement restreint dans elle est nécessairement nulle. $\text{let } x = e_1 \text{ in } e_2$ a comme interprétation abstraite celle de e_2 dans l'environnement dans lequel x est assigné à l'interprétation de $\mathbb{E}^\# \llbracket e_1 \rrbracket \sigma^\#$ - par définition

de la notation $\sigma^\sharp[x \rightarrow y]$, l'opérateur d'assignation du domaine numérique peut être utilisé pour ce faire. $e_1 \oplus e_2$ a comme interprétation abstraite l'expression numérique $x \oplus y$, où x et y sont des variables fraîches, assignées dans l'environnement de retour aux interprétations de $\mathbb{E}^\sharp[[e_1]]\sigma^\sharp$ et $\mathbb{E}^\sharp[[e_2]]\sigma^\sharp$ - ce qui appelle l'assignation numérique. assert e s'interprète en $\Omega = \{\omega\}$ s'il existe un environnement restreint non vide dans lequel l'interprétation de e est non vide, 1 sinon.

Annexe C Définitions

C.1 Concrétisation

Définition C.1. Pour $e \in \mathcal{E}$, $\sigma^\sharp \in \Sigma^\sharp$, en notant $\mathbb{E}^\sharp[[e]]\sigma^\sharp = (e^\sharp, (\epsilon, d))$, on a :

$$\gamma(\mathbb{E}^\sharp[[e]]\sigma^\sharp) = \begin{cases} \{ n \in \mathbb{Z} \mid \exists \delta \in (\mathbb{V} \rightarrow \mathbb{Z}), \delta(\epsilon(e)) = n \wedge \delta \in \gamma_{\mathbb{Z}}(d) \} & \text{si } e : \text{int} \\ \gamma_{\mathfrak{t}}(e^\sharp) & \text{sinon} \end{cases}$$

$$\text{en notant } \epsilon(e) = \begin{cases} n & \text{si } e = n \in \mathbb{Z} \\ \epsilon(x) & \text{si } e = x \in \mathbb{V}_{\mathbb{Z}} \\ \epsilon(e_1) \oplus \epsilon(e_2) & \text{sinon} \end{cases}$$

Définition C.2. Pour $\sigma \in \Sigma$, $\sigma^\sharp = (\epsilon, d) \in \Sigma^\sharp$, on note $\sigma \in \sigma^\sharp \iff \forall x \in \mathbb{V} \setminus \mathbb{V}_{\mathbb{Z}}, \sigma(x) \in \gamma(\epsilon(x)) \wedge \sigma|_{\mathbb{Z}} \in \gamma_{\mathbb{Z}}(d)$

Lemme C.1. Si $\sigma \in \sigma^\sharp$, alors $\forall x, \sigma(x) \in \gamma(\sigma^\sharp(x), \sigma^\sharp)$.

Rappelons maintenant que pour $e \in \mathcal{E}$, $\sigma_p \in \mathcal{P}(\Sigma)$, on a $\mathcal{F}[[e]]\sigma_p = \{ \sigma \in \sigma_p \mid \mathbb{E}[[e]]\sigma = n \in \mathbb{Z} \wedge n \neq 0 \}$. La définition de la fonction de filtre \mathcal{F}^\sharp , la fonction de filtre dans l'abstrait, dépend du domaine choisi pour représenter les entiers et est fournie avec celui-ci. Elle vérifie la propriété :

Lemme C.2. Pour $e \in \mathcal{E}$, $\sigma^\sharp \in \Sigma^\sharp$, si $\sigma_p \subseteq \sigma^\sharp$, $\mathcal{F}[[e]]\sigma_p \subseteq \gamma(\mathcal{F}^\sharp[[e^n]]\sigma_0^\sharp)$, avec $\mathbb{E}^\sharp[[e]]\sigma^\sharp = e^n, \sigma_0^\sharp$.

C.2 Union abstraite \cup^\sharp

Soient $(e_1^\sharp, (s_1, d_1)), (e_2^\sharp, (s_2, d_2)) \in \mathcal{V}^\sharp \times \Sigma^\sharp$. Alors :

$$(e_1^\sharp, (s_1, d_1)) \cup^\sharp (e_2^\sharp, (s_2, d_2)) = \begin{cases} (x, (s_1, d_1)[x \rightarrow e_1^\sharp] \cup_{\Sigma^\sharp}^\sharp (s_2, d_2)[x \rightarrow e_2^\sharp]) & \text{si } e_1^n, e_2^n \in \mathcal{E}_{\mathbb{Z}}, \\ & x \text{ fraîche} \\ e_1^\sharp \cup_{\mathcal{D}}^\sharp e_2^\sharp, (s_1, d_1) \cup_{\Sigma^\sharp}^\sharp (s_2, d_2) & \text{sinon} \end{cases}$$

où $\cup_{\Sigma^\sharp}^\sharp$ est définie comme suit :

$$(s_1, d_1) \cup_{\Sigma^\sharp}^\sharp (s_2, d_2) = \begin{cases} x \rightarrow x_i, & \text{si } x \in \mathbb{V}_{\mathbb{Z}}, x_i \text{ fraîche} \\ x \rightarrow s_1(x) \cup_{\mathcal{D}} s_2(x) & \text{sinon} \end{cases} \quad \begin{matrix} v_i = s_1(x) \cup_{\mathbb{Z}}^\sharp s_2(x) \\ , d_0[x_i \rightarrow v_i] \end{matrix}$$

avec $d_0 = d_1 \cup_{\mathcal{D}_{\mathbb{Z}}} d_2$

Lemme C.3. \cup^\sharp est sûr.

C.3 Fonction match^\sharp

On définit ici match^\sharp , utilisé dans la définition de la sémantique abstraite.

$$\begin{aligned} \text{match}^\sharp(\cdot, \cdot, \cdot) : \Sigma^\sharp \times \mathcal{V}^\sharp \times \Pi &\rightarrow \Sigma^\sharp \times \Sigma^\sharp \\ \text{match}^\sharp(\sigma^\sharp, v^\sharp, _) &= \sigma^\sharp, \perp \\ \text{match}^\sharp(\sigma^\sharp, v^\sharp, x \in \mathcal{V}) &= \sigma^\sharp[x \rightarrow v^\sharp], \perp \\ \text{match}^\sharp(\sigma^\sharp, v^\sharp, n_2 \in \mathbb{Z}) &= \mathcal{F}^\sharp[[v^\sharp]]\sigma^\sharp, \mathcal{F}^\sharp[[-v^\sharp]]\sigma^\sharp \\ \text{match}^\sharp(\sigma^\sharp, v^\sharp, C_i(p_1, \dots, p_n)) &= \sigma_n^\sharp, \sigma_{-,n}^\sharp \\ \text{avec } \begin{cases} v^\sharp = (g, \mathcal{C}) \in \mathcal{D}_t \\ C_i \in \mathcal{C}, \sigma_0^\sharp = \sigma^\sharp \\ \forall j \in \llbracket 1, n \rrbracket, \sigma_{j+1}^\sharp, \sigma_{-,j+1}^\sharp = \text{match}^\sharp(\sigma_j^\sharp, g_{i,j}, p_j) \end{cases} \\ \text{match}^\sharp(\sigma^\sharp, v^\sharp, p_1 \text{ when } e_1) &= \mathcal{F}^\sharp[[e_1^n]]\sigma_1^\sharp, \sigma_{-,0}^\sharp \\ \text{avec } \text{match}(\sigma^\sharp, v^\sharp, p) &= \sigma_0^\sharp, \sigma_{-,0}^\sharp, \mathbb{E}^\sharp[[e_1]]\sigma_0^\sharp = e_1^n, \sigma_1^\sharp \end{aligned}$$

Notons que la fonction de filtre \mathcal{F}^\sharp est fournie par le domaine numérique. Étant donnée une expression numérique et un environnement abstrait, elle restreint cet environnement abstrait de sorte que l'expression numérique en paramètre soit non nulle.

Lemme C.4. Pour $\sigma, v, p \in \Sigma \times \mathcal{V} \times \Pi$, $\sigma^\sharp \in \Sigma^\sharp, v^\sharp \in \mathcal{V}^\sharp$ tel que $\sigma \in \sigma^\sharp, v \in \gamma(v^\sharp, \sigma^\sharp)$, alors en notant $\text{match}(\sigma, v, p) = \sigma_m, b$ et $\text{match}^\sharp(\sigma^\sharp, v^\sharp, p) = \sigma_m^\sharp, \sigma_{-,m}^\sharp$, on a $b = \mathbf{true} \implies \sigma_m \in \sigma_m^\sharp$ et $b = \mathbf{false} \implies \sigma_m \in \sigma_{-,m}^\sharp$.

Démonstration. On procède par récurrence sur la syntaxe des motifs. On ne s'intéressera qu'au cas suivant :

- Si $p = C_l(p_1, \dots, p_n)$.
 - Si $b = \mathbf{true}$, alors par définition de match , $v = C_l(v_1, \dots, v_n)$, et comme $v \in \gamma(v^\sharp, \sigma^\sharp) = \gamma_t(v^\sharp, \sigma^\sharp)$, $v_i \in \gamma(g_{i,j}, \sigma^\sharp)$. Alors en notant $\sigma_0 = \sigma$ et pour $i \in \llbracket 1, n \rrbracket$, $\text{match}(\sigma_i, v_i, p_i) = \sigma_{i+1}, b_{i+1}$, et en constatant que $\forall i, b_i = \mathbf{true}$, par une récurrence simple sur $i \in \llbracket 1, n \rrbracket$, on prouve que $\sigma_i \in \sigma_i^\sharp$ (on applique successivement l'hypothèse de récurrence. Alors en particulier,
 - Si $b = \mathbf{false}$, le raisonnement est le même en remplaçant σ_i^\sharp par $\sigma_{-,i}^\sharp$.

Les autres cas sont plus simples, et utilisent la correction de \mathcal{F}^\sharp . □

Annexe D Preuves

D.1 Sûreté de \cup_{τ}^{\sharp} et \cap_{τ}^{\sharp}

Démonstration. Étant donné un type algébrique \mathfrak{t} , on veut prouver que l'opérateur de $\mathcal{D}_{\mathfrak{t}} \cup_{\mathfrak{n}}^{\sharp}$ est une abstraction sûre de \cup , c'est-à-dire :

$$\forall (g_1, \mathcal{C}_1), (g_2, \mathcal{C}_2) \in \mathcal{D}_{\mathfrak{t}}, \gamma_{\mathfrak{t}}((g_1, \mathcal{C}_1)) \cup \gamma_{\mathfrak{t}}((g_2, \mathcal{C}_2)) \subseteq \gamma_{\mathfrak{t}}((g_1, \mathcal{C}_1) \cup_{\tau}^{\sharp} (g_2, \mathcal{C}_2))$$

Rappelons que :

$$(g^1, \mathcal{C}_1) \cup_{\mathfrak{t}}^{\sharp} (g^2, \mathcal{C}_2) = (g, \mathcal{C}_1 \cup \mathcal{C}_2) \quad \text{avec } g_{i,j} = g_{i,j}^1 \cup_{\mathcal{D}_{i,j}} g_{i,j}^2$$

Soit maintenant $x \in \gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1)) \cup \gamma_{\mathfrak{t}}((\mathcal{C}_2, g_2))$. Supposons sans perte de généralité que $x \in \gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1))$. Alors remarquons que $\mathcal{C}_1 \subseteq \mathcal{C}_1 \cup \mathcal{C}_2$ et que $\forall i, j, g_{i,j}^1 \leq_{i,j} g_{i,j}^2$. D'où $(\mathcal{C}_1, g_1) \sqsubseteq (\mathcal{C}_1 \cup \mathcal{C}_2, g)$. Donc par croissance de $\gamma_{\mathfrak{t}}$, on a bien $\gamma_{\mathfrak{t}}((g_1, \mathcal{C}_1)) \cup \gamma_{\mathfrak{t}}((g_2, \mathcal{C}_2)) \subseteq \gamma_{\mathfrak{t}}((g_1, \mathcal{C}_1) \cup_{\tau}^{\sharp} (g_2, \mathcal{C}_2))$.

La preuve de sûreté de l'opérateur \cap_{τ}^{\sharp} est similaire. □

D.2 Sûreté de l'élargissement pour les objets algébriques

Démonstration. Étant donné un type \mathfrak{t} , on cherche à prouver que $\nabla_{\mathfrak{t}}$ est bien un opérateur d'élargissement pour $\mathcal{D}_{\mathfrak{t}}$. Rappelons que :

$$(\mathcal{C}_1, g_1) \nabla_{\mathfrak{t}} (\mathcal{C}_2, g_2) = (g, \mathcal{C}_1 \cup \mathcal{C}_2) \quad \text{avec } g_{i,j} = g_1|_{i,j} \nabla_{\mathcal{D}_{i,j}} g_2|_{i,j}$$

- $\nabla_{\mathfrak{t}}$ sur-approxime l'union. On note $X_{\leq h} = \{x \in X \mid h(x) \leq h\}$ avec $h(x)$ la profondeur de x . On prouve par récurrence sur h $\mathcal{P}(h)$: " $\forall (\mathcal{C}_1, g_1), (\mathcal{C}_2, g_2), \gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1))_{\leq h} \cup \gamma_{\mathfrak{t}}((\mathcal{C}_2, g_2))_{\leq h} \subseteq \gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1) \nabla_{\mathfrak{t}} (\mathcal{C}_2, g_2))_{\leq h}$ ". Pour $h = 0$, les ensembles sont vides ou contiennent uniquement des feuilles (aucun champ récurif) et le raisonnement du cas $\mathfrak{t}i, j \neq \mathfrak{t}$ ci-dessous s'applique.

Soit $h \in \mathbb{N}$, $(g_1, \mathcal{C}_1), (g_2, \mathcal{C}_2) \in \mathcal{D}_{\mathfrak{t}}$. Soit $x \in \gamma_{\mathfrak{t}}(((g_1, \mathcal{C}_1))_{\leq h+1} \cup \gamma_{\mathfrak{t}}((g_2, \mathcal{C}_2))_{\leq h+1})$. Supposons sans perte de généralités que $x \in \gamma_{\mathfrak{t}}(((g_1, \mathcal{C}_1))_{\leq h+1})$.

Alors $x = C_i(x_{i,1}, \dots, x_{i,n_i})$, avec $C_i \in \mathcal{C}_1$, et $h(x) \leq h + 1$.

- Supposons $\mathfrak{t}i, j \neq \mathfrak{t}$. On a $x_{i,j} \in \gamma_{\tau_{i,j}}(g_{1,i,j})$. Comme $\nabla_{\mathcal{D}_{i,j}}$ est un opérateur d'élargissement, on a $\gamma_{\tau_{i,j}}(g_{1,i,j}) \cup_{i,j}^{\sharp} \gamma_{\tau_{i,j}}(g_{2,i,j}) \subseteq \gamma_{\tau_{i,j}}(g_{1,i,j} \nabla_{i,j} g_{2,i,j})$, donc $\gamma_{\tau_{i,j}}(g_{1,i,j}) \subseteq \gamma_{\tau_{i,j}}(g_{1,i,j} \nabla_{i,j} g_{2,i,j})$. On en déduit :

$$x_{i,j} \in \gamma_{\tau_{i,j}}(g_1[i, j] \nabla_{\mathcal{D}_{i,j}} g_2[i, j]) = \gamma_{\tau_{i,j}}(g_{i,j}).$$

- Supposons $\mathfrak{t}i, j = \mathfrak{t}$. On a alors $x_{i,j} \in \gamma_{\mathfrak{t}}(g_1, g_{1,i,j})_{\leq h}$. Par hypothèse de récurrence, on a $\gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1))_{\leq h} \cup \gamma_{\mathfrak{t}}((\mathcal{C}_2, g_2))_{\leq h} \subseteq \gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1) \nabla_{\mathfrak{t}} (\mathcal{C}_2, g_2))_{\leq h}$, d'où $\gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1))_{\leq h} \subseteq \gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1) \nabla_{\mathfrak{t}} (\mathcal{C}_2, g_2))_{\leq h}$, donc $x \in \gamma_{\mathfrak{t}}((g, \mathcal{C}_1 \cup \mathcal{C}_2))_{\leq h}$.

Ainsi, $\mathcal{P}(h+1)$ est vrai. Cela prouve donc la propriété pour tout $h \in \mathbb{N}$. Alors on a bien $\nabla_{\mathfrak{t}}$ sur-approxime l'union : $\gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1)) \cup \gamma_{\mathfrak{t}}((\mathcal{C}_2, g_2)) \subseteq \gamma_{\mathfrak{t}}((\mathcal{C}_1, g_1) \nabla_{\mathfrak{t}} (\mathcal{C}_2, g_2))$

- $\nabla_{\mathfrak{t}}$ assure la convergence en temps fini. Considérons la suite $((g_n, \mathcal{C}_n))_{n \in \mathbb{N}}$. On cherche à prouver que la suite

$$((g_n^\nabla, C_n^\nabla))_{n \in \mathbb{N}} : \begin{cases} (g_0^\nabla, C_0^\nabla) = (g_0, \mathcal{C}_0) \\ (g_{n+1}^\nabla, C_{n+1}^\nabla) = (g_n^\nabla, C_n^\nabla) \nabla_{\mathfrak{t}}(g_{n+1}, \mathcal{C}_{n+1}) \end{cases}$$

est stationnaire.

Pour i, j , on remarque que $g_n^\nabla|_{i,j} = \nabla_{i,j}^{1 \leq l \leq n} g_l|_{i,j}$. Or $\nabla_{i,j}$ est un opérateur d'élargissement, donc cette suite est stationnaire. Soit $n_{i,j}$ l'indice à partir duquel cette suite est stationnaire. Notons $n_g = \max_{i,j} n_{i,j}$. Alors à partir du rang n_g , toutes les suites $(g_n^\nabla|_{i,j})_{n \in \mathbb{N}}$ sont stationnaires. Cela implique que la suite $(g_n^\nabla)_{n \in \mathbb{N}}$ est stationnaire à partir du rang n_g . Enfin, \cup est un opérateur d'élargissement dans $\mathcal{P}(\mathbb{C}_{\mathfrak{t}})$ donc il existe un rang n_C à partir duquel $(\mathcal{C}_n^\nabla)_{n \in \mathbb{N}}$ est stationnaire. Ainsi, à partir du rang $N = \max(n_g, n_C)$, $((g_n^\nabla, C_n^\nabla))_{n \in \mathbb{N}}$ est stationnaire.

Donc $\nabla_{\mathfrak{t}}$ est bien un widening.

□

D.3 Sûreté de la sémantique abstraite

Démonstration. On cherche à prouver la sûreté de la sémantique abstraite \mathbb{S}^\sharp , c'est-à-dire

$$\forall \sigma \in \Sigma, \forall e \in \mathcal{E}, \sigma \in \sigma^\sharp \implies \mathbb{E}[e]\sigma \in \gamma(\mathbb{E}^\sharp[e]\sigma^\sharp) \wedge \sigma \in \sigma_0^\sharp \quad \text{avec } \mathbb{E}^\sharp[e]\sigma^\sharp = e^\sharp, \sigma_0^\sharp$$

On effectue la preuve par récurrence sur la syntaxe de $e \in \mathcal{E}$.

- Soit e une expression telle qu'une sous-expression e_0 renvoie une erreur. Par hypothèse sur la sémantique abstraite, on a aussi $\mathbb{E}^\sharp[e]\sigma^\sharp = \Omega$. On suppose l'expression bien typée, donc les seules erreurs proviennent des échecs de filtrage et des échecs d'assertion. Par hypothèse de récurrence, $\mathbb{E}[e_0]\sigma \in \gamma(\mathbb{E}^\sharp[e_0]\sigma^\sharp) = \gamma(\Omega) = \{\omega\}$, l'interprétation de e_0 renvoie une erreur, donc l'interprétation concrète de e renvoie une erreur. On a donc bien $\mathbb{E}[e]\sigma \in \gamma(\mathbb{E}^\sharp[e]\sigma^\sharp)$, et $\sigma \in \sigma_0^\sharp = \sigma^\sharp$. Par la suite, on supposera donc qu'aucune sous-expression ne renvoie d'erreur.
- Si $e = x$, alors $\mathbb{E}[e]\sigma = \sigma(x)$, et comme $\sigma \in \sigma^\sharp$, on a $\sigma(x) \in \gamma(\sigma^\sharp(x), \sigma^\sharp) = \gamma(\mathbb{E}^\sharp[x]\sigma^\sharp)$ et $\sigma \in \sigma_0^\sharp = \sigma^\sharp$.
- Si $e = z \in \mathbb{Z}$, alors $\mathbb{E}[e]\sigma = z$. On a $\mathbb{E}^\sharp[z \in \mathbb{Z}](\epsilon, d) = z, (\epsilon, d)$. Or, $\gamma(n, (\epsilon, d)) = \{ n \in \mathbb{Z} \mid \exists \delta \in \mathcal{P}(\mathbb{V} \rightarrow \mathbb{Z}), \delta(z) = n \wedge \delta \in \gamma_{\mathbb{Z}}(d_0) \} = \{z\}$. D'où $\mathbb{E}[z \in \mathbb{Z}]\sigma \in \gamma(\mathbb{E}^\sharp[z \in \mathbb{Z}]\sigma^\sharp)$ et $\sigma \in \sigma_0^\sharp = \sigma^\sharp$.

- Si $e = e_1 \oplus e_2$, on a $\mathbb{E}^\sharp[[e_1 \oplus e_2]]\sigma^\sharp = x \oplus y, \sigma_2^\sharp[x \rightarrow e_1^n][y \rightarrow e_2^n] = e^\sharp, \sigma^\sharp$, avec

$$\begin{cases} \mathbb{E}^\sharp[[e_1]]\sigma^\sharp = e_1^n, \sigma_1^\sharp \\ \mathbb{E}^\sharp[[e_2]]\sigma_1^\sharp = e_2^n, (\epsilon_2, d_2) \end{cases}$$
 Alors $\gamma(\mathbb{E}^\sharp[[e_1 \oplus e_2]]\sigma^\sharp) = \{n \in \mathbb{Z} \mid \exists \delta \in \mathcal{P}(\mathbb{X}^n), \delta(v_x) \oplus \delta(v_y) = n \wedge \delta \in \gamma_{\mathbb{Z}}(d)\}$.
 Par hypothèse de récurrence, $\begin{cases} \mathbb{E}[[e_1]]\sigma \in \gamma(\mathbb{E}^\sharp[[e_1]]\sigma^\sharp) \wedge \sigma \in \sigma_1^\sharp \\ \mathbb{E}[[e_2]]\sigma \in \gamma(\mathbb{E}^\sharp[[e_2]]\sigma^\sharp) \wedge \sigma \in \sigma_2^\sharp \end{cases}$.
 On en déduit que $\sigma|_{\mathbb{Z}} \in \gamma_{\mathbb{Z}}(d_2)$. Enfin, par correction de l'assignation, on a $\sigma' = \sigma|_{\mathbb{Z}}[v_x \rightarrow \sigma(e_1^n)][v_y \rightarrow \sigma(e_2^n)] \in \sigma_2^\sharp[x \rightarrow e_1^n][y \rightarrow e_2^n] = (\epsilon, d)$. Donc $\sigma' \in \gamma_{\mathbb{Z}}(d)$, $\sigma' \in \mathbb{V} \rightarrow \mathbb{Z}$ et $\sigma'(v_x \oplus v_y) = \sigma'(v_x) \oplus \sigma'(v_y) = \sigma'(x) + \sigma'(y) = \mathbb{E}[[e_1]]\sigma \oplus \mathbb{E}[[e_2]]\sigma$. D'où $\mathbb{E}[[e_1 \oplus e_2]]\sigma = \mathbb{E}[[e_1]]\sigma \oplus \mathbb{E}[[e_2]]\sigma \in \mathbb{E}^\sharp[[e_1 \oplus e_2]]\sigma^\sharp$. Enfin, comme $\forall i, \sigma \in \sigma_i^\sharp$, $\sigma \in \sigma_0^\sharp \subseteq \sigma_0^\sharp[r.i.j \rightarrow h_{i,j}]$. Enfin, $\sigma \leq \sigma' \in \sigma_0^\sharp$.
- Si $e = C_l(e_1, \dots, e_n)$, on a $\mathbb{E}[[e]]\sigma = C_l(\mathbb{E}[[e_1]]\sigma, \dots, \mathbb{E}[[e_n]]\sigma)$ (puisque l'expression est bien typée par hypothèse). Rappelons que :

$$\mathbb{E}^\sharp[[C_l(e_1, \dots, e_n)]]\sigma^\sharp = (g, \{C_l\}), \sigma_0^\sharp[r.i.j \rightarrow h_{i,j}]$$

avec
$$\begin{cases} \mathbb{E}^\sharp[[e_i]]\sigma^\sharp = v_i, \sigma_i^\sharp, (s, \mathcal{C}) = \bigcup_{e_i: \mathbf{t}}^\sharp v_i, \sigma_0^\sharp = \bigsqcup_i^\sharp \sigma_i^\sharp \\ h_{i,j} = \begin{cases} s_{l,j} \cup_{\mathcal{D}_{l,j}} v_j & \text{si } i = l \wedge \neg(\tau_{l,j} : \mathbf{t}) \\ s_{i,j} & \text{sinon} \end{cases} \\ g_{i,j} = \begin{cases} r.i.j & \text{si } \tau_{i,j} : \mathbf{int} \\ h_{i,j} & \text{sinon} \end{cases} \end{cases}$$

On a $C_l \in \{C_l\}$. Remarquons que si $\mathbf{t}i, j = \mathbf{int}$, on a $\gamma(g_{i,j}, \sigma^\sharp) = \gamma(r.i.j, \sigma^\sharp) = \gamma(h_{i,j}, \sigma^\sharp)$ par correction de l'assignation, on peut sans perte de généralités considérer $g_{i,j} = h_{i,j}$ dans la suite. Procédons maintenant par disjonction de cas.

— Soit $j \in \llbracket 1, n_l \rrbracket$ tels que $\mathbf{t}1, j \neq \mathbf{t}$. Par hypothèse de récurrence, on sait que $\mathbb{E}[[e_j]]\sigma \in \gamma(\mathbb{E}^\sharp[[e_j]]\sigma^\sharp)$. Or, on sait que $g_{l,j} = s_{l,j} \cup_{\mathcal{D}_{l,j}} \mathbb{E}[[e_j]]\sigma$, d'où $g_{i,j} \subseteq \mathbb{E}^\sharp[[e_j]]\sigma^\sharp$, d'où par croissance de $\gamma_{\mathbf{t}}$, $\gamma_{\mathbf{t}}(g_{i,j}) \subseteq \gamma_{\mathbf{t}}(\mathbb{E}^\sharp[[e_j]]\sigma^\sharp)$. On obtient donc $\mathbb{E}[[e_j]]\sigma \in \gamma_{l,j}(g_{l,j})$.

— Soit $i \in \llbracket 1, n_l \rrbracket$ tels que $\mathbf{t}i, j = \mathbf{t}$. On a toujours $\mathbb{E}[[e_j]]\sigma \in \gamma(\mathbb{E}^\sharp[[e_j]]\sigma^\sharp)$. Notons $(\mathcal{C}_j, s_j) = \mathbb{E}^\sharp[[e_j]]\sigma^\sharp$. Remarquons alors que $s_j \sqsubseteq s \sqsubseteq g$. On a alors $(\mathcal{C}_j, s_j) \sqsubseteq (g_{i,j}, g)$, donc par croissance de $\gamma_{\mathbf{t}}$, $\gamma_{\mathbf{t}}((\mathcal{C}_j, s_j)) \subseteq \gamma_{\mathbf{t}}((g_{i,j}, g))$, c'est-à-dire $\gamma(\mathbb{E}^\sharp[[e_j]]\sigma^\sharp) \subseteq \gamma_{\mathbf{t}}((g_{i,j}, g))$, d'où $\mathbb{E}[[e_j]]\sigma \in \gamma_{\mathbf{t}}((g_{i,j}, g))$.

On a donc bien $C_l(\mathbb{E}[[e_1]]\sigma, \dots, \mathbb{E}[[e_n]]\sigma) \in \gamma(\mathbb{E}^\sharp[[e]]\sigma^\sharp)$. De plus, $\forall i, \sigma \in \sigma_i^\sharp \subseteq \sigma_0^\sharp \subseteq \sigma_0^\sharp[r.i.j \rightarrow h_{i,j}]$

- Si $e = \text{if } e_1 \text{ then } e_2 \text{ else } e_3$, on a :

$$\mathbb{E}[[\text{if } e_1 \text{ then } e_2 \text{ else } e_3]]\sigma = \begin{cases} \mathbb{E}[[e_2]]\sigma & \text{si } \mathbb{E}[[e_1]]\sigma = 0 \\ \mathbb{E}[[e_3]]\sigma & \text{si } \mathbb{E}[[e_1]]\sigma = z \in \mathbb{Z} \wedge z \neq 0 \end{cases}$$

Par disjonction de cas :

- Supposons $\mathbb{E}[[e_1]]\sigma = n \in \mathbb{Z}$ et $n \neq 0$, notons $\mathbb{E}^\sharp[[e_1]]\sigma^\sharp = e^n, \sigma_1^\sharp$. Alors par le lemme C.2, $\sigma \in \mathcal{F}[[e]]\{\sigma\} \subseteq \gamma(\mathcal{F}^\sharp[[e^n]]\sigma_1^\sharp)$. Donc par hypothèse de récurrence, en notant $\mathbb{E}^\sharp[[e_2]](\mathcal{F}^\sharp[[e^n]]\sigma_1^\sharp) = e_2^\sharp, \sigma_2^\sharp$ et $\mathbb{E}[[e_2]]\sigma \in \gamma(e_2^\sharp, \sigma_2^\sharp)$, d'où par croissance de γ , $\mathbb{E}[[e_2]]\sigma \in \gamma(\mathbb{E}^\sharp[\text{if } e_1 \text{ then } e_2 \text{ else } e_3])\sigma^\sharp$, et par définition de \cup^\sharp , $\sigma \in \sigma_2^\sharp \subseteq \sigma_0^\sharp$
- Si $\mathbb{E}[[e_i]] = 0$, en remplaçant e_1 par $\neg e_1$, le raisonnement est identique.

- Le cas du assert est identique à celui du if.
- Si $e = \text{match } e_0 \text{ with } p_1 \rightarrow e_1 \mid \dots \mid p_m \rightarrow e_m$, alors notons $\text{match}(\sigma, \mathbb{E}[[e_0]]\sigma, p_1) = (\sigma', b_{\text{guard}})$ et $\text{match}^\sharp(\sigma^\sharp, \mathbb{E}^\sharp[[e_0]]\sigma^\sharp, p) = \sigma_m^\sharp, \sigma_{\neg, m}^\sharp$. Procédons par disjonction de cas :
 - Si $b_{\text{guard}} = \mathbf{true}$, alors $\mathbb{E}[[e]]\sigma = \mathbb{E}[[e_1]]\sigma'$. Or, d'après le lemme C.4, $\sigma' \in \sigma_m^\sharp$. Donc $\mathbb{E}[[e_1]]\sigma' \in \mathbb{E}^\sharp[[e_1]]\sigma_m^\sharp$ par hypothèse de récurrence, et comme $\sigma_m^\sharp \neq \emptyset$ (il contient σ), on a $\mathbb{E}^\sharp[[e_1]]\sigma_m^\sharp \subseteq \mathbb{E}^\sharp[[e_1]]\sigma_m^\sharp \cup^\sharp \mathbb{E}^\sharp[[e_1']]\sigma_{\neg, m}^\sharp = \mathbb{E}^\sharp[[e]]\sigma^\sharp$, on a bien $\mathbb{E}[[e]]\sigma \in \gamma(\mathbb{E}^\sharp[[e]]\sigma^\sharp)$. Toujours par définition de \cup^\sharp , on a bien $\sigma \in \sigma_0^\sharp$.
 - Sinon, si $b_{\text{guard}} = \mathbf{false}$ et $m > 1$, alors par le lemme C.4, $\sigma' \in \sigma_{\neg, m}^\sharp$. Alors par hypothèse de récurrence, en notant $e_1' = \text{match } e_0 \text{ with } p_2 \rightarrow e_2 \mid \dots \mid p_m \rightarrow e_m$, on a $\mathbb{E}[[e_1']]\sigma \in \gamma(\mathbb{E}^\sharp[[e_1']]\sigma^\sharp)$ et comme $m > 1$, on a bien : $\mathbb{E}^\sharp[[e_1']]\sigma^\sharp \subseteq \mathbb{E}^\sharp[[e_1]]\sigma_m^\sharp \cup^\sharp \mathbb{E}^\sharp[[e_1']]\sigma_{\neg, m}^\sharp = \mathbb{E}^\sharp[[e]]\sigma^\sharp$. De même, $\sigma \in \sigma_0^\sharp$.
 - Enfin, si $b_{\text{guard}} = \mathbf{false}$ et $m = 1$, alors $\sigma' \in \sigma_{\neg, 1}^\sharp$, donc $\sigma_{\neg, 1}^\sharp \neq \emptyset$, donc on appelle récursivement l'analyse avec $m = 0$ d'où $\mathbb{E}^\sharp[[e]]\sigma^\sharp \subseteq \Omega$ donc on a bien $\omega \in \Omega$.
- Si $e = \text{let } x = e_1 \text{ in } e_2$, alors en notant $\mathbb{E}^\sharp[[e_1]]\sigma^\sharp = e^\sharp, \sigma^\sharp$, puisque $\mathbb{E}[[e_1]]\sigma \in \gamma(\mathbb{E}^\sharp[[e_1]]\sigma^\sharp)$ et $\sigma \in \sigma^\sharp$, on a $\sigma[x \rightarrow \mathbb{E}[[e_1]]\sigma] \in \sigma^\sharp[x \rightarrow e^\sharp]$ par correction de l'assignation, donc par hypothèse de récurrence, $\mathbb{E}[[e_2]]\sigma[x \rightarrow \mathbb{E}[[e_1]]\sigma] \in \gamma(\mathbb{E}^\sharp[[e_2]]\sigma^\sharp[x \rightarrow e^\sharp])$ et $\sigma[x \rightarrow \mathbb{E}[[e_1]]\sigma] \subseteq \sigma^\sharp[x \rightarrow e^\sharp]$, d'où $\mathbb{E}[[e]]\sigma \in \mathbb{E}^\sharp[[e]]\sigma^\sharp$ et $\sigma \in \sigma_0^\sharp$.
- Si $e = e_0 \dots e_n$. Comme e est bien typé, $\mathbb{E}^\sharp[[e_0]]\sigma^\sharp = (x_1, \dots, x_n, e^\sharp, \sigma_0^\sharp)$. Notons que par hypothèse de récurrence et par construction de l'abstraction de fun, $\sigma \in \sigma_0^\sharp$. Notons $\mathbb{E}^\sharp[[e_i]]\sigma^\sharp = e_i^\sharp, \sigma_i^\sharp$. Par hypothèse de récurrence, on a également $\forall i, \mathbb{E}[[e_i]]\sigma \in \gamma(e_i^\sharp, \sigma_i^\sharp)$. Or, par hypothèse de récurrence, $f = \mathbb{E}[[e_0]]\sigma \in \gamma(\mathbb{E}^\sharp[[e_0]]\sigma^\sharp)$, par définition de la concrétisation pour les abstractions de fonction, on en conclut que $\mathbb{E}[[e]] = f(\mathbb{E}[[e_1]]\sigma, \dots, \mathbb{E}[[e_n]]\sigma) \in \gamma(e^\sharp, \sigma_0^\sharp[x_i = e_i^\sharp]) = \gamma(\mathbb{E}^\sharp[[e]]\sigma^\sharp)$. Comme $x_i \notin \sigma$, on a également $x_i \in \sigma_0^\sharp[x_i = e_i^\sharp]$.
- Si $e = \text{fun } x_1 \dots x_n \rightarrow e_0$. On a $\mathbb{E}^\sharp[[e]]\sigma^\sharp = (x_1, \dots, x_n, \mathbb{E}^\sharp[[e_0]]\sigma^\sharp[x_1 \rightarrow \top, \dots, x_n \rightarrow$

\top), σ^\sharp) et $\mathbb{E}[e]\sigma = \lambda a_1 \dots a_n. \mathbb{E}[e_0]\sigma[x_i \rightarrow a_i]$. Soit $a_1 : \tau_1, \dots, a_n : \tau_n$ et $v_1 \in \mathcal{D}_1, \dots, v_n \in \mathcal{D}_n$ tels que $a_i \in \gamma(v_i, \sigma_i^\sharp)$. On a $f(a_1, \dots, a_n) = \mathbb{E}[e_0]\sigma[x_i \rightarrow a_i]$. On note $\mathbb{E}^\sharp[e_0]\sigma^\sharp[x_i \rightarrow \top] = e^\sharp, \sigma_1^\sharp$. Remarquons que $\sigma[x_i \rightarrow a_i] \in \sigma^\sharp[x_i \rightarrow \top]$. Alors par hypothèse de récurrence $\mathbb{E}[e_0]\sigma[x_i \rightarrow a_i] \in \gamma(\mathbb{E}^\sharp[e_0]\sigma^\sharp[x_i \rightarrow \top]) = \gamma(e^\sharp, \sigma_1^\sharp)$ et $\sigma[x_i \rightarrow a_i] \in \sigma_1^\sharp$, que $a_i \in \gamma(v_i, \sigma_i^\sharp)$, on a $\sigma[x_i \rightarrow a_i] \in \sigma_1^\sharp[x_i = v_i]$, donc on conclut que $\mathbb{E}[e_0]\sigma[x_i \rightarrow a_i] \in \gamma(e^\sharp, \sigma_1^\sharp[x_i = v_i])$. On obtient donc bien $f \in \gamma(\mathbb{E}^\sharp[e]\sigma^\sharp)$. On a $\sigma \in \sigma_0^\sharp = \sigma^\sharp$.

- Si $e = \text{let rec } x_f = e_1 \text{ in } e_2$. On a :

$$\mathbb{E}[\text{let rec } x_f = e_1 \text{ in } e_2]\sigma = \begin{cases} \mathbb{E}[e_2]\sigma[x_f \rightarrow \text{lfp}(F_{x_f, x, g})] & \text{si } \mathbb{E}[e_1]\sigma = \lambda x_1 \dots x_n. g \\ \omega & \text{sinon} \end{cases}$$

$$\text{avec } F_{x_f, x, g}(\phi) = \lambda y. \mathbb{E}[g]\sigma[x \rightarrow y, x_f \rightarrow \phi]$$

On sait que $\text{lfp}(F_{x_f, x, g})$ vaut $\bigcup_{n \in \mathbb{N}} (F_{x_f, x, g}^n(\perp))$. Remarquons qu'on peut écrire $\lambda y. \mathbb{E}[g]\sigma[x \rightarrow y, x_f \rightarrow \phi] = \mathbb{E}[e_1]\sigma[x_f \rightarrow \phi]$. Alors étant donné v , par hypothèse de récurrence, pour $v \in \gamma(v^\sharp)$, on a $F_{x_f, x, g}(v) = \mathbb{E}[e_1]\sigma[x_f \rightarrow v] \in \gamma(\mathbb{E}^\sharp[e_1]\sigma^\sharp[x_f \rightarrow v^\sharp]) = \gamma(F^\sharp(v^\sharp))$.

En conséquence, en remarquant que $(x_1, \dots, x_n, \perp, \sigma^\sharp[x_1, \dots, x_n]) = \perp$ dans le domaine des fonctions, on a bien $\bigcup_{n \in \mathbb{N}} (F_{x_f, x, g}^n(\perp)) \in \gamma(\bigcup_{n \in \mathbb{N}} ((F^\sharp)^n(\perp)))$ par sûreté de \bigcup^\sharp . Comme ∇ est un opérateur d'élargissement, donc cette opération converge et sur-approxime \bigcup^\sharp , on a donc $\text{lfp}(F_{x_f, x, g}) \in \gamma(\nabla_{n \in \mathbb{N}}^\sharp((F^\sharp)^n(\perp)))$.

D'où $\sigma[x_f \rightarrow \text{lfp}(F_{x_f, x, g})] \in \sigma^\sharp[x_f \rightarrow \nabla_{n \in \mathbb{N}}^\sharp((F^\sharp)^n(\perp))]$. Le raisonnement qui suit est similaire à celui du `let`. □

Annexe E Correspondance de Galois

Définition E.1. Si $\forall i, j, \mathcal{T}_{\mathbf{ti}, \mathbf{j}} \xrightarrow[\alpha_{i,j}]{\gamma_{\tau_{i,j}}} \mathcal{D}_{i,j}$, on peut définir α récursivement de la manière suivante :

$$\alpha(\mathbf{Ci}(x_{i,1}, \dots, x_{i,n_i})) = \left(\prod_{\substack{1 \leq i' \leq n \\ 1 \leq j \leq n_i}} \begin{cases} \alpha_{i,j}(x_{i,j}) & \text{si } i' = i \wedge \mathbf{ti}, \mathbf{j} \neq \mathbf{t} \\ \perp & \text{sinon} \end{cases} \times \{\mathbf{Ci}\} \right) \bigcup_{\mathbf{ti}, \mathbf{j} \neq \mathbf{t}}^\sharp \alpha(x_{i,j})$$

On a alors la correspondance de Galois $\mathcal{T}_{\mathbf{t}} \xrightarrow[\alpha]{\gamma} \mathcal{D}_{\mathbf{t}}$.

Annexe F Listings des benchmarks

```
let x = Cons(1, Cons(2, Cons(3, Nil))) in
let y = Nil in
let z = Cons(4, x) in
```

x : { Cons }	x _{1,1} : [1, 3]	x _{1,2} : { Nil, Cons }
y : { Nil }		
z : { Cons }	y _{1,1} : [1, 4]	z _{1,2} : { Nil, Cons }

Listing 1: list.ml

```
type tree = Node of tree * int * tree | Leaf of int
let x = Node(Node(Leaf(250), 100, Leaf(251)), 1, Leaf(252)) in
```

x: { Node }	x _{1,1} : { Node, Leaf }	x _{1,2} : [1, 100]	x _{1,3} {Leaf}
	x _{2,1} : [250,252]		

Listing 2: tree.ml

```
let x = match Cons(1, Nil) with
| Cons(h,q) -> h
;;
assert (x = 1)
```

x : [1,1]

Listing 3: match.ml

```
let x = match Cons(1,Cons(2, Nil)) with
| Cons(h,q) -> h
;;
assert (x < 3)
```

x : [1,2]

Listing 4: match2.ml

```
let x = match Cons(1,Cons(2, Nil)) with
| Cons(h,q) -> h
;;
assert (x = 1)
```

```
x : [1,2]
Warning : assertion failure
```

Listing 5: match_alarm.ml

L'abstraction de x est identique, mais renvoie cette fois : **Warning: assertion failure**. En effet, comme $x : [1,2]$, si $x=1$, il s'agit d'une fausse alarme - c'est effectivement le cas ici.

```
let x = match Cons(1,Cons(2, Nil)) with
| Cons(h,q) -> h
;;
assert (x = 3)
```

```
x : [1,2]
Error : assertion failure
```

Listing 6: match_error.ml

L'abstraction de x est identique, mais renvoie cette fois : **Error: assertion failure**. En effet, puisque l'intersection entre $[1,2]$ et $[3,3]$ est vide, il s'agit effectivement d'une erreur.

```
let f x y = x + y ;;
let z = f 1 2 ;;
assert (z = 3)
```

```
z : [3,3]
```

Listing 7: add.ml

Vérification symbolique de protocoles cryptographiques en F^* : application au sous-protocole TreeSync de MLS

Théophile Wallez

Inria Paris

Résumé

TreeSync est un sous-protocole d'authentification pour MLS, un protocole de messagerie de groupe sécurisé en cours de standardisation à l'IETF. Nous formalisons MLS en F^* , et présentons un théorème de sécurité pour TreeSync à l'aide du cadre de cryptographie symbolique DY^* . Afin de comprendre les hypothèses clés qui permettent d'exprimer un tel théorème, nous présentons en détail le fonctionnement interne de DY^* . Au cours de notre analyse, nous avons proposé plusieurs changements à TreeSync qui ont été intégrés à MLS, et avons développé de nouvelles méthodes de preuves pour passer à l'échelle avec DY^* afin de pouvoir s'attaquer à un gros protocole comme MLS, par exemple, un mécanisme permettant à la fois l'exécution concrète et symbolique du protocole, la génération automatique de parseurs et sérialiseurs binaires vérifiés, ou encore la génération d'invariants globaux du protocole à partir d'invariants locaux.

1 Introduction

Messagerie de groupe sécurisée. Que ce soit WhatsApp, Signal, Facebook Messenger ou Wire, toutes les applications de messagerie moderne exposent le chiffrement de bout-en-bout comme l'une de leurs fonctionnalités phare, ce qui confirme que la communication privée et sécurisée est maintenant un aspect important pour les utilisateurs. Contrairement aux connexions HTTPS de courte durée, les conversations peuvent s'étaler sur des années, donc les garanties de sécurité des messageries doivent prendre en compte la possibilité réaliste qu'un des appareils se fasse voler ou compromettre pendant la durée de vie de la conversation. Si un adversaire compromet un appareil, il pourra bien sûr lire les messages récents et envoyer de nouveaux messages, mais nous voulons protéger les messages envoyés dans le passé lointain (*confidentialité persistante*), ainsi que les messages envoyés ou reçus dans le futur après une période de guérison (*sécurité après compromission*).

Entre deux personnes, ces messageries utilisent le protocole Signal [22] qui possède toutes les propriétés de sécurité requises. Les conversations de groupes sont fondamentalement différentes : des participants peuvent entrer et partir de la conversation à n'importe quel moment, il faut donc qu'un participant enlevé d'un groupe ne puisse plus lire les messages envoyés après, propriété que l'on peut obtenir grâce à la sécurité après compromission. Les protocoles actuellement utilisés pour les conversations de groupe ayant la propriété de sécurité après compromission dépendent de n^2 canaux Signal (un pour chaque paire de participants), ce qui n'est pas efficace.

Messaging Layer Security (MLS). Le groupe de travail IETF MLS a pour but de concevoir un protocole de messagerie de groupe efficace ayant les propriétés de confidentialité persistante et de sécurité après compromission [6]. MLS peut se découper en trois sous-protocoles : TreeKEM se charge de calculer un secret partagé entre tous les membres d'un groupe à un instant donné, TreeDEM se charge de chiffrer les messages à partir du secret de groupe, et TreeSync se charge d'authentifier l'appartenance au groupe. L'authentification est un composant crucial : avant de dire que les communications sont confidentielles, il faut d'abord savoir entre qui elles pourraient être déclarées comme telles.

Preuve de protocole assistée par ordinateur. Nous avons pour objectif de faire une preuve formelle de sécurité pour MLS, c'est-à-dire une preuve vérifiée avec un assistant de preuve ou un outil dédié. Plusieurs méthodologies existent. Les preuves de sécurité *calculatoires*, en utilisant des outils comme Squirrel [5], EasyCrypt [7] ou CryptoVerif [12], utilisent des jeux cryptographiques pour borner la probabilité qu'un attaquant puisse casser un protocole cryptographique. Les preuves de sécurité *symboliques*, en utilisant des outils comme ProVerif [13], Tamarin [25] ou DY* [9], supposent que les primitives cryptographiques sont parfaites.

Nous choisissons ici d'utiliser le modèle symbolique, parce que les preuves sont moins complexes qu'avec le modèle calculatoire, ce qui permet d'étudier des protocoles gros et complexes comme MLS.

Contributions. Nous présentons un théorème de sécurité pour TreeSync prouvé dans le modèle symbolique en F* [31] à l'aide du cadriciel DY* [9], sous des hypothèses raisonnables sur TreeKEM et TreeDEM. Durant notre analyse, nous avons trouvé une attaque exploitant les interactions entre TreeSync et TreeDEM, et montré que les garanties de TreeSync n'étaient pas aussi fortes qu'initialement espérées. Nous avons proposé des modifications du protocole résolvant ces problèmes, qui sont depuis intégrées à MLS. Notre spécification est précise à l'octet près et est interopérable avec d'autres implémentations. La description complète du protocole et de sa preuve font l'objet d'un autre article [38].

Afin de bien comprendre la signification d'un tel théorème, nous expliquons en détail le fonctionnement interne de DY*, afin de saisir quelles sont les hypothèses clés qui font tenir l'édifice. Au cours de notre analyse, nous avons développé de nouveaux outils et méthodes de preuves afin de passer à l'échelle et accomplir la preuve d'un protocole de cette taille en DY*.

Sommaire. Nous commençons par une introduction à la preuve de protocole symbolique (§2) avant d'expliquer comment DY* permet de réaliser de telles preuves (§3), puis montrons les nouvelles méthodes de preuve que nous avons développées (§4) pour étudier le protocole TreeSync et décrivons l'impact qu'a eu notre travail sur MLS (§5), et concluons avec les travaux connexes (§6).

2 Analyse symbolique de protocoles

2.1 Modélisation de l'attaquant

L'attaquant. On se place dans le pire cas et on suppose que le monde entier cherche à casser notre protocole. Si le protocole est sécurisé malgré tous ces efforts, alors il sera aussi sécurisé dans un cas plus réaliste. Ainsi, on considère que l'attaquant représente tout ce qui est extérieur aux participants honnêtes du protocole, qu'il a un contrôle total du réseau et peut donc lire tous les messages envoyés, les modifier ou décider qu'ils ne soient pas livrés, envoyer des messages en se faisant passer pour d'autres personnes, et enfin faire des manipulations cryptographiques avec les données qu'il connaît.

Cryptographie symbolique. L'analyse symbolique de protocole, à la Dolev-Yao [18], abstrait les primitives cryptographiques en supposant qu'elles sont parfaites : par exemple, une fonction de hachage n'est pas inversible et est sans collision, ou encore le chiffrement symétrique d'un message à l'aide d'une clé ne peut se déchiffrer qu'à partir de la même clé.

Dans des outils d'analyse symbolique comme Tamarin [25], Proverif [13] ou DY* (§3.1), cela se modélise via des constructeurs et des règles de réduction : par exemple le chiffrement symétrique se modélise avec le constructeur `SymEnc(cle, msg)` et se détruit avec la règle de

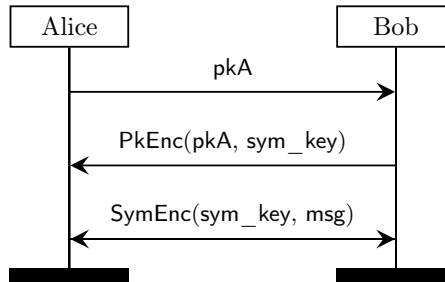


FIGURE 1 – Un premier essai pour MiniTLS. $\text{PkEnc}(pk, \text{msg})$ est le chiffrement de msg pour la clé publique de chiffrement pk , et $\text{SymEnc}(\text{key}, \text{msg})$ est le chiffrement de msg pour la clé symétrique key .

réduction $\text{SymDec}(\text{cle}, \text{SymEnc}(\text{cle}, \text{msg})) = \text{msg}$. Sans autre règle de réduction, cela capture bien l'idée que $\text{SymEnc}(\text{cle}, \text{msg})$ ne peut se déchiffrer qu'avec la clé cle .

2.2 Cas d'étude : MiniTLS

Un protocole utilisé au quotidien est celui qui sécurise les connexions entre les navigateurs web et les sites web : il se nomme TLS. Essayons de construire un protocole ayant un cas d'usage similaire, que nous appellerons MiniTLS malgré les différences flagrantes qu'il a avec TLS : en effet, TLS dispose de nombreuses fonctionnalités dont nous ne nous soucierons pas ici ; MiniTLS est un protocole jouet qui n'a pas vocation à être utilisé dans la vie courante. Afin de rester dans la tradition, on appellera le navigateur web Alice et le site web Bob. Mallory essayera par la suite d'écouter la conversation.

Le chiffrement asymétrique étant en général moins efficace que le chiffrement symétrique, on l'utilise pour se mettre d'accord sur une clé symétrique (sym_key), qui est ensuite utilisée pour échanger les messages. Une première tentative basée sur ce principe est décrite Figure 1 : Alice envoie une clé publique de chiffrement à Bob, celui-ci génère sym_key , la chiffre avec la clé publique de chiffrement d'Alice et la lui envoie. Alice et Bob peuvent ensuite communiquer en chiffrant symétriquement avec sym_key .

Ce protocole n'est pas sécurisé. En effet, Mallory (un attaquant Dolev-Yao) peut effectuer une attaque dite de l'« homme du milieu » pour obtenir le message confidentiel, comme décrit dans la Figure 2 : Mallory intercepte pkA , envoie sa propre clé pkM à Bob. Ensuite Mallory déchiffre le message de Bob et récupère sym_key avant de le chiffrer pour Alice. Mallory peut donc lire tous les messages msg par la suite.

Pour réparer ce protocole et obtenir la version finale de MiniTLS décrite dans la Figure 3, on ajoute une signature de la part de Bob (que Mallory ne peut pas falsifier), qu'Alice vérifie par la suite avec la clé publique de vérification de signature de Bob.

Un problème subsiste : comment Alice peut-elle faire le lien entre l'identité de Bob et la clé de publique de vérification de signature de Bob ? Ce lien est crucial, sinon Alice pourrait utiliser la clé publique de vérification de signature de Mallory à la place de celle de Bob sans le savoir, et une attaque très similaire à celle décrite Figure 2 serait possible.

Ce problème ne se résout usuellement pas à l'aide de la cryptographie. Par exemple, HTTPS résout ce problème en le déléguant à un tiers de confiance [30, C.2] : les autorités de certifications sont responsables de faire le lien entre identité et clé publique, et fournissent des certificats que

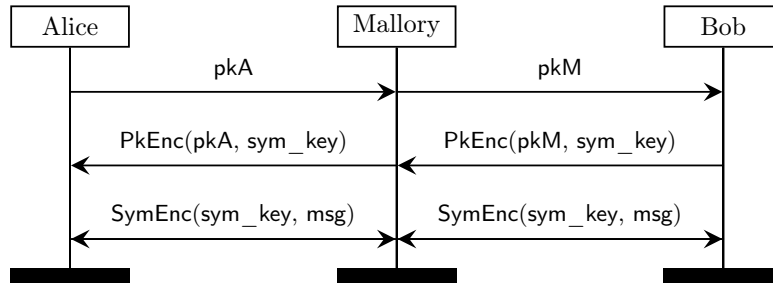


FIGURE 2 – Une attaque sur le premier essai de MiniTLS décrit dans la Figure 1. L’attaquant Dolev-Yao nommé « Mallory » obtient sym_key et peut déchiffrer tous les messages suivants.

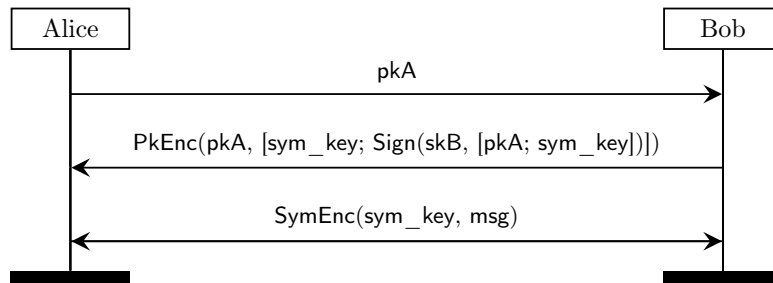


FIGURE 3 – Une version corrigée et sécurisée de MiniTLS. Une signature est rajoutée par rapport au premier essai décrit dans la Figure 1. $\text{Sign}(sk, \text{msg})$ est la signature de msg avec la clé de signature sk , et $[a; b]$ représente une concaténation non-ambigue de a et b .

les navigateurs peuvent vérifier. Les navigateurs possèdent une liste d’autorités de certification dignes de confiance, et rejettent les certificats n’émanant pas de cette liste. Signal propose une autre solution à ce problème [23, 4.1] et les utilisateurs doivent vérifier leur identité via un autre canal. C’est un problème qu’on ne traitera pas dans cette étude de cas.

Sécurité de MiniTLS. Une première chose à remarquer avant d’énoncer un théorème de sécurité sur MiniTLS, est qu’Alice sait qui est Bob, mais Bob ne sait pas qui est Alice. Ainsi on ne peut pas garantir qu’un message envoyé par Bob est confidentiel entre Alice et Bob, on ne pourra que prouver qu’il est confidentiel entre Bob et les personnes connaissant la clé privée associée à pkA .

Le théorème de sécurité que nous allons énoncer est informel, et nous ne couvrirons que dans les grandes lignes sa preuve, informelle elle aussi. Le théorème sera plus tard formalisé précisément en F^* à l’aide du cadriciel DY^* (§3.4), et la preuve formelle sera similaire à la preuve informelle ci-dessous.

Au cours de la preuve, nous allons parfois dire qu’une donnée « ne peut être connue que de A et B », cela veut dire que par la suite nous vérifierons que cette donnée ne peut pas fuiter et être connue (par exemple) de C.

Théorème. *Pour chaque message msg transféré dans la dernière phase du protocole, si, du point de vue d’Alice, msg ne peut être connu que d’Alice et Bob, ou du point de vue de Bob, msg ne peut être connu que de Bob et des personnes connaissant la clé privée associée à pkA , alors*

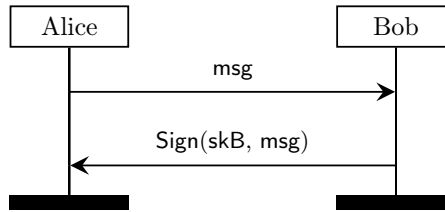


FIGURE 4 – Protocole SigneTout.

le chiffrement de msg avec sym_key et l'envoi sur le réseau comme fait dans la dernière phase du protocole ne fait pas fuiter d'information.

Esquisse de preuve de la sécurité de msg . Il suffit de prouver que sym_key , du point de vue d'Alice, ne peut être connu que d'Alice et Bob, ou du point de vue de Bob, ne peut être connu que de Bob et des personnes connaissant la clé privée associée à pkA . En effet, pour des messages msg vérifiant les conditions ci-dessus, chiffrer msg avec sym_key et le diffuser sur le réseau implique que msg sera divulgué aux personnes connaissant sym_key , ce qui correspond aux personnes qui ont le droit de connaître msg donc cela ne fait pas fuiter d'information. \square

Esquisse de preuve de la sécurité de sym_key . Prouvons maintenant la propriété pour sym_key .

Alice commence par générer une paire de clés de chiffrement asymétrique, la clé privée ne pouvant être connue que d'elle-même et la clé publique étant publique. Alice envoie la clé publique pkA sur le réseau, ce qui ne fait pas fuiter d'informations puisqu'elle est publique.

Bob reçoit pkA , une clé publique de chiffrement. Bob génère sym_key avec l'intention que cela ne puisse être connu que de lui-même et des personnes connaissant la clé privée associée à pkA . Bob effectue sa signature sur $[pkA; sym_key]$, et chiffre avec la clé publique de chiffrement pkA un message contenant sym_key ainsi que la signature, avant de l'envoyer sur le réseau. Divulguer ce chiffrement au réseau de fait pas fuiter d'informations : seules les personnes connaissant la clé privée associée à pkA peuvent déchiffrer le message, et le contenu du message pouvant en effet être connu de ces personnes.

Alice reçoit le message, déchiffre et vérifie la signature. Puisque la clé de signature de Bob skB n'est connue que de Bob, Alice sait que c'est bien Bob qui a fait la signature et pas un attaquant. Lorsque Bob a fait la signature, la propriété suivante était vraie (c'est un invariant pour toutes les signatures du protocole) : sym_key peut être connue exactement par Bob et par les personnes connaissant la clé privée associée à pkA . Alice sait qu'elle seule connaît la clé privée associée à pkA , elle en déduit donc que sym_key n'est connu que d'elle-même et Bob. \square

Un argument récurrent au cours de la preuve est qu'on ne chiffre un message que lorsque celui-ci est moins secret que la clé qui permet déchiffrer le message (que ce soit dans le cas symétrique ou asymétrique). De plus, la preuve repose sur un invariant sur les signatures : l'invariant est vrai lorsque Bob effectue la signature, plus tard quand Alice vérifie la signature elle sait que l'invariant est vrai. Ces techniques de preuves seront plus tard utilisées en F* (§3.2).

2.3 Modularité des protocoles

Nous étudions la sécurité de MLS en le décomposant en trois sous-protocoles (TreeSync, TreeKEM et TreeDEM). Cependant, la sécurité des protocoles cryptographiques ne se compose pas bien.

En effet, considérons le protocole `SigneTout` décrit Figure 4, qui signe n'importe quel message qu'on lui envoie. Si en parallèle de `MiniTLS`, nous exécutons le protocole `SigneTout` avec la même clé de signature, alors `MiniTLS` n'est plus sécurisé : en effet la signature dans le deuxième message de `MiniTLS` est obtainable par Mallory en utilisant le protocole `SigneTout` avec `msg = [pkA; sym_key]`. Ainsi, l'attaque de l'homme du milieu décrite dans la Figure 2 est à nouveau réalisable.

Une façon de composer les protocoles est de prouver qu'ils n'interfèrent pas entre eux : par exemple s'ils utilisent des clés de signature différentes, où s'ils signent des ensembles de messages disjoints. Cette dernière possibilité peut être réalisée en incluant un tag dans le message signé : par exemple `MiniTLS` signera le message `["MiniTLS"; pkA; sym_key]` et `SigneTout` signera le message `["SigneTout"; msg]`, rendant ainsi toute signature réalisée dans un des protocoles inutilisable dans l'autre protocole.

Nous avons trouvé une attaque exploitant l'interférence entre les signatures de `TreeSync` et `TreeDEM`, que nous avons corrigée en ajoutant un tag de façon systématique dans toutes les signatures de `MLS` (§5).

3 Preuve symbolique de protocoles cryptographiques avec DY*

Nous allons décrire ici le fonctionnement interne de `DY*` [9], un cadriciel de vérification de protocole contre un attaquant Dolev-Yao (§2.1), écrit en F* [31].

Les analyseurs de protocole symbolique comme `ProVerif` [13] ou `Tamarin` [25] sont complètement automatisés : à partir d'une description de protocole, ils prouvent la sécurité du protocole dans le modèle symbolique ou exhibent une attaque, ceci sans interaction supplémentaire de la part de l'utilisateur. Les preuves en `DY*` sont beaucoup plus manuelles, ce qui en contrepartie donne certains avantages : si les analyseurs automatiques brillent sur des protocoles avec quelques échanges de message (comme `TLS` [10]), `DY*` peut s'attaquer à des protocoles beaucoup plus gros, comme une preuve de `Signal` avec un nombre non borné de messages [9, §4], ou encore `MLS` (nombre non-borné de messages et de participants), l'objectif de cet article. De plus, la façon dont les preuves sont faites en `DY*` est compréhensible par un humain, et ressemble dans les grandes lignes à la preuve de `MiniTLS` (§2.2).

Le fonctionnement décrit par la suite est simplifié : `DY*` permet de modéliser un attaquant qui compromet dynamiquement l'état (`secret`) des participants du protocole au cours du temps, ainsi la plupart des prédicats de `DY*` sont paramétrés par le temps. Nous omettons ceci dans la description qui suit.

3.1 Bytes symboliques et pouvoir de l'attaquant

Constructeurs. La modélisation des primitives cryptographiques se fait à l'aide de constructeurs et de règles de réduction (§2.1). Nous modélisons les suites d'octets à l'aide d'un type inductif appelé `bytes`, où les constructeurs de `bytes` correspondent aux constructeurs des primitives cryptographiques.

En F*, cela ressemble à :

```
type bytes =
| PublicVal: public_bytes → bytes // Donnée publique
| Rand: len:nat → timestamp → label → bytes // Aléatoire
| Concat: lhs:bytes → rhs:bytes → bytes // Concaténation
```

```

| Hash: bytes → bytes // Fonction de hachage
| Pk: sk:bytes → bytes // Clé publique de chiffrement à partir de clé privée de déchiffrement
| PkEnc: pk:bytes → msg:bytes → bytes // Chiffrement à clé publique
| SymEnc: key:bytes → msg:bytes → bytes // Chiffrement symétrique
| Vk: sk:bytes → bytes // Clé publique de vérification à partir de clé privée de signature
| Sign: sk:bytes → msg:bytes → bytes // Signature d'un message
| ... // Insérer ici tous les autres constructeurs à considérer (MAC, KDF, DH, ...)

val length: bytes → nat // On dispose d'une fonction de longueur

```

Le type `bytes` modélise des suites d'octets que l'on peut concaténer, hacher, signer, chiffrer symétriquement et asymétriquement.

Les suites d'octets publiques sont représentées par le constructeur `PublicVal`, et les suites d'octets aléatoires fraîchement générées sont représentées par le constructeur `Rand`. Les suites d'octets aléatoires possèdent une longueur afin de pouvoir définir la fonction `length` dessus. La fraîcheur est modélisée par le `timestamp`, ce champ n'étant pas choisi par l'utilisateur de `DY*` : ce dernier ne peut construire des données aléatoires qu'à travers un effet (décrit dans §3.3) qui donne la garantie que tous les `timestamp` sont différents. Les personnes ayant le droit de connaître une donnée aléatoire doivent être choisies dès sa génération : c'est ce à quoi correspond le type `label` dans le constructeur `Rand`, qui sera décrit plus tard dans cette section.

En pratique, le type `bytes` est totalement opaque pour le code utilisant `DY*`. Ainsi les constructeurs sont cachés derrière des fonctions :

```

let concat (lhs:bytes) (rhs:bytes): bytes = Concat lhs rhs
let hash (msg:bytes): bytes = Hash bytes
// ...

```

Un utilisateur de `DY*` ne peut donc pas inspecter la façon dont un `bytes` a été construit, à part à travers les règles de réduction.

Règles de réduction. Les règles de réduction sont implémentées à l'aide de fonctions en `F*` qui opèrent sur le type `bytes`. Par exemple, un déchiffrement asymétrique réussit uniquement lorsque la suite d'octets est un chiffrement asymétrique avec une clé publique associée à la bonne clé privée.

```

let pk_dec (sk:bytes) (msg_chiffre:bytes): option bytes =
  match msg_chiffre with
  | PkEnc (Pk sk') msg_clair →
    if sk = sk' then Some msg_clair
    else None // Pas la bonne clé
  | _ → None // Pas un chiffré

```

De la même manière, on peut séparer une suite d'octets en deux uniquement lorsque la suite d'octets est une concaténation, et que l'indice auquel la séparation s'effectue se situe exactement entre les deux suites d'octets concaténées.

```

let split (i:nat) (msg:bytes): options bytes =
  match msg with
  | Concat lhs rhs →
    if i = length lhs then
      Some (lhs, rhs)
    else None // Pas le bon indice
  | _ → None // Pas une concaténation

```

Pouvoir de l'attaquant. On modélise le pouvoir de l'attaquant Dolev-Yao (§2.1) avec le prédicat `attacker_knows` : l'attaquant connaît les valeurs publiques, les messages envoyés sur le réseau, et peut appliquer les fonctions cryptographiques sur les valeurs déjà connues.

La modélisation de la connaissance des messages envoyés sur le réseau se fait via un effet (décrit dans §3.3). Comme cet effet modélise un état croissant (un journal d'évènements, stockant entre-autres les messages envoyés sur le réseau), nous pouvons utiliser les fonctionnalités de témoin [1] de F*, et ainsi définir le prédicat `msg_sent_on_network` comme témoin de l'existence du message dans le journal.

```
let attacker_knows (b:bytes): prop =
  (msg_sent_on_network b) ∨
  (∃ public_bytes. b == PublicVal public_bytes) ∨
  (∃ b1 b2. attacker_knows b1 ∧ attacker_knows b2 ∧ b == concat b1 b2) ∨
  (∃ pk msg. attacker_knows pk ∧ attacker_knows msg ∧ b == pk_enc pk msg) ∨
  (∃ sk msg. attacker_knows sk ∧ attacker_knows msg ∧ Some b == pk_dec sk msg) ∨
  ...
```

Ce prédicat est crucial dans la modélisation du pouvoir de l'attaquant, et fait partie de la base de confiance de DY*. On ne peut pas prouver que rien n'a été oublié lors de sa définition, mais on peut s'en convaincre, par exemple en instanciant la classe de type permettant de faire des opérations cryptographiques (§4.2) sur le type `bytes` raffiné avec la propriété `attacker_knows`. Si une propriété manquait dans le `attacker_knows`, alors cela se remarquerait lors de la définition de la fonction associé dans la classe de type.

Labels. De la même façon qu'il est courant de renforcer un théorème pour pouvoir le prouver par induction, le prédicat `attacker_knows` est trop peu précis pour faire des preuves de sécurité. Nous introduisons un treillis, permettant de décrire l'ensemble des personnes connaissant une donnée. Les labels permettent de décrire des choses comme « cette donnée n'est connue que d'Alice et Bob », ou encore « cette donnée n'est connue que de Bob et des personnes qui connaissent la clé privée associée à une clé publique », comme nous l'avons fait dans la preuve de sécurité de MiniTLS (§2.2).

Les labels formant un treillis, on peut comparer deux labels et dire qu'un label est moins sécurisé qu'un autre : toute personne pouvant connaître les données associées au label plus sécurisé peut connaître les données associées au label moins sécurisé. Cela permet de modéliser le fait que quelqu'un, par exemple Alice, a le droit de connaître une certaine donnée, en disant que le label de la donnée est moins sécurisé que le label « connu que d'Alice ». Ce treillis n'est pas total, par exemple le label « connu que d'Alice » et le label « connu que de Bob » ne sont pas comparables.

```
// Opérations de treillis
val (<#): l1:label → l2:label → prop // Relation d'ordre : l1 est moins secret que l2
val union: label → label → label // Borne inférieure
val intersection: label → label → label // Borne supérieure

val public: label // Donnée connue par tout le monde : c'est un minimum global du treillis
val readers: list identity → label // Donnée connue que par la liste d'identités
```

Pour chaque suite d'octets, on peut obtenir son label :

```
let get_label (b:bytes): label =
  match b with
  | PublicVal _ → public
```

```

| Rand len lab → lab
//...
| Hash b → get_label b
| Pk sk → public
| PkEnc sk msg → public
| SymEnc sk msg → public
//...

```

Lien entre attaquant et label. On prouve le théorème suivant : si une donnée est associée à un label secret, alors l'attaquant ne peut pas connaître cette donnée.

```

val attacker_dont_know_secret_values:
  b:bytes → identities:list identity →
  Lemma (get_label b == readers identities ⇒ ¬(attacker_knows b))

```

Ce théorème se déduit à partir de deux théorèmes plus bas niveau. Le premier dit que si un attaquant connaît une donnée, alors le label associé à cette donnée est moins sécurisée que `public`, et le second dit que le label `readers identities` n'est pas moins sécurisé que le label `public`.

3.2 Propriété clé des messages circulant sur le réseau

Dans les analyseurs de protocole symbolique automatiques comme ProVerif ou Tamarin, l'analyseur dispose de la spécification complète du protocole, et connaît donc toutes les utilisations de la cryptographie faites par les personnes honnêtes : ils peuvent raisonner sur le protocole dans sa globalité. Par exemple, ils peuvent raisonner sur l'ensemble des signatures effectuées dans un protocole, et prouver qu'elles n'interfèrent pas entre elles. Ce n'est pas le cas dans un assistant de preuve générique comme F*, qui ne peut raisonner qu'à une échelle plus locale, sur un ensemble fixé de fonctions effectuant de la cryptographie. C'est en fait un problème similaire à la composition de la sécurité des protocoles (§2.3), si nous prouvons qu'une certaine fonction fait des opérations sécurisées avec une clé de signature, rien ne nous empêche d'écrire après-coup une autre fonction qui signe n'importe quel message.

Invariant clé des messages circulant sur le réseau. Nous imposons donc que les participants honnêtes d'un protocole obéissent à certaines règles. Par exemple, ils ne peuvent envoyer sur le réseau que des données publiques (c'est-à-dire, dont le label est moins sécurisé que `public`). De plus, on impose une certaine discipline sur les opérations cryptographiques exécutées : par exemple, on autorise à chiffrer un message avec une clé uniquement lorsque le message est moins secret que la clé. Cela empêche par exemple le cas où Alice chiffrerait une donnée connue que par Alice et Bob avec une clé connue de Charlie, sinon Charlie pourrait déchiffrer ces données confidentielles et apprendre des choses qu'il n'est pas censé savoir. Cet argument est utilisé plusieurs fois dans la preuve de sécurité de MiniTLS (§2.2).

Toutes ces règles de discipline que les participants honnêtes doivent respecter sont encodées dans un prédicat dit de *validité*, et on impose aux participants honnêtes de n'envoyer sur le réseau que des messages publics et valides. En contrepartie, lorsque un participant reçoit un message sur le réseau, ce dernier sait que ce message est valide et est publique.

Validité des suites d'octets. Le prédicat de validité est au cœur des preuves de sécurité en DY*. Il doit prendre en compte les utilisations honnêtes de la cryptographie (comme brièvement décrit ci-dessus), mais aussi les utilisations malhonnêtes. En effet, puisqu'un attaquant peut envoyer des messages sur le réseau, si nous voulons la garantie que les messages reçus sur le réseau sont valides, il faut donc que les messages construits par l'attaquant soient valides.

Voici par exemple la règle de validité dans le cas du chiffrement symétrique :

```
let is_valid ... (b:bytes): prop =
  match b with
  ...
  | SymEnc key msg →
    is_valid key ∧ is_valid msg ∧ ( // key et msg sont récursivement valides
      ((get_label msg) <# (get_label key)) ∨ // cas honnête : msg est moins secret que key
      // cas malhonnête : b est construit par l'attaquant, qui connaît donc key et msg
      ((get_label key) <# public ∧ (get_label msg) <# public)
    )
  ...
```

Concernant les signatures, il n'y a pas de façon évidente de restreindre leur utilisation de façon générique comme nous le faisons avec les chiffrements. Chaque protocole définit un prédicat de signature, qui doit être vrai pour toutes les suites d'octets signées. Ce prédicat de signature fait partie d'une liste de prédicats globaux, spécifique au protocole. Ces prédicats sont au cœur de la preuve, c'est à partir d'eux que découlent toutes les propriétés de sécurité que l'on peut prouver, comme nous l'avons vu avec MiniTLS (§2.2).

```
let is_valid (preds:global_predicates) (b:bytes): prop =
  match b with
  ...
  | Sign sk msg →
    // sign sk msg est valide lorsque
    is_valid sk ∧ is_valid msg ∧ ( // sk et msg sont récursivement valides
      (preds.sign_pred (Vk sk) msg) ∨ // cas honnête : le prédicat est vrai
      // cas malhonnête : b est construit par l'attaquant, qui connaît donc sk et msg
      ((get_label sk) <# public ∧ (get_label msg) <# public)
    )
  ...
```

Quand une signature est valide, cela permet de savoir que soit la clé privée de signature associée à la clé publique de vérification est connue de l'attaquant, soit le prédicat de signature est vrai sur le message signé.

```
val verify_lemma:
  preds:global_predicates →
  verification_key:bytes → msg:bytes → signature:bytes →
  Lemma
  (requires
    // si tous les arguments sont valides
    is_valid preds verification_key ∧ is_valid preds msg ∧ is_valid preds signature ∧
    // et que la signature est valide
    verify verification_key msg signature
  )
  (ensures
    ( // alors soit elle a été construite par un participant honnête et le prédicat de signature est vrai
      preds.sign_pred verification_key msg
    ) ∨ ( // soit la signature a été construite par l'attaquant
      (get_label msg <# public) ∧
      // (get_signkey_label récupère le label de la clé de signature associée à verification_key)
      (get_signkey_label verification_key) <# public
    )
  )
```

)

3.3 Modélisation des effets de bord avec un effet de journal

Les protocoles cryptographiques ont besoin de faire des opérations impures : envoi et réception de messages sur le réseau, stockage d'état, génération de données aléatoires.

Cette impureté est encodée dans un journal d'effets de bord, qui grandit au fur et à mesure que les participants du protocole ont des effets de bord. Ainsi, les messages envoyés sur le réseau sont ajoutés au journal, et les messages sont par la suite réceptionnés en regardant parmi les messages envoyés dans le journal. D'une façon similaire, un participant qui veut stocker son état l'ajoute au journal, et pourra par la suite récupérer son état en le cherchant dans le journal. Chaque génération de données aléatoires est ajoutée dans le journal afin de permettre de garantir leur fraîcheur, comme discuté dans la description du constructeur `Rand` du type `bytes` (§3.1).

De la même façon que les protocoles définissent des invariants de signature que les participants honnêtes se doivent de respecter (§3.2), les protocoles peuvent définir des invariants sur le journal. Par exemple on peut imposer un certain prédicat à être vrai sur tous les états que l'on stocke, en contrepartie lorsque l'on récupère un état, on sait que le prédicat est vrai.

Ce journal d'effets de bord est modifié avec une monade d'état monotone encodée comme un effet F^* [29]. Comme le journal ne fait que grandir, si à un instant donné on a la propriété « un certain effet de bord existe dans le journal », alors cette propriété restera vraie lorsque le journal grandira. Cela permet d'utiliser les fonctionnalités de témoin de F^* [1] et de définir des prédicats comme `msg_sent_on_network` (§3.1).

3.4 Application au protocole MiniTLS

Nous utilisons DY^* pour prouver la sécurité de MiniTLS (§2.2), et allons présenter une petite partie de la preuve de sécurité, à savoir les garanties qui sont offertes par la signature.

Le message à signer est représenté par le type `respond_handshake_tbs`. Un parseur et sérialiseur, prouvés inverses l'un de l'autre, sont automatiquement générés à l'aide de la bibliothèque `Comparse` (§4.3).

```
type respond_handshake_tbs = {
  pk_a: bytes;
  sym_key: bytes;
}

// génère le parseur et sérialiseur automatiquement avec un métaprogramme de Comparse
%splice (gen_parser ('respond_handshake_tbs))
```

Le code de vérification de signature s'écrit alors de la façon suivante :

```
let verify_respond_handshake (verif_key:bytes) (pk_a:bytes) (sym_key:bytes) (signature:bytes): bool =
  let tbs = {
    pk_a = pk_a;
    sym_key = sym_key;
  } in
  verify verific_key (serialize tbs) signature
```

Nous écrivons ensuite l'invariant de signature : on ne signe que des sérialisations de `respond_handshake_tbs`, de plus son champ `sym_key` n'est connu que des personnes connaissant

la clé privée de déchiffrement associée à `pk_a` ainsi que des personnes connaissant la clé privée de signature associée à clé publique de vérification de signature `vk`.

```
let sign_pred (vk:bytes) (msg:bytes) =
  match parse msg with
  | None → ⊥
  | Some tbs → get_label tbs.sym_key == union (get_sk_label tbs.pk_a) (get_signkey_label vk)
```

On peut alors prouver un lemme de sécurité donnant les garanties de la vérification de signature : si la signature est bien authentique et provient de Bob, et que toutes les suites d'octets que l'on considère vérifient les invariants des messages qui passent sur le réseau (§3.2), alors on peut en déduire le label de `sym_key`.

```
val verify_respond_handshake_lemma:
  a:identity → b:identity →
  verif_key:bytes → pk_a:bytes → sym_key:bytes → signature:bytes →
  Lemma
  (requires
    verify_respond_handshake verif_key pk_a sym_key signature ∧ // la signature est authentique
    // verif_key provient de Bob et toutes les suites d'octets obéissent aux invariants du réseau (§3.2)
    get_signkey_label verif_key == readers [b] ∧ is_valid preds verif_key ∧
    is_valid preds pk_a ∧ is_valid preds sym_key ∧ is_valid preds signature
  )
  // alors on en déduit le label de sym_key
  (ensures get_label sym_key == union (get_sk_label pk_a) (readers [b]))
```

En combinant de tels lemmes de sécurité pour toutes les sous-fonctions du protocole, et en modélisant l'exécution complète du protocole avec ses effets de bord (§3.3), nous obtenons une preuve de sécurité pour MiniTLS avec DY*.

4 Techniques de preuve modulaires pour DY*

Maintenant que nous avons vu les outils fournis par DY* pour exprimer des théorèmes de sécurité sur un protocole cryptographique, nous pouvons nous intéresser à la façon précise dont nous allons spécifier le protocole, énoncer le théorème, et effectuer les preuves. Pour cela, il y a de multiples choix techniques à effectuer, et chacun de ces choix aura un impact sur la façon dont seront structurées les preuves. L'ensemble de ces choix techniques constitue donc une méthode qui permet de s'attaquer à la preuve de sécurité d'un protocole. Même si n'importe quelle méthode permet de prouver des petits protocoles avec suffisamment de travail, il convient d'adopter une méthode adéquate afin de s'attaquer à un protocole conséquent comme MLS.

4.1 Idioms utilisés dans les exemples DY*

Les exemples fournis avec DY* ont tous une façon idiomatique d'être définis, qui fonctionne bien sur des protocoles de taille similaire à MiniTLS, et que nous documentons dans cette section. Il nous était difficile d'envisager d'utiliser ces idiomes pour un protocole aussi gros et complexe que MLS, nous présenterons alors d'autres techniques que nous avons utilisées pour prouver TreeSync (§5), un sous-protocole de MLS dédié à l'authentification.

Exécution symbolique et exécution concrète. DY* est une bibliothèque de cryptographie symbolique pour faire des preuves de sécurité, qui n'a donc pas vocation à s'exécuter sur des

suites d'octets concrètes. Cependant elle supporte l'exécution concrète des fonctions cryptographiques d'une façon détournée : la définition du type des suites d'octets et l'implémentation des opérations dessus sont cachées derrière un fichier interface, ce qui fait qu'on peut choisir, à la compilation, d'utiliser une implémentation symbolique des suites d'octets, ou d'utiliser une implémentation concrète qui encapsule HACL* [39], une bibliothèque d'implémentations de primitives cryptographiques vérifiées avec F*. Cette façon de faire implique que ce choix entre exécution concrète et symbolique se fait à la compilation, et implique que les théorèmes vrais seulement dans le monde symbolique (par exemple l'injectivité des fonctions de hachage) doivent être admis dans le cas concret. De plus, nous n'avons pas le choix sur les primitives cryptographiques choisies. Nous résolvons ce problème en utilisant des classes de types (§4.2), que nousinstancions séparément avec DY* comme bibliothèque de cryptographie symbolique et HACL* comme bibliothèque de cryptographie concrète.

Parseurs et sérialiseurs pour les messages et l'état. Les messages envoyés sur le réseau ainsi que l'état stocké ont le type `bytes`, une première étape consiste donc à définir des convertisseurs entre `bytes` et des types plus haut niveau, communément appelés parseurs et sérialiseurs. Les parseurs et sérialiseurs sont écrits à la main, avec des preuves manuelles d'inversion (e.g., sérialiser puis parser donne l'objet de départ). Ceci est possible parce qu'il y a peu de structures en jeu (en général, un type pour le message et un type pour l'état), et parce que le protocole est un modèle qui n'a pas vocation à être interopérable : le format des messages peut être choisi de façon à simplifier les preuves. La spécification du protocole MLS [6] possède plus de cinquante structures binaires, qui doivent toutes être parsés et sérialisés d'une façon précise. Nous résolvons ce problème en développant une nouvelle bibliothèque de parseurs et sérialiseurs binaires vérifiés en F*, appelé `Comparse` (§4.3).

Style intrinsèque. Afin de respecter la discipline que les participants honnêtes doivent suivre (§3.2), une interface à la bibliothèque cryptographique raffinée est mise à disposition par DY*. Dans cette interface, avant de chiffrer un message avec une clé symétrique, il faut prouver que le message est moins secret que la clé. Ou encore, avant d'effectuer une signature, il faut prouver que l'invariant de signature est vrai sur le contenu signé. Cela a pour conséquence que la définition du protocole est mélangée avec les preuves de sécurité, ce qui pose plusieurs problèmes : cela rend la définition du protocole moins lisible (et donc auditable) ; ces obligations de preuve n'ont pas lieu d'être dans une exécution concrète du protocole ; et cela fait que toutes les preuves de sécurité d'une fonction doivent être faites simultanément. Nous utilisons un style extrinsèque qui consiste à totalement séparer code et preuves, ce qui permet de mieux modulariser les preuves et garder une définition du protocole lisible et indépendante des preuves (§4.4).

Des prédicats de sécurité globaux. Les différents prédicats de sécurité décrivant les invariants, comme le prédicat de signature (§3.2) ou les prédicats d'état (§3.3), sont définis de façon globale et sont référencés explicitement tout le long de la preuve de sécurité. Cela ne fonctionne pas pour prouver les protocoles de façon modulaire comme nous voulons le faire sur MLS : si un sous-protocole est prouvé en DY* avec ses prédicats définis de façon globale, et que par la suite nous voulons prouver un nouveau sous-protocole qui s'exécutera en parallèle, dans ce cas il faudrait modifier les prédicats globaux, ce qui risquerait certainement de casser les preuves du premier sous-protocole. Nous résolvons ce problème en définissant des prédicats globaux à partir de multiples prédicats locaux (§4.5).

4.2 Classes de types pour les suites d'octets

Exécution concrète et symbolique. Nous paramétrons le protocole par une classe de type sur les suites d'octets, d'une façon similaire à ce qu'on pourrait faire avec une `Section` en Coq, ou un foncteur en OCaml. Cette classe de type est construite en deux étages. Une première classe de type `bytes_like` permet de faire des opérations non-cryptographiques sur les suites d'octets, comme la concaténation ou le calcul de longueur. Une seconde classe de type `crypto`, dépendant d'une instance de `bytes_like` permet d'effectuer des opérations cryptographiques sur les suites d'octets.

Conséquences techniques. D'une part, à partir d'une unique description du protocole, l'instance de la classe de type permet de choisir entre exécution concrète (pour réellement exécuter le protocole) et exécution symbolique (pour faire des preuves de sécurité), d'autre part, plusieurs instances concrètes permettent de choisir les primitives cryptographiques utilisées par le protocole. De plus, les preuves de sécurité se font sur l'instance symbolique de la classe de type, on peut donc utiliser des propriétés vraies seulement dans les preuves symboliques, comme l'injectivité des fonctions de hachages, sans avoir à les admettre sur les instances concrètes.

Importance de l'exécution concrète. Pouvoir exécuter l'instance concrète du protocole permet de tester que la spécification du protocole est interopérable avec d'autres implémentations, et ainsi s'assurer que les preuves sont faites sur la spécification précise du protocole, jusqu'à la façon dont les données haut-niveau sont sérialisées en suites d'octets, et pas seulement sur un modèle approximatif. Ainsi les théorèmes de sécurité ne sont pas déconnectés de la réalité : l'interopérabilité de la spécification du protocole renforce la confiance dans le fait que celle-ci est une implémentation correcte du standard étudié. Cette façon de faire a par exemple permis de trouver l'attaque exploitant l'ambiguïté des signatures dans MLS (§5), qui ne peut se trouver en faisant la preuve que lorsque celle-ci raisonne sur la façon dont les contenus des différentes signatures sont sérialisés en suites d'octets.

4.3 Compare : bibliothèque pour parseurs et sérialiseurs vérifiés

Nous avons développé une nouvelle bibliothèque appelée « Compare » qui permet d'obtenir des parseurs et sérialiseurs binaires vérifiés, interopérables, adaptés à la preuve symbolique, le tout de façon automatique.

Fonctionnalités de Compare. Les parseurs et sérialiseurs produits par Compare sont prouvés inverses l'un de l'autre. Compare fournit des parseurs et sérialiseurs pour des types de base, et un métaprogramme écrit en Meta-F* [24] permet de générer des parseurs et sérialiseurs pour des types algébriques à partir de leur définition, lorsqu'ils sont construits à partir de types déjà connus de Compare. De plus, des annotations permettent de contrôler la façon dont le type est sérialisé afin d'être conforme à un format binaire précis. Nous arrivons ainsi à définir automatiquement des parseurs et sérialiseurs vérifiés pour de nombreux types, dont les plus de cinquante décrits dans la définition du protocole MLS [6].

Suites d'octets concrètes et symboliques. La classe de type `bytes_like` axiomatise de façon minimale les suites d'octets : existence d'une suite d'octets vide, d'une longueur, d'une concaténation, d'une séparation, et de conversions depuis et vers les entiers de taille bornée, ainsi que de lemmes de compatibilité des différentes opérations. Par exemple, on ne suppose pas l'existence de lemme sur l'associativité de la concaténation, ni même de l'unicité de la suite d'octets de taille nulle ! En effet, ces propriétés ne sont pas vérifiées sur les `bytes` symboliques de DY^* (§3.1).

Propriétés de sécurité. Des théorèmes sur les combinateurs et les types de base permettent de prouver des théorèmes utiles à la preuve symbolique, par exemple si un type algébrique ne contient que des données secrètes pour un label donné, alors sa sérialisation est elle aussi secrète pour ce label, et inversement.

Fonctionnement interne de Comparse. Les parseurs et sérialiseurs sont générés à l'aide d'un type intermédiaire qui se prête mieux à la composition, sur lequel nous disposons de quelques instances pour des types de base, ainsi que de quelques combinateurs. Un premier combinateur crée une instance pour une paire dépendante, un second combinateur crée une instance pour un type isomorphe à un autre type qui possède une instance. Ainsi, si un type est isomorphe à un embriquement de paire dépendantes de type de base (ce qui est le cas des types algébriques) alors il est possible de définir un parseur et sérialiseur pour ce type.

4.4 Utilisation du style extrinsèque

Dans la preuve de TreeSync, nous séparons la définition du protocole, les preuves de théorèmes vrais quelle que soit l'instance des suites d'octets, les preuves de théorèmes vrais sur l'instance symbolique des suites d'octets, et la gestion de l'état interne du protocole. Ce style a été montré dans la preuve de MiniTLS en DY* (§3.4).

Ce style est plus verbeux, mais a des avantages : la définition du protocole n'est pas entrelacée avec des obligations de preuve ce qui permet de préserver sa lisibilité, de plus les différentes propriétés de sécurité peuvent être prouvées séparément, augmentant ainsi la lisibilité des preuves.

4.5 Prédicats globaux à partir de prédicats locaux

Prédicats globaux dans DY*. Nous avons vu (§3.2 et §3.3) que les preuves en DY* reposent sur des prédicats globaux, par exemple sur les signatures ou sur l'état. Ces prédicats doivent être prouvés lors de la signature ou lorsque l'on met à jour l'état, en contrepartie on obtient que le prédicat est vrai lorsque l'on vérifie une signature ou que l'on récupère l'état. Étudions plus spécifiquement les signatures ; la gestion de l'état possède un problème similaire qui se traite aussi de façon similaire.

Le prédicat de signature est un prédicat sur le message signé, ainsi que la clé publique de vérification de signature utilisée (comme vu dans §3.2). Une difficulté arrive lorsqu'il existe plusieurs types de signatures dans le protocole. Le prédicat de signature doit disposer d'un moyen de différencier les différents types de signatures (comme discuté dans §2.3), que ce soit en raisonnant sur la clé de vérification ou sur le message signé. Le prédicat global de signature doit donc procéder en deux étapes : (1) déduire de la clé de vérification ou du message le type de signature dont il s'agit, (2) appeler le prédicat local correspondant à ce type de signature.

Stabilité des preuves. Une preuve de sécurité qui exploite la définition du prédicat global est fragile : si le prédicat global est modifié (par exemple pour inclure un nouveau prédicat local), alors les preuves qui en dépendent risquent de casser. De plus, de multiples prédicats doivent être définis lors d'une preuve avec DY*, ce qui invite à résoudre ce problème une fois pour toute.

Nous introduisons une façon générique de construire un prédicat global à partir d'une liste de prédicats locaux, dont nous présentons ici une version simplifiée. Par la suite, les preuves de sécurité ne dépendront pas directement du prédicat global (et de la façon dont il a été défini), elles dépendront seulement du fait qu'un quelconque prédicat global contient un certain prédicat local.

Construction d'un prédicat global. Étant donné un type de donnée `data`, un type de catégorie `category` et une fonction qui déduit la catégorie associée à une donnée, nous construisons un prédicat global à partir d'une liste de catégories et de prédicats locaux. Ce prédicat global est prouvé correct, c'est-à-dire qu'il est équivalent à chacun des prédicats locaux sur les données ayant la catégorie correspondante.

```

val data: Type // le type sur lequel nous voulons construire un prédicat
val category: Type // le type décrivant la catégorie à laquelle appartient une donnée
val get_category: data → option category // une fonction de désambiguation

val mk_global_pred: list (category & (data → prop)) → (data → prop)

let global_pred_has_local_pred (global_pred:data → prop) (cat:category) (local_pred:data → prop) =
  ∀(x:data). get_category x == Some cat ⇒ (global_pred x ⇔ local_pred x)

val mk_global_pred_correct:
  local_preds:list (category & (data → prop)) → cat:category → local_pred:(data → prop) →
  Lemma
  (requires
    no_repeats (map fst local_preds) ∧ // les catégories sont disjointes
    mem (cat, local_pred) local_preds // cat and local_pred sont dans la liste local_preds
  )
  (ensures global_pred_has_local_pred (mk_global_pred local_preds) cat local_pred)

```

Par la suite, les preuves n'ont pas besoin de savoir comment est construit le prédicat global : elles peuvent être paramétrées par ce prédicat global et seulement savoir qu'il contient un prédicat local à l'aide de la propriété `global_pred_has_local_pred`. Une fois les preuves faites, on peut alors modifier le prédicat global sans que les preuves existantes cassent, puisqu'elles ne dépendent que du fait que le prédicat global contient un prédicat local donné.

5 Impact sur la standardisation du protocole MLS

Nous présentons ici brièvement une version simplifiée du protocole `TreeSync` et de son théorème de sécurité, ainsi que les améliorations que nous avons apporté à ce protocole. La description complète du protocole `TreeSync` et de sa preuve font l'objet d'un autre article [38], et la formalisation complète est disponible sur GitHub [20].

TreeSync et TreeKEM. `TreeKEM` et `TreeSync` travaillent sur un même arbre : un arbre binaire complet où les participants sont dans les feuilles (voir Figure 5). Chaque nœud interne contient une paire de clés de chiffrement asymétrique, avec la propriété suivante : la clé privée d'un nœud n'est connue que des participants dans son sous-arbre (propriété assurée par `TreeKEM`). La clé privée de la racine est alors un secret qui n'est connu que du groupe. Lorsqu'un nouveau participant rejoint le groupe, ce dernier reçoit un arbre rempli de clés publiques, et lorsqu'il chiffre un message pour une de ces clés publiques, s'attend à ce que ce message ne soit déchiffrable seulement par les participants dans le sous-arbre associé. Pour établir cette propriété, et éviter des attaques similaires à celle sur la première version de `MiniTLS` (§2.2), on peut se reposer sur les garanties d'authenticité offertes par `TreeSync`.

Théorème de sécurité pour TreeSync. Nous prouvons, à l'aide de `DY*`, que pour chaque nœud de l'arbre, nous pouvons trouver un participant dans le sous-arbre enraciné en ce nœud (celui qui l'a modifié en dernier) dont la signature authentifie ce sous-arbre tel qu'il était au

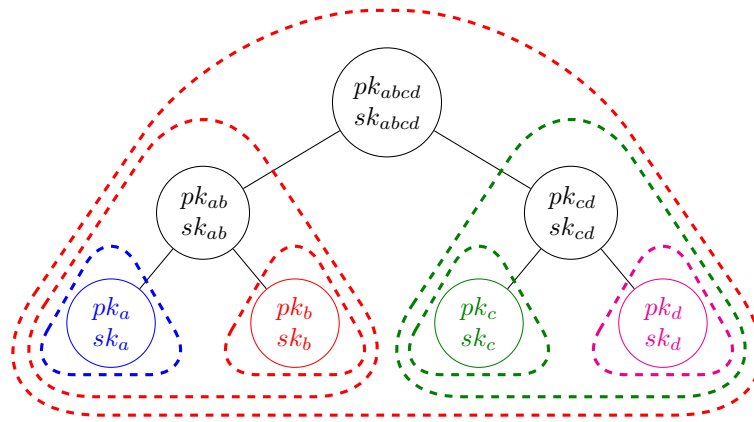


FIGURE 5 – Un arbre de clés TreeKEM, authentifié par TreeSync, où chaque sous-arbre est entouré de la couleur de la feuille qui l’authentifie. TreeKEM s’assure du fait qu’une clé privée sk_n n’est connue que des feuilles du sous-arbre enraciné en n , par exemple sk_{ab} n’est connu que de a et b . TreeSync s’assure du fait que chaque sous-arbre est authentifié par une de ses feuilles, par exemple le nœud ab est authentifié par b (en rouge). Une feuille peut authentifier plusieurs sous-arbres en même temps, par exemple b authentifie trois sous-arbres : celui enraciné en b , en ab et en $abcd$.

moment de la signature. Cette signature permet entre-autres d’initialiser la propriété de TreeKEM grâce à un invariant de signature. La preuve est complexe pour deux raisons : d’une part, il est possible d’ajouter des participants dans les feuilles d’un sous-arbre sans pour autant ré-authentifier celui-ci, il faut donc retrouver l’arbre tel qu’il était au moment de son authentification ; d’autre part, une seule signature authentifie de multiples sous-arbres (tous ceux dont la racine a été modifiée par le participant) et cette signature reste vérifiable même après que certains de ces sous-arbres ont été modifiés par d’autres personnes.

Attaque sur les signatures. Nous avons trouvé une attaque similaire à celle entre MiniTLS en SigneTout (§2.3), dû au fait que TreeSync et TreeDEM utilisent les mêmes clés de signature. Nous avons modifié le protocole MLS pour ajouter un tag de façon systématique dans les signatures, ce qui permet de les distinguer [32].

Renforcement des invariants. Les invariants des arbres manipulés dans MLS doivent être vrais lors de la création du groupe et être préservés par toutes les modifications du groupe. Mais ils doivent aussi être vérifiés au moment de rejoindre un groupe, sinon nous ne pouvons pas les utiliser lors des preuves de sécurité. Au cours de notre analyse de TreeSync, nous avons remarqué que les invariants vérifiés au moment de rejoindre un groupe n’étaient pas assez forts pour être préservés par les modifications du groupe. Nous les avons renforcés pour que ce soient de vrais invariants de MLS [35, 36].

Renforcement du théorème de sécurité. Le théorème de sécurité de TreeSync tel que décrit au début de cette section n’était initialement pas vrai : le contenu couvert par la signature était légèrement malléable, ce qui aurait rendu le théorème d’authentification plus compliqué à écrire. Nous avons renforcé la signature [37, 33, 34] pour pouvoir obtenir ce théorème de sécurité à la fois puissant et simple à énoncer : la signature des participants couvre exactement tous les sous-arbres modifiés par ce participant, là où auparavant elle ne couvrait qu’une partie de ces sous-arbres qu’il aurait fallu caractériser.

6 Travaux connexes

Analyse de la sécurité de MLS. J. Alwen et al. [2] étudie la sécurité de TreeKEM du brouillon 7 de MLS contre un adversaire passif. J. Alwen et al. [3] étudie modulairement la sécurité du brouillon 11 de MLS contre un adversaire actif. Ils ne trouvent pas l’attaque exploitant l’ambiguïté des signatures, que nous avons trouvé en faisant les preuves jusqu’aux manipulations de suites d’octets. C. Brzuska et al. [14] analyse les dérivations de clé du brouillon 11 de MLS. C. Cremers et al. [15] étudie la sécurité de MLS dans un contexte multi-groupe.

J. Alwen et al. [4] étudie la sécurité de TreeKEM contre un adversaire interne au groupe, et analyse les signatures du brouillon 11 de MLS. Cependant, ils étudient TreeSync et TreeKEM comme un protocole monolithique.

Tous ces travaux se basent sur des preuves à la main, papier et crayon. Au fur et à mesure que MLS grandit, ces preuves manuelles deviennent complexes, difficiles à vérifier et à maintenir. Nous utilisons des outils de vérification assistée par ordinateur afin de faire des preuves sur une implémentation de MLS interopérable.

Une analyse de la confidentialité persistante de TreeKEM en Tamarin a été réalisée dans [17], sans considérer la sécurité après compromission ni l’authentification. K. Bhargavan et al. [8] utilise F* pour analyser TreeKEM du brouillon 7 de MLS et trouve une attaque sur l’authentification, mais ne prouve pas l’authentification de façon indépendante.

Preuve assistée par ordinateur de protocoles cryptographiques. Certains outils de vérification de protocole se basent sur le modèle symbolique, qui traite la cryptographie de façon abstraite. Certains outils sont automatiques, comme ProVerif [13] et Tamarin [25], d’autres outils se basent sur un assistant de preuve général et permettent des preuves manuelles, comme DY* [9] qui se repose sur F* [31], ou les travaux de L. Paulson [27] qui se reposent sur Isabelle [26]. Des outils comme CryptoVerif [12], EasyCrypt [7], et Squirrel [5], se basent sur le modèle calculatoire, ce qui permet d’avoir une modélisation plus précise de la cryptographie mais permet moins d’automatisation. Les deux types d’outils ont été utilisés pour des protocoles de la vie courante, comme Signal [21, 9] et TLS 1.3 [10, 16].

Finalement, beaucoup de travaux vérifient la correction fonctionnelle d’implémentations de protocoles comme Signal [28], Noise [19], et TLS [11].

7 Conclusion

Nous avons produit une spécification formelle et interopérable de la version actuelle de MLS, ainsi qu’une preuve assistée par ordinateur de la sécurité de son sous-protocole TreeSync à l’aide du cadriciel DY*. Au cours de ce travail nous avons corrigé divers bogues et amélioré la conception de MLS, et nous avons développé de nouveaux outils pour faire des preuves de sécurité avec DY*.

8 Remerciements

Je remercie Lucas Franceschino pour ses remarques sur le premier brouillon de cet article, ainsi que (par ordre alphabétique) Adrien Koutsos, Antonin Reitz, Aymeric Fromherz, Jonathan Protzenko, Sylvain Wallez, et les rapporteurs anonymes pour leur relecture de cet article et leurs nombreuses suggestions.

Références

- [1] Danel Ahman, Cédric Fournet, Catalin Hritcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. Recalling a witness : foundations and applications of monotonic state. *Proc. ACM Program. Lang.*, 2(POPL) :65 :1–65 :30, 2018.
- [2] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In *CRYPTO*, volume 12170 of *Lecture Notes in Computer Science*, pages 248–277. Springer, 2020.
- [3] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1463–1483, 2021.
- [4] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. Cryptology ePrint Archive, Paper 2020/1327, 2020.
- [5] David Baelde, Stéphanie Delaune, Charlie Jacomme, Adrien Koutsos, and Solène Moreau. An interactive prover for protocol verification in the computational model. In *IEEE Symposium on Security and Privacy (S&P)*, pages 537–554. IEEE, 2021.
- [6] R. Barnes, J. Millican B. Beurdouche, R. Robert, E. Omara, and K. Cohn-Gordon. The messaging layer security protocol. IETF Internet Draft, September 2022. version 16.
- [7] Gilles Barthe, Benjamin Grégoire, Sylvain Héraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO*, pages 71–90, 2011.
- [8] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal Models and Verified Protocols for Group Messaging : Attacks and Proofs for IETF MLS. Research report, Inria Paris, December 2019.
- [9] Karthikeyan Bhargavan, Abhishek Bichhawat, Quoc Huy Do, Pedram Hosseini, Ralf Küsters, Guido Schmitz, and Tim Würtele. DY* : A modular symbolic verification framework for executable cryptographic protocol code. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 523–542. IEEE, 2021.
- [10] Karthikeyan Bhargavan, Bruno Blanchet, and Nadim Kobeissi. Verified models and reference implementations for the TLS 1.3 standard candidate. In *IEEE Symposium on Security and Privacy (S&P)*, pages 483–502. IEEE, 2017.
- [11] Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security and Privacy, (S&P)*, pages 445–459, 2013.
- [12] Bruno Blanchet. CryptoVerif : Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar “Formal Protocol Verification Applied*, volume 117, page 156, 2007.
- [13] Bruno Blanchet et al. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends in Privacy and Security*, 1(1-2) :1–135, 2016.
- [14] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Security analysis of the MLS key derivation. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2535–2553. IEEE, 2022.
- [15] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging : Why cross-group effects matter. In *USENIX Security Symposium*, pages 1847–1864. USENIX Association, 2021.
- [16] Cas Cremers, Marko Horvat, Jonathan Hoyland, Sam Scott, and Thyla van der Merwe. A comprehensive symbolic analysis of TLS 1.3. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, page 1773–1788, 2017.
- [17] Cas Cremers, Charlie Jacomme, and Philip Lukert. Subterm-based proof techniques for improving the automation and scope of security protocol analysis. Cryptology ePrint Archive, Paper 2022/1130, 2022. <https://eprint.iacr.org/2022/1130>.
- [18] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on*

- information theory*, 29(2) :198–208, 1983.
- [19] Son Ho, Jonathan Protzenko, Abhishek Bichhawat, and Karthikeyan Bhargavan. Noise* : A library of verified high-performance secure channel protocol implementations. In *IEEE Symposium on Security and Privacy (S&P)*, pages 107–124, 2022.
 - [20] Inria-Prosecco. TreeSync : Supplementary material, 2022. <https://github.com/Inria-Prosecco/treesync>.
 - [21] Nadim Kobeissi, Karthikeyan Bhargavan, and Bruno Blanchet. Automated verification for secure messaging protocols and their implementations : A symbolic and computational approach. In *IEEE European symposium on security and privacy (EuroS&P)*, pages 435–450. IEEE, 2017.
 - [22] Moxie Marlinspike and Trevor Perrin. Signal protocol, 2016. <https://signal.org/docs>.
 - [23] Moxie Marlinspike and Trevor Perrin. The X3DH key agreement protocol, 2016. <https://signal.org/docs/specifications/x3dh/>.
 - [24] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Catalin Hritcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. Meta-F* : Proof automation with SMT, tactics, and metaprograms. In Luís Caires, editor, *Programming Languages and Systems - European Symposium on Programming, ESOP, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 30–59. Springer, 2019.
 - [25] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The Tamarin prover for the symbolic analysis of security protocols. In *International conference on computer aided verification*, pages 696–701. Springer, 2013.
 - [26] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.
 - [27] Lawrence Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6, 12 2000.
 - [28] Jonathan Protzenko, Benjamin Beurdouche, Denis Merigoux, and Karthikeyan Bhargavan. Formally verified cryptographic web applications in webassembly. In *IEEE Symposium on Security and Privacy (S&P)*, pages 1256–1274. IEEE, 2019.
 - [29] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Programming and proving with indexed effects, 2021. <https://www.fstar-lang.org/papers/indexedeffects/>.
 - [30] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018.
 - [31] Nikhil Swamy, Cătălin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguélin. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270, 2016.
 - [32] Théophile Wallez. Unambiguous signatures. MLS protocol, pull-request #526, 2021. <https://github.com/mlswg/mls-protocol/pull/526>.
 - [33] Théophile Wallez. Improve parent hash guarantees. MLS protocol, pull-request #713, 2022. <https://github.com/mlswg/mls-protocol/pull/713>.
 - [34] Théophile Wallez. Include leaf index in LeafNodeTBS for better parent-hash guarantees. MLS protocol, pull-request #731, 2022. <https://github.com/mlswg/mls-protocol/pull/731>.
 - [35] Théophile Wallez. MLS protocol, comment on pull-request #713, 2022. <https://github.com/mlswg/mls-protocol/pull/713#issuecomment-1146371281>.
 - [36] Théophile Wallez. MLS protocol, comment on pull-request #713, 2022. <https://github.com/mlswg/mls-protocol/pull/713#issuecomment-1146599668>.
 - [37] Théophile Wallez. Stronger parent hashes for authenticated identities. MLS protocol pull-request

- #527, 2022. <https://github.com/mlswg/mls-protocol/pull/527>.
- [38] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. Tree-Sync : Authenticated group management for messaging layer security. Cryptology ePrint Archive, Paper 2022/1732, 2022. <https://eprint.iacr.org/2022/1732>.
- [39] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL* : A verified modern cryptographic library. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1789–1806, 2017.

Articles courts

Comment dompter un troupeau de flottants sauvages ?

Arthur Correnson

CISPA Helmholtz Center for Information Security

`arthur.correnson@cispa.de`

École Normale Supérieure de Rennes

`arthur.correnson@ens-rennes.fr`

Résumé

Les nombres flottants tels que définis dans le standard IEEE-754 sont la solution la plus répandue pour approximer les nombres réels en machine. Toutefois, en pratique, de nombreux environnements logiciels et matériels présentent des subtilités non conformes avec ce standard. Cela complique la tâche de raisonner formellement sur les programmes effectuant des calculs numériques sans connaître *a priori* leur environnement d'exécution. Dans cet article, nous montrons l'impact de ce problème sur le compilateur CompCert, un compilateur formellement vérifié pour le langage C et proposant un support pour les flottants IEEE-754. Nous présentons les résultats d'un travail d'investigation visant à mesurer les difficultés liées à l'intégration de nouveaux types de flottants dans CompCert. Cette étude est motivée par la volonté d'étendre le compilateur à des cibles non conformes IEEE-754 utilisées, par exemple, en informatique embarquée.

Introduction

Contexte Représenter de manière exacte les nombres réels est impossible dans la mémoire finie d'un ordinateur. Les nombres flottants répondent à ce problème et propose une méthode pour approximer les nombres réels de manière finie au prix d'une perte de précision. Le standard IEEE-754 [2] normalise la notion de nombres flottants et définit un encodage binaire de ces nombres ainsi que le comportement attendu des opérations arithmétiques associées.

Malgré l'existence d'un standard pour l'implémentation des nombres flottants, de nombreux environnements logiciels et matériels présentent des spécificités non conformes IEEE-754. Ces particularités résultent souvent de contraintes techniques comme par exemple le coût de fabrication d'un processeur flottant ou la rapidité des calculs. Cette grande diversité des nombres flottants pose un véritable problème de compréhension des résultats produits par les programmes effectuant des calculs numériques. En effet, sans connaître au préalable l'environnement dans lequel va être exécuté un programme, anticiper le résultat des opérations flottantes devient pratiquement impossible. Ce problème est d'autant plus flagrant dans le contexte de la programmation certifiée dont le but est de prouver formellement que le comportement d'un programme est préservé par la compilation. Dans ce contexte, il est nécessaire de s'entendre sur l'interprétation des nombres flottants dans les langages de programmation mais également dans les environnements d'exécution ciblés par la compilation.

Le compilateur CompCert [10, 11, 9] est un compilateur certifié pour le langage C entièrement prouvé grâce à l'assistant de preuve Coq [1]. CompCert fait le choix de fixer l'usage du standard IEEE-754 [5], limitant ainsi la compilation à des cibles dont le traitement des flottants est conforme au standard. Toute la chaîne de compilation de CompCert dépendant fortement de ce choix, l'intégration d'un support pour de nouveaux types de nombres flottants pose plusieurs défis en matière d'ingénierie logicielle. Une première problématique est de rendre interchangeable le modèle de calcul flottant du compilateur sans invalider la chaîne de compilation et sa preuve de correction. Au delà de cette question, la tâche de formaliser en Coq de nouveaux

modèles de calcul flottant est difficile en elle-même et impose en particulier de s'assurer que les spécifications en Coq sont bien conformes aux documentations de référence.

Contributions Dans cet article nous présentons les résultats d'un travail d'investigation visant à mesurer les difficultés liées à l'intégration de nouveaux types de flottants dans CompCert. Cette étude effectuée au sein de l'entreprise AbsInt est motivée par la volonté d'étendre le compilateur à des cibles non conformes IEEE-754 utilisées notamment en informatique embarquée.

Plan Nous commençons par introduire le problème de la diversité des nombres flottants en présentant des exemples concrets (1). Nous exposons ensuite la notion de compilation certifiée et en particulier la compilation des nombres flottants dans le compilateur CompCert (2). Enfin, nous discutons des difficultés engendré par l'intégration de nouveaux types de flottants dans CompCert (3) et nous proposons plusieurs pistes de réflexion (4).

1 Les nombres flottants et le standard IEEE-754

Les nombres flottants IEEE Le standard IEEE-754 définit une représentation binaire des nombres flottants sur 16, 32 et 64 bits. Pour le cas des flottants 32 bits par exemple, l'encodage comporte 1 bit de signe (**s**), 8 bits d'exposant (**exp**) et les 23 bits restants pour représenter une partie fractionnaire (**frac**). En fonction des valeurs exactes de chaque champs, les nombres flottants sont interprétés au choix parmi 5 types de valeurs :

Valeurs	Interprétation	condition
NaN	valeur indéterminée	$\mathbf{exp} = 255, \mathbf{frac} > 0$
$+\infty, -\infty$	infinis signés en fonction du bit s	$\mathbf{exp} = 255, \mathbf{frac} = 0$
$+0, -0$	zéros signés en fonction du bit s	$\mathbf{exp} = 0, \mathbf{frac} = 0$
Nombres normalisés	$(-1)^{\mathbf{s}} \times 2^{\mathbf{exp}-127} \times (1.\mathbf{frac})_2$	$1 \leq \mathbf{exp} < 255$
Nombres dénormalisés	$(-1)^{\mathbf{s}} \times 2^{\mathbf{exp}-127} \times (0.\mathbf{frac})_2$	$\mathbf{exp} = 0, \mathbf{frac} > 0$

Le standard introduit également plusieurs notion d'arrondi associant à tout réel x une représentation flottante $\mathbf{round}_m(x)$ (m est un mode d'arrondi). Les opérations arithmétiques sont ensuite définies comme devant être l'arrondi de leur pendant réel. Par exemple pour l'addition flottante en mode d'arrondi m (notée \oplus_m), on a $x \oplus_m y := \mathbf{round}_m(x + y)$. Notons que la présence d'un arrondi dans la définition suffit à invalider certaines propriétés arithmétiques comme l'associativité de l'addition.

Le standard et le mode d'arrondi par défaut Parmi les 5 modes d'arrondi définis dans le standard IEEE-754, le mode "au plus proche pair" arrondit tout réel au plus proche flottant dont la représentation binaire a un bit de poids faible pair. Ce mode spécifique est utilisé par défaut dans de très nombreux environnements, si bien que l'on précise rarement le mode d'arrondi explicitement dans les langages de programmation. Cependant, il est parfois préférable d'utiliser d'autres modes pour effectuer certains calculs. Ce choix d'un mode d'arrondi est donc un facteur qui contribue aux différences observables entre environnements de calcul flottants, y-compris entre environnements conformes au standard IEEE-754.

Des imprécisions au niveau logiciel Les langages de programmation ne sont pas toujours précis en ce qui concerne l'arithmétique flottante. La norme ISO du langage C [8] par exemple

laisse une grande liberté dans le traitement des opérations flottantes (voir figure 1). En particulier, l'usage du standard IEEE-754 n'est pas requis mais plutôt recommandé. Il en résulte qu'un même programme C peut produire des résultats flottants différents selon le compilateur utilisé, la machine qui exécute le programme ou même selon la machine qui exécute le compilateur. Ces imprécisions du langage C ainsi que d'autres langages sont davantage détaillées par Sylvie Boldo et ses contributeurs [6, 4].

*The accuracy of the floating-point operations (+, -, *, /) and of the library functions in <math.h> and <complex.h> that return floating-point results is implementation-defined. The implementation may state that the accuracy is unknown*

FIGURE 1 – Extrait de *ISO/IEC 9899, section 5.2.4.2.2 "characteristics of floating types"*

Des divergences au niveau matériel La plupart des processeurs modernes embarquent une unité de calcul flottant. Les processeurs grand-public que l'on trouve dans les ordinateurs personnels implémentent généralement le standard IEEE-754. Toutefois, certaines familles de processeurs utilisés dans des domaines spécifiques (informatique embarquée, traitement du signal, etc) prennent des libertés par rapport au standard. Par exemple, le standard SPE (Signal Processing Engine) et un document normatif décrivant une famille d'architectures matérielles dédiées au traitement du signal. Cette norme diffère explicitement du standard IEEE en autorisant que les opérations usuelles ne produisent pas de valeurs spéciales (infinis, NaN) ou de nombres dénormalisés (voir figure 2). Des règles spécifiques sont définies en remplacement.

Embedded floating-point operations do not produce +Inf, -Inf, NaN, or a denormalized number. [...] Thus, whenever an input operand of the embedded floating-point instruction has data values that are +infinity, -infinity, aliased, NaN, or when the result of an operation produces an overflow or an underflow, an embedded floating-point data interrupt may be taken and the interrupt handler is responsible for delivering IEEE 754-compliant behavior if desired

FIGURE 2 – Extrait de *Signal Processing Engine (SPE) Programming Environments Manual, section 3.3.1.4 about IEEE Std 754 Compliance*

2 Vérifier la compilation des nombres flottants

2.1 Préserver la sémantique

La compilation certifiée a pour but de s'assurer que le sens des programmes est conservé lorsqu'ils sont traduits en exécutables. Cela permet de garantir que l'intention du programmeur n'est pas détériorée par le compilateur, par exemple en effectuant des optimisations hasardeuses. Le caractère parfois contre-intuitif de l'arithmétique flottante est typiquement source

d'optimisations incorrectes. Par exemple, en nombres flottants, $x \oplus 0.0$ ne peut pas toujours être optimisé en x . En effet, dans le cas où $x = -0.0$, le standard IEEE impose que le résultat de $-0.0 + 0.0 = +0.0$ et $+0.0 \neq -0.0$. Certifier un compilateur permet de contrôler ce type d'erreurs.

Pour garantir qu'un compilateur préserve le sens des programmes, il est nécessaire de définir au préalable la sémantique des langages sources et cibles considérés pour la compilation. Pour un langage donné \mathcal{L} , un programme $P \in \mathcal{L}$ et un comportement observable C , on définit la relation $P \Downarrow C$ qui se lit comme " C est un comportement observable lors de l'exécution du programme P ". Dans le cadre de la compilation d'un langage \mathcal{L}_{source} vers \mathcal{L}_{cible} , un compilateur peut être vu comme une fonction partielle $\text{compile} : \mathcal{L}_{source} \rightarrow \mathcal{L}_{cible}$. Étant données deux sémantiques \Downarrow_{source} et \Downarrow_{cible} décrivant l'exécution des programmes sources et cibles, certifier le compilateur compile revient alors à démontrer le théorème suivant [10] :

$$\boxed{\forall P \in \mathcal{L}_{source}, \forall P' \in \mathcal{L}_{cible}, \forall C, \text{compile}(P) = P' \Rightarrow (P \Downarrow_{source} C \iff P' \Downarrow_{cible} C)}$$

Cette équivalence assure que les programmes compilés se comportent exactement comme dicté par leur code source. Notons que cette équivalence est souvent trop restrictive, en particulier dans le cas où le langage source est non-déterministe. D'autres variations de l'énoncé de correction d'un compilateur sont discutées dans les travaux de Xavier Leroy [10]. Le logiciel CompCert suit cette approche et propose d'implémenter un compilateur C dans l'assistant à la démonstration Coq. Cela permet de réaliser la preuve formelle de la correction du compilateur.

2.2 Modéliser la sémantique des nombres flottants

La définition d'une sémantique pour le langage C (ou tout autre langage proposant un support les nombres flottants) doit notamment modéliser l'évaluation des expressions manipulant des valeurs numériques. On donne pour cela un modèle mathématique de l'arithmétique flottante composé d'un ensemble support \mathbb{F} représentant les nombres flottants ainsi qu'un ensemble de fonctions \oplus, \ominus, \otimes etc pour interpréter les opérateurs. Deux questions se posent alors. D'une part quelle modélisation choisir parmi les différentes arithmétiques flottantes existantes ? D'autre part, une fois le modèle choisi, comment formaliser ce modèle dans le langage d'un assistant à la démonstration comme Coq ? Plusieurs réponses à ces problèmes existent dans la littérature [6, 7, 5]. Nous décrivons brièvement ici celles déployées dans CompCert.

Axiomatiser les flottants Une première solution pour formaliser les nombres flottants et de les axiomatiser : on postule leur existence ainsi que celle des opérations arithmétiques utiles. On peut également axiomatiser quelques propriétés nécessaires pour prouver la correction du compilateur. Cette approche a été initialement utilisée dans les premières versions du compilateur et pose deux principaux problèmes. Tout d'abord, elle nécessite de s'entendre sur les propriétés que l'on peut raisonnablement admettre. Ces propriétés peuvent varier d'une arithmétique à l'autre. Par ailleurs, en choisissant d'axiomatiser les nombres flottants, on s'interdit la possibilité d'effectuer des calculs lors de la compilation pour le besoin de certaines optimisations par exemple. Une solution pour pouvoir tout de même calculer lors de la compilation est d'utiliser les flottants du processeur de la machine qui exécute le compilateur. Toutefois, cette démarche fait l'hypothèse que le processeur employé traite les flottants de manière fidèle à l'axiomatisation choisie. Cette hypothèse n'est pas toujours réaliste et l'article [6] relève déjà cette limitation.

Imposer le standard IEEE-754 Une autre méthode pour formaliser le comportement des nombres flottants est d'imposer l'usage du standard IEEE-754. Ce choix est compatible avec la norme ISO C et permet de fixer une implémentation logicielle et formellement prouvée du standard IEEE-754. Cette approche est celle actuellement mise en œuvre dans le compilateur CompCert [5]. Les flottants sont représentés grâce à la bibliothèque Flocq [6] qui propose une formalisation du standard IEEE-754 en Coq ainsi qu'une implémentation prouvée des différentes opérations flottantes. Si cette solution résout le problème du calcul, elle ne permet cependant pas de décrire la sémantique des cibles de compilation qui ne supportent pas le standard IEEE-754.

3 Le défis de l'extension de CompCert à des cibles non conformes IEEE-754

L'approche par axiomatisation ainsi que celle consistant à fixer l'emploi du standard IEEE-754 limitent les possibilités. La première offre une certaine souplesse mais rend impossible le calcul. La deuxième permet de faire des calculs mais restreint la compilation à des cibles compatibles avec le standard. Nous explorons une troisième approche dans laquelle plusieurs standards sont modélisés. Le modèle de calcul flottant utilisé par CompCert est alors déterminé au moment du choix de la cible de compilation. Cette approche révèle de nombreux défis d'ingénierie logicielle que nous présentons ici.

Contrôler les conséquences sur la chaîne de compilation L'architecture du compilateur CompCert comprend une partie commune à toutes les cibles de compilation (analyse syntaxique, typage, optimisations de haut niveau, etc) ainsi que des parties spécifiques à chaque cible de compilation (jeux d'instructions, génération de code, optimisations de bas niveau, etc). Ces deux parties dépendent d'une seule et même modélisation des nombres flottants qui est utilisée aussi bien pour décrire la sémantique du langage C que celle de toutes les cibles de compilation. En rendant interchangeable le modèle de calcul flottant, on risque donc d'invalider toute la chaîne de compilation et sa preuve de correction. En particulier, il faut identifier quels sont les théorèmes d'arithmétique flottante qui peuvent être utilisés dans la partie commune du code si le modèle de calcul flottant est fixé par la cible de compilation. Un travail d'isolation des propriétés communes et spécifiques des différents modèles de calculs numériques est donc nécessaire. En assurant que seule des propriétés universelles sont utilisées dans la partie commune du compilateur, on garanti la possibilité de changer le modèle de calcul sans invalider les preuves déjà effectuées.

Valider les spécifications Pour chaque architecture présentant une unité de calcul flottante non conforme, il est nécessaire de modéliser son arithmétique associée en Coq. En multipliant le nombre de modèles différents, nous prenons davantage de risque d'introduire des erreurs dans leurs formalisations. De plus, les documents normatifs préfèrent souvent la prose aux équations et l'écart de formalisme entre le langage naturel et une définition Coq rend difficile la relecture et la validation *a posteriori* des spécifications. Pour illustrer cette difficulté, nous prenons l'exemple de l'addition flottante et sa spécification dans le standard IEEE-754 (voir figure 3).

Factoriser pour ne pas re-coder et re-prouver La tâche de formalisation d'un modèle de calcul flottant dans un assistant à la démonstration peut se décomposer en trois temps. Un première étape est de modéliser les nombres flottants en choisissant une structure de données adaptée. On peut dans un second temps implémenter les opérations arithmétiques sous forme de

Each of the computational operations that return a numeric result specified by this standard shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format (see 4 and 7). Clause 6 augments the following specifications to cover ± 0 , $\pm\infty$, and NaN.

FIGURE 3 – Spécification des opérations flottantes dans le standard IEEE

fonctions Coq opérant sur la représentation préalablement choisie. Enfin, on développe la preuve formelle des propriétés arithmétiques vérifiées par les opérateurs. Ces propriétés seront utilisées pour justifier la correction du compilateur. Cette suite d'étapes est très consommatrice de temps, en particulier le développement des preuves. Nous constatons toutefois que les différents modèles de calcul flottants étudiés partagent un socle commun avec le standard IEEE-754, en particulier en matière de choix de représentation des nombres flottants et de vocabulaire employé dans les documents normatifs. Il est donc possible (voire nécessaire) de factoriser les étapes de modélisation, par exemple en utilisant les structures de données et les définitions fournies par la bibliothèque Flocq. En revanche, des différences notables subsistent dans le comportement des opérations et donc des propriétés arithmétiques. Les preuves doivent donc être traitées au cas par cas. Notons toutefois qu'utiliser le même vocabulaire (le même catalogue de définitions Coq) pour exprimer tous les modèles de calcul rend possible le partage de lemmes intermédiaires entre les différentes preuves.

4 Les solutions explorées

Au regard des points mentionnés dans la section précédente, nous proposons différentes approches pour intégrer dans CompCert de nouveaux types de flottants.

4.1 Des flottants paramétrables ?

Fonctoriser le module d'arithmétique flottante Une première solution explorée pour modéliser de nouveaux nombres flottants est de rendre paramétrable l'implémentation actuelle reposant sur la bibliothèque Flocq. Cette solution permet par exemple de rendre modifiable le mode d'arrondi utilisé (fixé au plus proche pair dans CompCert). La bibliothèque Flocq étant générique sur les modes d'arrondi, les changements impliqués en matière d'implémentation sont presque immédiats : il suffit d'étendre le module `Floats` modélisant l'arithmétique flottante en un foncteur `Floats(M : MODE)` dépendant d'un mode d'arrondi fixé par la cible de compilation. Notons toutefois qu'en rendant générique le mode d'arrondi utilisé par les opérations, il faut également rendre générique les théorèmes d'arithmétiques associés. Nous avons observé que les théorèmes utilisés dans CompCert sont indépendants du mode d'arrondi utilisé. La plupart des preuves peuvent donc être conservées. Certaines preuves doivent être adaptées mais les énoncés restent valides.

Limitations Cette approche permet de modéliser des subtiles nuances compatibles avec le standard IEEE-754 comme les différents modes d'arrondi. Toutefois, pour la modélisation de

flottants non conformes, cette approche n'est pas suffisante. En effet, si les écarts avec le standard sont trop importants, il est difficile d'identifier les paramètres à faire varier et de contrôler leur impact sur la validité des théorèmes.

4.2 Spécification formelle de flottants non conformes

Du pseudo-code pour réduire les écarts de formalisme Comme mentionné dans la section 3, un des principaux obstacles à la traduction de documents normatifs en code Coq est l'écart entre niveaux de vocabulaire utilisés. Il est donc utile de se doter d'outils facilitant la traduction des documentations à dispositions (standards, manuels des processeurs, etc) en code Coq. On peut par exemple passer par l'intermédiaire de pseudo-code. Cela permet d'éviter les transitions abruptes du langage naturel au langage formel tout en faisant apparaître explicitement les objets mathématiques nécessaires à l'expression des spécifications en Coq. Nous prenons pour exemple la spécification de l'addition flottante du processeur PowerPC e200z4 (voir figure 4).

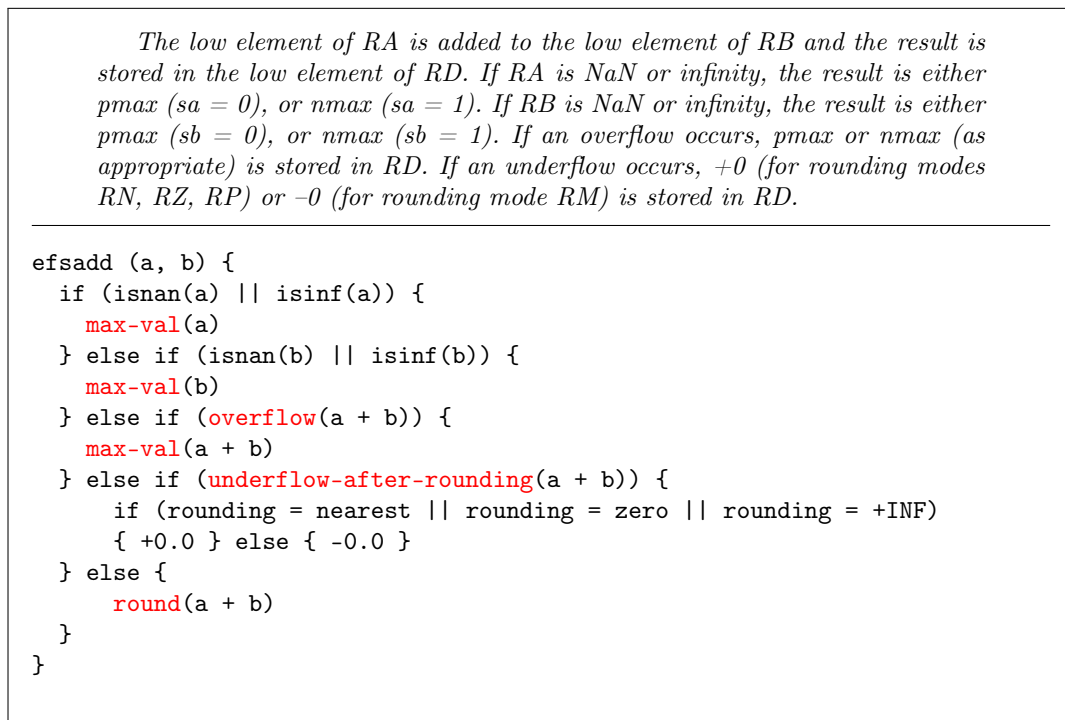


FIGURE 4 – Spécification de l'addition flottante dans le manuel du processeur e200z4 et sa traduction sous forme de pseudo-code

Conversion du pseudo-code en Coq Une fois le pseudo-code écrit et les définitions mathématiques utiles identifiées, nous pouvons le traduire manuellement en Coq. Cette étape consiste essentiellement à choisir les types appropriés pour représenter les différents objets. Nous identifions également une collection de définitions primitives systématiquement utilisées dans les

documents normatifs pour l'arithmétique flottante (par exemple `round`, `overflow`, `underflow` apparaissant en rouge dans la figure 4). Nous implémentons ces primitives en Coq à l'aide de Floq ce qui permet de ré-exploiter les types et les théorèmes déjà établies dans la bibliothèque.

Validation Passer par l'intermédiaire du pseudo-code permet de répartir l'effort nécessaire à la validation des spécifications. Une première relecture du pseudo-code est faite pour s'assurer de son adéquation par rapport au standard de référence. Notons que cette relecture ne nécessite aucune connaissance du langage Coq. Une deuxième relecture permet ensuite de vérifier la correction du code Coq vis à vis du pseudo-code. L'usage de pseudo-code comme interface rend la relecture plus efficace et plus fiable en séparant explicitement deux champs de compétences : la connaissance du standard considéré et celle du langage Coq.

Limitations La traduction de standards en Coq en passant par l'intermédiaire de pseudo-code facilite la tâche de validation des spécifications. Toutefois, la traduction doit être faite manuellement et les spécifications Coq obtenues ne sont pas nécessairement exécutables. En particulier, elles utilisent des opérations sur les nombres réels qui ne peuvent pas être calculées en Coq. En revanche, ces spécifications formelles peuvent servir de référence pour l'implémentation et la vérification d'opérations exécutables.

5 Remerciements

Merci à François Bobot pour m'avoir fait découvrir (et sombrer dans) l'univers des nombres flottants. Merci également à Bernhard Schommer et Christoph Mallon pour les discussions interminables que nous avons eu sur l'interprétation du standard IEEE et de la norme ISO C. Enfin, merci au comité des JFLA pour les commentaires très constructifs qui ont guidés la rédaction de la version finale de cet article.

6 Conclusion

Le compilateur CompCert propose un support pour les flottants IEEE-754 et restreint la compilation à des cibles conformes au standard. Nous avons étudié la possibilité d'intégrer dans CompCert un support pour d'autres arithmétiques flottantes et concluons cette étude par deux constats. D'une part, une telle extension nécessite un travail initial d'isolation de certaines composantes logicielles. Cela permettrait de rendre interchangeable le modèle de calcul flottant tout en limitant les conséquences sur la chaîne de compilation et en particulier sa preuve de correction. D'autre part, l'écart de formalisme entre les documents normatifs qui standardisent l'arithmétique flottante et le langage logique de Coq rend difficile la formalisation de nouveaux modèles de calculs. Passer par l'intermédiaire du pseudo-code peut faciliter cette tâche. Des questions restent cependant ouvertes, notamment concernant la possibilité d'exécuter les spécifications Coq. Développer un langage de spécification formelle dédié à l'arithmétique flottante et pouvant être compilé vers des spécifications Coq exécutables pourrait être une piste de recherche intéressante. Des travaux similaires ont déjà été menés pour la spécification d'architectures matérielles [12, 3].

Références

- [1] The coq proof assistant. <https://coq.inria.fr>.
- [2] Iso/iec/ieee international standard - floating-point arithmetic. *ISO/IEC 60559 :2020(E) IEEE Std 754-2019*, pages 1–86, 2020.
- [3] Alasdair Armstrong, Thomas Bauerei, Brian Campbell, Shaked Flur, Kathryn E. Gray, Prashanth Mundkur, Robert M. Norton, Christopher Pulte, Alastair David Reid, Peter Sewell, Ian David Bede Stark, and Mark Wassell. Detailed models of instruction set architectures : From pseudocode to formal semantics. 2018.
- [4] Sylvie Boldo. L'arithmétique des ordinateurs et sa formalisation. <https://www.college-de-france.fr/agenda/seminaire/semantiques-mecanisees-quand-la-machine-raisonne-sur-ses-langages/arithmetique-des-ordinateurs-et-sa-formalisation>, 2019.
- [5] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified Compilation of Floating-Point Computations. *Journal of Automated Reasoning*, 54(2) :135–163, February 2015.
- [6] Sylvie Boldo and Guillaume Melquiond. Flocq : A Unified Library for Proving Floating-point Algorithms in Coq. In Elisardo Antelo, David Hough, and Paolo Ienne, editors, *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, July 2011.
- [7] John Harrison. Floating-point verification using theorem proving. In Marco Bernardo and Alessandro Cimatti, editors, *Formal Methods for Hardware Verification*, pages 211–242, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.
- [8] ISO. *ISO/IEC 9899 :2011 Information technology — Programming languages — C*. International Organization for Standardization, 2011.
- [9] Daniel Kästner, Xavier Leroy, Sandrine Blazy, Bernhard Schommer, Michael Schmidt, and Christian Ferdinand. Closing the gap – the formally verified optimizing compiler CompCert. In *SSS'17 : Developments in System Safety Engineering : Proceedings of the Twenty-fifth Safety-critical Systems Symposium*, pages 163–180. CreateSpace, 2017.
- [10] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, July 2009.
- [11] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *ERTS 2016 : Embedded Real Time Software and Systems*. SEE, 2016.
- [12] Mark Wassell. Minisail - a kernel language for the isa specification language sail. *Archive of Formal Proofs*, June 2021. <https://isa-afp.org/entries/MiniSail.html>, Formal proof development.

L'arithmétique de séparation

Jean-Christophe Filiâtre et Andrei Paskevich

¹ LMF — Université Paris-Saclay, CNRS, ENS Paris-Saclay
² Inria

Résumé

Nous, praticiens de la preuve de programmes, souhaitons que le processus de la vérification soit le plus automatique possible. Les meilleurs outils pour cela sont à l'heure actuelle les démonstrateurs SMT, qui combinent notamment la logique du premier ordre et l'arithmétique linéaire. Par opposition, le raisonnement inductif n'est pas un point fort des démonstrateurs automatiques. Or, les programmes utilisant des pointeurs le font souvent pour manipuler des structures récursives : listes, arbres, etc.

Dans cet article, nous décrivons une approche qui permet d'amener la preuve de programmes avec pointeurs à la portée des démonstrateurs automatiques. L'idée consiste à projeter une structure récursive sur un domaine numérique, de sorte que les propriétés de possession et de séparation deviennent exprimables en terme de simples inégalités arithmétiques. En plus de simplifier la preuve, cela permet une spécification claire et naturelle. On illustre cette approche avec l'exemple classique du renversement en place d'une liste simplement chaînée.

1 Introduction

Dans le domaine de la preuve de programmes, et plus spécifiquement dans le domaine de la vérification déductive, nous sommes aujourd'hui grandement aidés par les démonstrateurs SMT, qui combinent notamment la logique du premier ordre et l'arithmétique linéaire. Ils nous permettent parfois d'obtenir des preuves entièrement automatiques, là où hier encore il fallait laborieusement construire une preuve interactivement et la faire vérifier par un assistant de preuve.

Les démonstrateurs SMT ont cependant des limites. En particulier, le raisonnement inductif n'est pas leur point fort. Or, les programmes utilisant des pointeurs le font souvent pour manipuler des structures qui sont récursives par nature, telles que des listes ou des arbres. Il est alors naturel de recourir à des définitions récursives pour les propriétés que l'on cherche à démontrer. Il y a alors une tension évidente entre deux principes, à savoir d'une part le principe qui affirme

La bonne spécification, c'est celle qui est facile à comprendre.

qui va dans le sens d'une définition récursive, naturelle pour la personne qui va lire et approuver la spécification, et le principe qui affirme d'autre part

Le bon invariant, c'est celui qui est facile à prouver.

qui va plutôt dans le sens d'une définition non récursive, plus adaptée aux démonstrateurs SMT.

Dans cet article, nous décrivons une approche qui permet dans certains cas de réconcilier ces deux principes, en amenant la preuve de programmes avec pointeurs à la portée des démonstrateurs automatiques. Nous l'illustrons sur l'exemple classique du renversement en place d'une liste simplement chaînée. L'idée clé consiste à se ramener à de simples inégalités dans l'arithmétique linéaire, un domaine de prédilection des démonstrateurs SMT, ce qui justifie le titre de cet article.

L'essentiel de cet article est composé autour de l'exemple du renversement d'une liste, avec sa spécification naturelle (section 2), une tentative échouée de preuve directe (section 3) et la proposition de notre approche arithmétique (section 4), que nous illustrons ensuite rapidement sur un autre cas d'étude (section 5). Nos expériences ont été conduites avec l'outil Why3 [2] et le code complet est donné en annexe. Toutefois, dans le but de donner plus d'universalité à notre propos, les éléments de spécification et de preuve sont donnés dans une syntaxe allégée.

2 Un grand classique

On illustre notre propos avec l'exemple classique du renversement en place d'une liste simplement chaînée. La figure 2 contient un code C qui implémente cet algorithme et illustre son fonctionnement sur une liste de cinq éléments. L'algorithme parcourt la liste une fois, avec la variable `l`, et redirige le pointeur `next` sur l'élément précédent, stocké dans la variable `r`.

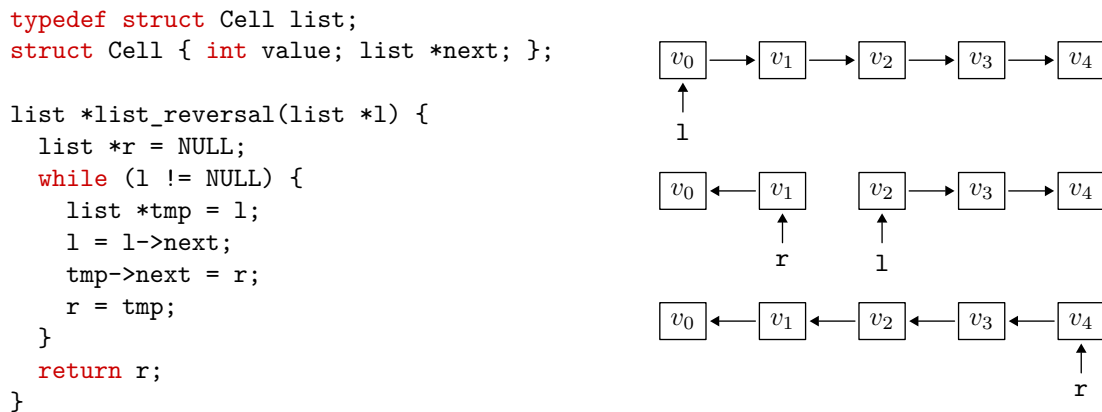


FIGURE 1 – Renversement en place d'une liste chaînée.

L'intérêt de ce programme pour la vérification est la coexistence, pendant la durée de la boucle, de deux listes simplement chaînées. Le fait que ces deux listes sont spatialement séparées nous permet d'établir la préservation des invariants. Plus précisément, l'affectation du pointeur `tmp->next` dans le code préserve la queue de la liste `l` d'une part et la liste `r` d'autre part.

Une spécification de la fonction `list_reversal` doit exprimer que les cellules de la liste initiale sont chaînées dans l'ordre inverse à l'issue de la fonction et le résultat de la fonction est la tête de cette nouvelle liste. Notons que la spécification ne peut pas se contenter de parler seulement du *contenu* des cellules, ce qui serait moins précis. En effet, le code pourrait par exemple inverser les valeurs contenues dans les cellules, mais laisser les champs `next` inchangés.

Pour écrire confortablement la spécification de la fonction, on doit représenter dans la logique, en plus des pointeurs manipulés par le programme, la mémoire elle-même (qu'on va appeler `mem`) et le modèle mathématique de la liste, sous la forme d'une séquence d'adresses mémoire (qu'on va appeler `s`). Étant donnés `mem`, `l` et `s`, nous pouvons définir un *prédicat de représentation*, `list mem l s`, qui exprime que, à l'adresse `l` dans la mémoire `mem`, se trouve une liste simplement chaînée dont les cellules forment la séquence `s`. En particulier, `l` est égal à `NULL` si et seulement si la séquence `s` est vide. Il est naturel de définir ce prédicat récursivement, sur la longueur de la séquence `s`.

```

predicate list mem l s =
  if length s = 0 then l = NULL
  else l = s[0] <> NULL && list mem[l].next s[1:]

```

Ici, `mem[l].next` désigne le contenu du champ `next` de la cellule à l'adresse `l` dans `mem` et `s[1:]` désigne la séquence `s` privée de son premier élément.

La spécification de `list_reversal` est alors très simple, en supposant donnée une fonction logique `rev` qui renverse une séquence :

```

list_reversal l (ghost s) : r
  requires list mem l s
  modifies mem
  ensures list mem r (rev s)

```

Ici, la séquence `s` est fournie comme un argument fantôme [3] de la fonction `list_reversal`, c'est-à-dire un argument qui n'est là que pour les besoins de la spécification.

La variable globale `mem` représente l'état de la mémoire : dans la précondition, à l'entrée de la fonction ; dans la postcondition, à la sortie. La spécification ci-dessus est incomplète si on ne dit pas que la mémoire reste inchangée en dehors des adresses qui composent la liste `l`. On peut exprimer cette propriété avec une seconde postcondition

```

ensures forall p. not (mem p s) -> mem[p] = old mem[p]

```

où `old mem[p]` désigne le contenu de la mémoire à l'adresse `p` à l'entrée de la fonction.

3 Droit dans le mur

Dans cette section, nous allons montrer en quoi la spécification ci-dessus, aussi naturelle qu'elle soit, ne se prête pas facilement à une preuve automatique. La première étape de la vérification consiste à exhiber les invariants de boucle. Ils se déduisent trivialement de la figure 2, en exprimant que `l` est un suffixe de la liste initiale et `r` un renversement du préfixe correspondant. En se donnant une variable fantôme `i` qui maintient le nombre de cellules déjà renversées, on a donc les deux invariants suivants

```

invariant list mem l s[i:]
invariant list mem r (rev s[:i])

```

où `s[i:]` est le suffixe de `s` à partir de la position `i` incluse et `s[:i]` le préfixe de `s` jusqu'à la position `i` exclue. En particulier, la sortie de boucle, où `i` est égal à la longueur de `s`, nous donne exactement la postcondition attendue. Jusque là, tout est parfaitement naturel et plutôt simple.

Les choses se compliquent lorsque l'on s'attaque à prouver que ces invariants de boucle sont préservés par une itération de la boucle. En effet, considérons l'état du programme à la fin d'une itération. L'invariant sur la nouvelle valeur de `l` se déduit directement de l'invariant sur sa valeur précédente, car on a uniquement suivi le champ `next` et `s[i+1:]` est égal à `s[i:] [1:]`. L'invariant sur la nouvelle valeur de `r` se déduit de l'invariant sur la valeur précédente, car à la tête de `rev s[:i+1]` on trouve `r` et `(rev s[:i+1]) [1:]` est égal à `rev s[:i]`.

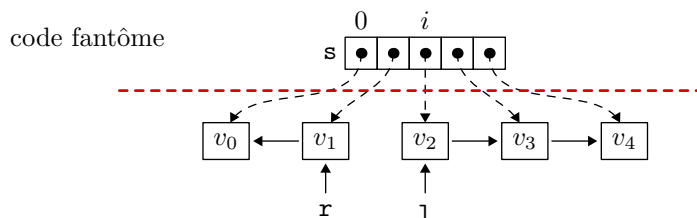
La difficulté consiste alors à prouver que les deux portions de la liste, à gauche de `r` et à droite de `l`, ont conservé la propriété `list`, puisque la modification de `tmp->next` n'a pas affecté les cellules qui les composent. La définition de `list` étant récursive, cette preuve nécessite une induction. Bien qu'il soit possible d'effectuer une telle induction, par exemple en utilisant un

démonstrateur comme Coq ou un autre moyen de preuve interactive, on préférerait une solution qui soit entièrement à la portée d'un démonstrateur automatique.

4 Approche arithmétique

La définition récursive du prédicat `list` introduite dans la section 2 nous pousse à représenter tout fragment de liste par un objet mathématique différent (ici une séquence). On a vu dans la section précédente en quoi cette approche est inadaptée à la preuve. Or, nous pouvons choisir une approche différente, “bas niveau”, et naviguer dans une description *globale* à l'aide d'indices.

Reprenons la séquence `s` des adresses mémoire des cellules de la liste initiale. À toute itération de l'algorithme, un préfixe $[0, i[$ de cette séquence, lu de droite à gauche, correspond à la liste `r` et le suffixe $[i, n[$ correspond à la liste `l`, ce que l'on peut représenter ainsi :



Pour l'instant, on va supposer de plus que les éléments de `s` sont non nuls et deux à deux distincts ; nous montrerons plus loin que ces deux propriétés peuvent être déduites du prédicat de représentation `list`. C'est ce que décrit le prédicat suivant :

```
predicate valid_cells s =
  (forall i. 0 <= i < length s -> s[i] <> null) &&
  (forall i j. 0 <= i, j < length s -> i <> j -> s[i] <> s[j])
```

On peut maintenant définir deux nouveaux prédicats de représentation pour une liste correspondant à *un préfixe* ou *un suffixe* de `s` :

```
predicate prefix mem s r i =
  0 <= i <= length s &&
  if i = 0 then r = null else
    r = s[i-1] && mem[s[0]].next = null &&
    forall k. 0 < k < i -> mem[s[k]].next = s[k-1]

predicate suffix mem s l i =
  0 <= i <= length s &&
  if i = length s then l = null else
    l = s[i] && mem[s[length s - 1]].next = null &&
    forall k. i < k < length s -> mem[s[k-1]].next = s[k]
```

Ces prédicats remplacent la récursivité par un quantificateur universel sur les indices à l'intérieur de `s`. C'est pourquoi on parle ici d'approche *arithmétique*.

Avec ces définitions, la fonction `list_reversal` a maintenant une spécification un peu différente de celle de la section 2 :

```
list_reversal (ghost s) l : r
```

```

requires valid_cells s
requires suffix mem s l 0
modifies mem
ensures prefix mem s r (length s)
ensures forall p. not (mem p s) -> mem[p] = old mem[p]

```

Mais surtout, les *invariants* de la boucle s'expriment eux aussi en termes des prédicats `suffix` et `prefix`,

```

invariant suffix mem s l i
invariant prefix mem s r i

```

et c'est là la clé d'une démonstration aisée. De fait, la preuve ne nécessite plus de raisonnement par induction et est maintenant entièrement à la portée des démonstrateurs automatiques.

Retomber sur ses pattes. Pour être complets, il nous reste cependant à faire le lien entre la spécification naturelle, introduite dans la section 2, et la preuve ci-dessus. Pour cela, il convient d'établir, à partir de la définition du prédicat récursif `list`, les deux préconditions de la fonction `list_reversal`, c'est-à-dire, `valid_cells` et `suffix`.

On peut le faire agréablement à l'aide d'une fonction fantôme pure, dite *fonction-lemme*, dont le seul objectif est de construire cette preuve.

```

lemma cells_of_list l s
requires list mem l s
ensures valid_cells s
ensures suffix mem s l 0
variant length s
= if s <> empty then cells_of_list mem[l].next s[1:]

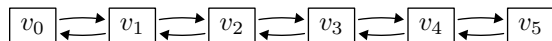
```

Cette fonction récursive suit la structure du prédicat `list`. De facto, le code de cette fonction représente une preuve par induction, écrite par l'utilisateur. Par le mécanisme de vérification de Why3, cette fonction, et donc le lemme, peuvent être vérifiés de manière purement automatique, quand bien même l'énoncé du lemme est en soi hors de portée d'un démonstrateur automatique. De la même façon, on peut écrire et prouver une fonction-lemme `list_of_cells` qui conclut `list mem r (rev s)` à partir des hypothèses `valid_cells` et `prefix`.

En composant ces deux lemmes et la fonction `list_reversal`, on obtient une fonction vérifiée, avec la spécification naturelle introduite au début de cet article.

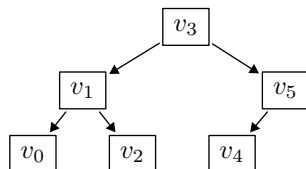
5 Un autre exemple

Nous décrivons ici un autre exemple de la même approche, un peu plus complexe. Il s'agit d'un problème proposé en 2021 dans le cadre de la compétition de preuve de programmes VerifyThis [1]. Un algorithme pour convertir une liste doublement chaînée en arbre binaire était donné, et il fallait en proposer une implémentation vérifiée. Plus précisément, on part d'une liste doublement chaînée contenant une séquence de valeurs v_0, v_1, \dots :



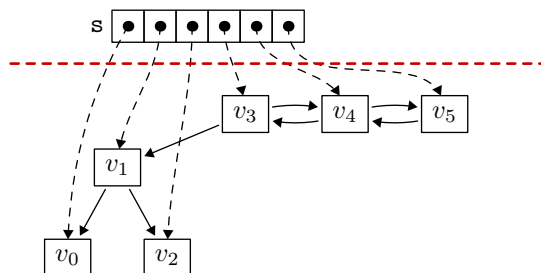
Chaque cellule contient un pointeur `prev` vers la cellule précédente (ou la valeur `null` pour la première cellule) et un pointeur `next` vers la cellule suivante (ou la valeur `null` pour la dernière cellule). L'algorithme réorganise alors les pointeurs `prev` et `next` pour former un arbre binaire,

le pointeur `prev` pointant maintenant vers le sous-arbre gauche et le pointeur `next` vers le sous-arbre droit.



Pour cela, l'algorithme commence par calculer la longueur n de la liste, puis se repose sur une fonction récursive qui prend en paramètres un pointeur vers le début de la liste et un nombre d'éléments à considérer, et qui renvoie la racine de l'arbre construit et le pointeur vers le premier élément non consommé. L'ordre des éléments est conservé, au sens où un parcours infixe de l'arbre énumère les valeurs v_0, v_1, \dots dans l'ordre.

Parmi les propriétés de l'algorithme à vérifier, il y avait le caractère équilibré de l'arbre obtenu au final, ou encore le fait que, si les valeurs v_i sont triées par ordre croissant, alors l'arbre obtenu est un arbre binaire de recherche. Pour cette vérification, nous pouvons avantageusement utiliser l'approche arithmétique. Comme pour le renversement de liste, on introduit une séquence fantôme s de toutes les cellules mémoires :



Sur la base de cette séquence, il est maintenant possible de définir un prédicat `dll s p lo hi` qui exprime que le pointeur `p` est la première cellule d'une liste doublement chaînée formée par les cellules de s situées entre les indices `lo` inclus et `hi` exclu, et de même un prédicat `tree s p lo hi` qui exprime que le pointeur `p` est la racine d'un arbre binaire dont les nœuds, dans l'ordre infixe, sont les cellules de s entre `lo` et `hi`. Une preuve Why3 réalisant cette idée est disponible en ligne [4]. Elle est majoritairement automatique.

6 Conclusion

Sur la base de deux exemples, nous avons montré comment, dans le contexte de la vérification déductive, une spécification récursive peut être avantageusement remplacée par une spécification ne faisant intervenir que des inégalités arithmétiques, et conduire alors à une preuve bien plus automatique. Nous anticipons que cette technique s'applique au-delà de ces deux exemples, sur toute structure récursive où un ordre de parcours définit une sérialisation de ses éléments. Pour poursuivre cette étude, il conviendrait d'explorer aussi des programmes qui allouent dynamiquement de la mémoire.

Bien évidemment, l'approche que nous avons présentée ici n'a aucune prétention à être aussi expressive et puissante que la logique de séparation [7]. Néanmoins, elle a le mérite de ne pas nécessiter une logique spécifique et des outils dédiés. Puisque nous travaillons dans un modèle mémoire plat, nous pouvons par ailleurs nous abstraire de toute problématique de ressource,

de possession, d'alias, etc. Et puisque notre représentation logique est construite sur la base d'adresses explicites, elle nous donne immédiatement accès à l'empreinte mémoire (*footprint*) de la structure. On se rapproche là des *dynamic frames* [5], une autre technique de preuve de programmes avec pointeurs utilisée par exemple dans Dafny [6]. Mais contrairement à cette approche, nos empreintes sont dotées d'une structure supplémentaire, à savoir leur caractère séquentiel, ce qui nous permet justement de faire de l'*arithmétique de séparation*.

Remerciements. Nous remercions nos collègues Claude Marché et Jacques-Henri Jourdan pour toutes les discussions relatives à la vérification du renversement d'une liste.

Références

- [1] The VerifyThis competition, Since 2011. <https://www.pm.inf.ethz.ch/research/verifythis.html>.
- [2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. The Why3 platform. <http://why3.lri.fr/>.
- [3] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3):152–174, 2016.
- [4] Jean-Christophe Filliâtre and Andrei Paskevich. Solution to verifythis 2021 challenge 2, 2021. https://toccata.gitlabpages.inria.fr/toccata/gallery/verifythis_2021_dll_to_bst.en.html.
- [5] I. T. Kassios. The dynamic frames theory. *Formal Aspects of Computing*, 23(3):267–288, May 2011.
- [6] 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.
- [7] 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.

A Le source Why3 complet

Le code ci-dessous a été vérifié avec la version 1.5.1 de Why3 (<https://why3.lri.fr/>) et les deux démonstrateurs SMT Alt-Ergo 2.4.0 (<https://alt-ergo.ocamlpro.com/>) et CVC5 1.0.0 (<https://github.com/cvc5/cvc5/>). La session Why3 peut être téléchargée à l'adresse https://www.lri.fr/~filliatr/jfla-2023/list_reversal.zip et ouverte avec les commandes suivantes :

```
$ unzip -x list_reversal.zip
$ cd jfla-2023
$ why3 ide list_reversal
```

Le code Why3 commence par modéliser les pointeurs (type `loc`) et la mémoire (variable globale `mem`), car les structures récursives mutables ne sont pas acceptées nativement par Why3. La séquence `s` est représentée par une simple fonction de type `int -> loc`.

(* Le modèle mémoire *)

```
use int.Int
use map.Map
```

```

type loc

val (=) (l1 l2: loc) : bool ensures { result <-> l1 = l2 }

val constant null : loc

type mem = { mutable next: loc -> loc }

val mem: mem

let cdr (p: loc) : loc
  requires { p <> null }
  ensures { result = mem.next p }
= mem.next p

let set_cdr (p: loc) (v: loc) : unit
  requires { p <> null }
  ensures { mem.next = (old mem.next)[p <- v] }
= let m = mem.next in
  mem.next <- fun x -> if x = p then v else m x

```

(Les définitions de prédicats, avec l'approche arithmétique *)*

```

predicate valid_cells (s: int -> loc) (n: int) =
  (forall i. 0 <= i < n -> s i <> null) &&
  (forall i j. 0 <= i < n -> 0 <= j < n -> i <> j -> s i <> s j)

predicate listLR (m: mem) (s: int -> loc) (l: loc) (lo hi: int) =
  0 <= lo <= hi &&
  if lo = hi then l = null else
    l = s lo && m.next (s (hi-1)) = null &&
    forall k. lo <= k < hi-1 -> m.next (s k) = s (k+1)

predicate listRL (m: mem) (s: int -> loc) (l: loc) (lo hi: int) =
  0 <= lo <= hi &&
  if lo = hi then l = null else
    m.next (s lo) = null && l = s (hi-1) &&
    forall k. lo < k < hi -> m.next (s k) = s (k-1)

predicate frame (m1 m2: mem) (s: int -> loc) (n: int) =
  forall p. (forall i. 0 <= i < n -> p <> s i) ->
    m1.next p = m2.next p

```

(Le code *)*

```

let list_reversal (ghost s: int -> loc) (ghost n: int) (l: loc) : (r: loc)
  requires { valid_cells s n }
  requires { listLR mem s l 0 n }

```

```

ensures { listRL mem s r 0 n }
ensures { frame mem (old mem) s n }
= let ref l = l in
  let ref r = null in
  let ghost ref i = 0 in
  while l <> null do
    invariant { if n = 0 then l = r = null else
      i = 0      && r = null      && l = s 0
      || i = n   && r = s (n-1) && l = null
      || 0 < i < n && r = s (i-1) && l = s i }
    invariant { listRL mem s r 0 i }
    invariant { listLR mem s l i n }
    invariant { frame mem (old mem) s n }
    variant { n - i }
    let tmp = l in
    l <- cdr l;
    set_cdr tmp r;
    r <- tmp;
    i <- i + 1
  done;
  return r

```

(* Avec la spécification récursive cette fois *)

```

let rec ghost predicate is_list (m: mem) (l: loc) (s: int -> loc) (n: int)
  requires { n >= 0 }
  variant { n }
= if n = 0 then l = null else
  l = s 0 <> null && is_list m (m.next l) (fun i -> s (i+1)) (n - 1)

let rec lemma cells_of_list (l: loc) (s: int -> loc) (n: int)
  requires { n >= 0 }
  requires { is_list mem l s n }
  variant { n }
  ensures { valid_cells s n }
  ensures { listLR mem s l 0 n }
= if n <> 0 then cells_of_list (cdr l) (fun i -> s (i+1)) (n - 1)

let rec lemma list_of_cells (r: loc) (s: int -> loc) (n: int)
  requires { n >= 0 }
  requires { valid_cells s n }
  requires { listRL mem s r 0 n }
  variant { n }
  ensures { is_list mem r (fun i -> s (n-1-i)) n }
= if n <> 0 then list_of_cells (cdr r) s (n - 1)

let list_reversal_final (ghost s) (ghost n: int) (l: loc) : (r: loc)
  requires { n >= 0 }

```

```
  requires { is_list mem l s n }
  ensures { is_list mem r (fun i -> s (n-1-i)) n }
  ensures { frame mem (old mem) s n }
= cells_of_list l s n;
  let r = list_reversal s n l in
  list_of_cells r s n;
  r
```

Démonstrations

GOOSE: an OCaml environment for quantum computing

Denis Carnier¹, Arthur Correnson²,
Christopher McNally³, and Youssef Moawad⁴

¹ imec-DistriNet, KU Leuven, Belgium
denis.carnier@kuleuven.be

² CISA Helmholtz Center for Information Security, Germany
arthur.correnson@cispa.de

³ Massachusetts Institute of Technology, United States
mcnallyc@mit.edu

⁴ University of Glasgow, United Kingdom
y.moawad.1@research.gla.ac.uk

Abstract

In this presentation, we showcase GOOSE: an OCaml library to model, simulate and compile low-level quantum programs. GOOSE is designed as a playground with an emphasis on extensibility and accessibility to non-experts. The library is compatible with the OPEN-QASM standard, and targets a variety of backends with a minimalistic, circuit-based IR.

1 Introduction

Quantum computing [8] is an emerging model of computation that exploits non-classical effects like *superposition* and *entanglement* to achieve algorithmic speedups. A radical break from classical computing, the model presents new challenges for programming languages research.

In particular, researchers are interested in more expressive ways to design and develop quantum algorithms, and the ability to leverage existing tooling for classical programming languages in the construction of quantum programming languages (QPLs) [1, 4, 9, 3, 6, 2].

OCaml is a mature ecosystem, widely used for programming languages research. However, to the best of our knowledge, it has a scarcity of tools for quantum programming. We therefore present GOOSE¹: an OCaml library to model, simulate and compile quantum programs. The library is designed to act as a playground to experiment with quantum programming in the OCaml ecosystem.

2 Description and architecture

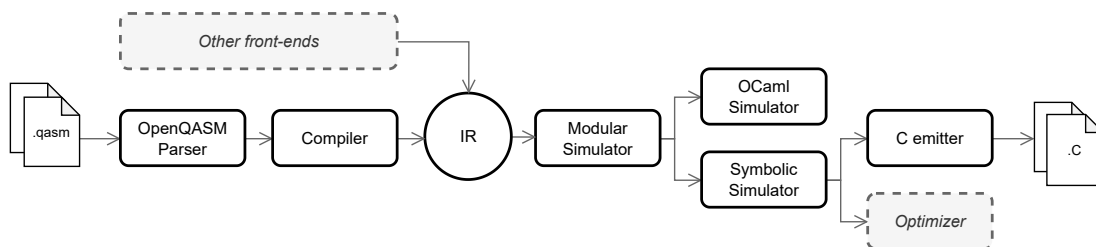


Figure 1: A high-level overview of GOOSE’s modular pipeline.

¹GOOSE is available at <https://qgoose.github.io>.

Figure 1 gives a high-level overview of GOOSE. The library is built around an intermediate representation (IR) for *quantum circuits*. This IR can be used as an entry point to, for instance, build simulators, compilers, and static analyzers for quantum programs. GOOSE includes a full state-based simulator to interpret the IR, designed as a functor parametrized by a linear algebra module. This design choice allows one to implement many tools simply by specializing the simulator. We demonstrate it by deriving both a symbolic simulator and an emitter to C. Further tools could be implemented, for example optimizers and compilers for other languages. To ensure compatibility with other existing tools related to quantum computing, we provide a frontend for OPENQASM 2.0, a standard low-level quantum circuit representation [5].

3 An example

In Figure 2, we provide an example of a GHZ [8] circuit entangling three qubits. Listing 3a shows the corresponding GOOSE IR representation. This is implemented with a list comprehension that specifies the *CX* gates, with an initial *H* gate. In Listing 3b, we display the C code that was generated based on the results of the symbolic simulator. `cadd`, `cmul`, and `csub` are C functions for complex number arithmetic. Finally, `s` and `o` are arrays for storing the input and output state, respectively.

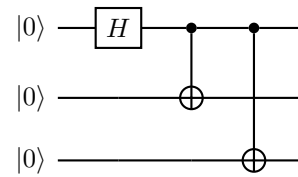


Figure 2: 3-qubit GHZ circuit.

```
let entanglement n = {
  qbits = n;
  gates = List.cons ({
    target = A 0;
    kind = H;
    controls = []
  }) (List.init (n-1)
    (fun i -> {
      target = A (i + 1);
      kind = X;
      controls = [A 0]
    }
  ))
}
```

(a) Explicit construction of a GHZ circuit in the GOOSE IR.

```
cfloat *s = (cfloat*)malloc(N*sizeof(cfloat));
cfloat *o = (cfloat*)malloc(N*sizeof(cfloat));

for (int i = 0; i < N; i++)
  s[i] = (cfloat) {0.0, 0.0};

o[0] = cmul(SQRT1_2, cadd(s[0], s[1]));
o[1] = cmul(SQRT1_2, csub(s[6], s[7]));
o[2] = cmul(SQRT1_2, cadd(s[2], s[3]));
o[3] = cmul(SQRT1_2, csub(s[4], s[5]));
o[4] = cmul(SQRT1_2, cadd(s[4], s[5]));
o[5] = cmul(SQRT1_2, csub(s[2], s[3]));
o[6] = cmul(SQRT1_2, cadd(s[6], s[7]));
o[7] = cmul(SQRT1_2, csub(s[0], s[1]));
```

(b) Sample of generated C code to simulate this circuit for $n = 3$. Each line computes one amplitude in the output state: $\frac{1}{\sqrt{2}}(1, 0, 0, 0, 0, 0, 0, 1)^T$

4 Conclusion and future work

We presented GOOSE: an open source library for quantum computing in the OCaml ecosystem. Currently, the library’s front-end supports a subset of the OPENQASM 2.0 circuit representation, a standard for interoperability between quantum computing tools. Additionally, GOOSE implements a generic simulator for quantum circuits. We specialized this simulator to both a symbolic simulator and a C emitter.

In the future, we want to connect GOOSE to other existing projects, for instance TWIST [10], a QPL whose existing OCaml-based interpreter calls out to C++ quantum simulation libraries, or VOQC [7], a formally verified compiler for quantum circuits that can be extracted to OCaml.

Acknowledgements

We would like to thank the anonymous reviewers who have helped to improve this paper through their criticism and suggestions. Moreover, we especially thank Paulette Koronkevich for inspiring the name GOOSE.

C.M. was supported by the CQE-LPS Doc Bedard Fellowship. This research is co-funded by a PhD studentship from the College of Science and Engineering at the University of Glasgow, by the Research Fund KU Leuven, by the Flemish Research Programme Cybersecurity and by the European Union (ERC, UniversalContracts, 101040088). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

- [1] Qiskit: An open-source framework for quantum computing.
- [2] Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. *arXiv:quant-ph/0409065*, April 2005.
- [3] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. Silq: A high-level quantum language with safe uncomputation and intuitive semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '20*, pages 286–300, New York, NY, USA, June 2020. Association for Computing Machinery.
- [4] Cirq Developers. Cirq, July 2018.
- [5] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open Quantum Assembly Language. *arXiv:1707.03429 [quant-ph]*, July 2017.
- [6] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper: A Scalable Quantum Programming Language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13*, pages 333–342, New York, NY, USA, June 2013. Association for Computing Machinery.
- [7] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. A verified optimizer for Quantum circuits. *Proceedings of the ACM on Programming Languages*, 5(POPL):37, January 2021.
- [8] M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.
- [9] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A practical quantum instruction set architecture, 2016.
- [10] Charles Yuan, Christopher McNally, and Michael Carbin. Twist: Sound reasoning for purity and entanglement in Quantum programs. *Proceedings of the ACM on Programming Languages*, 6(POPL):30:1–30:32, January 2022.

Caractériser des propriétés de confiance d'IA avec Why3

Julien Girard-Satabin, Michele Alberti, François Bobot, and Zakaria Chihani

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
julien.girard2@cea.fr
firstname.lastname@cea.fr

Résumé

Nous proposons de faire la démonstration de CAISAR, un logiciel libre en développement au sein du laboratoire de sûreté et sécurité des logiciels du CEA LIST. CAISAR est une plateforme permettant de caractériser la sûreté des logiciels résultant d'un protocole d'apprentissage (réseaux de neurones, machines à vecteur de support, forêts aléatoires). CAISAR se base sur la plateforme de vérification Why3 pour fournir un langage typé de modélisation de problèmes, ainsi qu'un moteur de raisonnement logique éprouvé. Cette approche autorise par exemple la vérification de systèmes composés de plusieurs blocs d'IA, ainsi qu'une meilleure expressivité des propriétés à vérifier. CAISAR est disponible à <https://git.frama-c.com/pub/caisar>.

Proposition de présentation

Contexte

Ces cinq dernières années ont vu naître le champ scientifique de la vérification formelle de programmes appris par réseaux de neurones. Ce champ est prospère : plus de vingt outils ont été ainsi développés sur cette période [12, 13, 21, 22, 18, 19, 16, 1, 11, 6, 14, 20, 7]. Ces travaux utilisent différentes techniques (interprétation abstraite, calcul SMT, programmation par contrainte), et utilisent souvent des heuristiques spécialisées sur des propriétés très spécifiques ; le plus souvent une invariance de la sortie par rapport à une perturbation de l'entrée circonscrite à un voisinage bien défini. Enfin, ces outils sont parfois limités quant aux programmes qu'ils peuvent vérifier, se limitant exclusivement aux réseaux de neurones convolutifs. Toutefois, nous observons que des idées développées dans certains outils percolent dans d'autres : ainsi, une heuristique spécifique faisant appel à des multiplicateurs de Lagrange dans l'outil *OVAL*¹ s'est ensuite vue réutilisée par ailleurs dans $\alpha\beta$ -*CROWN* [22] et *ERAN* [19].

L'évolution rapide de ce domaine peut se comprendre comme la recherche, l'évaluation et la sélection de bonnes techniques de vérification par la communauté. Toutefois, à mesure que les programmes appris se voient intégrés dans des systèmes industriels, les exigences de sûreté et de sécurité vont croissant. Comment vérifier un système composé de différents programmes d'IA hétérogènes (par exemple : enchaînement de réseaux et de machines à vecteur de support) ? Comment vérifier des propriétés de conformité par rapport à des exigences fournies ? Pour tenter de répondre à ces questions, nous avons développé une plateforme dédiée à la caractérisation de la robustesse et de la sûreté des systèmes à base d'intelligence artificielle, CAISAR [9].

Principes guidant la conception de CAISAR

L'objectif de CAISAR est de fournir une interface unifiée de modélisation de problème de vérification sur les programmes à base d'IA. Pour se faire, CAISAR se base sur la plateforme de vérification *Why3* [8] et son langage *WhyML*. CAISAR modélise les propriétés à vérifier via des prédicats *WhyML*.

1. <https://github.com/oval-group/oval-bab>

Une propriété classique dans la littérature [22, 18, 19, 3] est la robustesse locale de la prédiction par rapport à une perturbation locale, dont voici la définition. Soit un espace d'entrée \mathcal{X} (qu'on peut restreindre sans perte de généralité à un sous-ensemble de \mathbb{R}^d). Soit un programme $f : \mathcal{X} \mapsto \mathbb{R}$ associant à un échantillon $x \in \mathcal{X}$ une valeur réelle (par exemple, dans un problème de classification d'images, le programme associe à chaque échantillon une classe). Soit $\varepsilon \in \mathbb{R}$, et soit d une distance (on utilise typiquement les normes l_p pour $p \geq 1$). Un programme est alors localement robuste si, étant donné un échantillon $x_0 \in \mathcal{X}$:

$$\forall x, d(x, x_0) < \varepsilon \implies f(x) = f(x_0)$$

La Figure 1 montre comment écrire un prédicat WhyML pour modéliser cette propriété à vérifier, avec la norme l_∞ comme fonction de distance. CAISAR, s'inspirant de *Why3*, propose une interface de modélisation unifiée avant de décharger des obligations de preuves à différents prouveurs automatiques.

Quelques fonctionnalités

La plateforme gère principalement les réseaux de neurones sous Open Neural eXchange Format (ONNX)². Ce choix est motivé par l'interopérabilité observée entre différents cadres d'apprentissage automatique (PyTorch, TensorFlow, Keras, Scikit-Learn). Les machines à vecteur de support sont gérées par un format csv *ad-hoc*. Enfin, il est possible pour l'utilisateur de spécifier un jeu de données particulier sur lequel vérifier les propriétés, via un fichier csv contenant les valeurs de chaque point de donnée. Actuellement, seuls les jeux de données de problèmes de classification peuvent être spécifiés ainsi : il est nécessaire de fournir une annotation explicite de la classe attendue pour chaque point.

Elle dispose d'une représentation intermédiaire du réseau de neurone permettant une traduction directe du réseau sous forme de formule logique, extractible en SMTLIB [2] grâce à *Why3*. Cette représentation intermédiaire ouvre également la voie à différentes transformations pour rendre la vérification plus simple, telles que la partition des entrées en régions linéaires [10].

CAISAR gère différents outils de vérification :

1. le Support Vector Machine reachability analysis tool (*SAVer* [15]), spécialisé dans les machines à vecteur de support
2. tous les solveurs standards prenant en entrée du SMTLIB (dont *Alt-Ergo* [4] et *Z3* [5]); les solveurs spécialisés en réseaux de neurones tels que *nnenum* [1] supportent un format voisin, *VNN-Lib*³
3. le prouveur SMT *Marabou* [13] spécialisé dans le traitement des réseaux de neurones grâce à différentes optimisations d'algorithme de pivot et de parallélisme
4. un outil de test métamorphique pour les IA, *AIMOS*, développé en interne et pour l'instant en source fermée
5. l'analyseur abstrait *PyRAT*, également développé en interne <https://pyrat-analyzer.com/>

Déroulement de la démonstration

La démonstration consiste d'abord à présenter le contexte au sein duquel se place CAISAR, et à quel besoin il répond. Nous présenterons les outils qu'il supporte ; et la nature des programmes traités (type de programme, taille, composabilité). Nous procéderons également à un comparatif avec d'autres approches comparables à CAISAR, telles que *DNNV* [17] et le tout récent *Vehicle-lang*⁴. Nous ferons

2. <https://onnx.ai/>

3. <http://www.vnnlib.org/>

4. <https://github.com/vehicle-lang/vehicle>

ensuite une démonstration de la spécification d'un problème de validation sur plusieurs problèmes jouets, avec différentes propriétés, en détaillant quand il est possible de le faire les mécanismes sous-jacents de la génération d'obligations de preuves. Nous ouvrirons enfin sur de potentielles futures fonctionnalités de la plateforme.

```
theory DataSetClassification
```

```
  type features = array t
  type class = int
```

```
  type datum = (features, class)
  type label_ = int
```

```
  type datum = (features, label_)
```

```
  type dataset = {
    nb_features: int;
    nb_label_s: int;
    data: array datum
  }
```

```
  [...]
```

```
  type model = {
    nb_inputs: int;
    nb_outputs: int;
  }
```

```
  function predict: model → features → label_
  [...]
```

```
  predicate linfty_distance (a: features) (b: features) (eps: t) =
    a.length = b.length ∧
    forall i: int. 0 ≤ i < a.length → .- eps .< a[i] .- b[i] .< eps
```

```
  predicate correct_at (m: model) (d: datum) =
    let (x, y) = d in
    y = predict m x
```

```
end
```

FIGURE 1 – Exemple de prédicats définis dans la bibliothèque standard de CAISAR. La fonction `predict` spécifie l'application d'un modèle `m` sur des entrées `x`. Le prédicat `linfty_distance` définit la norme l_∞ , et `correct_at` est un prédicat qui vérifie si la prédiction d'un programme de classification est conforme à la vérité-terrain.

Remerciements

Le développement de CAISAR est partiellement soutenu par le programme Confiance.ai⁵, et le projet PRISMA⁶.

Références

- [1] Stanley Bak. Nnenum : Verification of ReLU Neural Networks with Optimized Abstraction Refinement. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez, editors, *NASA Formal Methods*, Lecture Notes in Computer Science, pages 19–36, Cham, 2021. Springer International Publishing.
- [2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [3] Marco Casadio, Ekaterina Komendantskaya, Matthew L Daggitt, Wen Kokke, Guy Katz, Guy Amir, and Idan Refaeli. Neural network robustness as a verification property : a principled case study. In *International Conference on Computer Aided Verification*, pages 219–231. Springer, 2022.
- [4] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-Ergo 2.2. In *SMT Workshop : International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018.
- [5] Leonardo de Moura and Nikolaj Bjørner. Z3 : An Efficient SMT Solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 337–340, Berlin, Heidelberg, 2008. Springer.
- [6] Souradeep Dutta, Susmit Jha, Sriram Sanakaranarayanan, and Ashish Tiwari. Output Range Analysis for Deep Neural Networks. *arXiv:1709.09130 [cs, stat]*, September 2017.
- [7] Ruediger Ehlers. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks. *arXiv:1705.01320 [cs]*, May 2017.
- [8] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - Where Programs Meet Provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 125–128, Berlin, Heidelberg, 2013. Springer.
- [9] Julien Girard-Satabin, Michele Alberti, François Bobot, Zakaria Chihani, and Augustin Lemesle. CAISAR : A platform for Characterizing Artificial Intelligence Safety and Robustness. In *AI Safety*, CEUR-Workshop Proceedings, Vienne, Austria, July 2022.
- [10] Julien Girard-Satabin, Aymeric Varasse, Marc Schoenauer, Guillaume Charpiat, and Zakaria Chihani. DISCO : Division of input space into convex polytopes for neural network verification. *JFLA*, 2021.
- [11] P Henriksen and A Lomuscio. Efficient Neural Network Verification via Adaptive Refinement and Adversarial Search. In *24th European Conference on Artificial Intelligence - ECAI 2020*, page 8, Santiago de Compostela, Spain, 2020.
- [12] Guy Katz, Clark Barrett, David Dill, Kyle Julian, and Mykel Kochenderfer. Reluplex : An efficient smt solver for verifying deep neural networks. *arXiv preprint arXiv:1702.01135*, 2017.
- [13] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljić, David L. Dill, Mykel J. Kochenderfer, and Clark Barrett. The Marabou Framework for Verification and Analysis of Deep Neural Networks. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, volume 11561, pages 443–452. Springer International Publishing, Cham, 2019.
- [14] Alessandro De Palma, Rudy Bunel, Alban Desmaison, Krishnamurthy Dvijotham, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Improved branch and bound for neural network verification via lagrangian decomposition.
- [15] Francesco Ranzato and Marco Zanella. Robustness verification of support vector machines. In *International Static Analysis Symposium*, pages 271–295. Springer, 2019.

5. <https://www.confiance.ai/>

6. <https://prisma.univ-gustave-eiffel.fr/>

- [16] Zhouxing Shi, Huan Zhang, Kai-Wei Chang, Minlie Huang, and Cho-Jui Hsieh. Robustness Verification for Transformers. In *International Conference on Learning Representations*, 2020.
- [17] David Shriver, Sebastian Elbaum, and Matthew B. Dwyer. DNNV : A Framework for Deep Neural Network Verification. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification*, Lecture Notes in Computer Science, pages 137–150, Cham, 2021. Springer International Publishing.
- [18] Gagandeep Singh, Timon Gehr, Matthew Mirman, Markus Püschel, and Martin Vechev. Fast and Effective Robustness Certification. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 10802–10813. Curran Associates, Inc., 2018.
- [19] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–30, 2019.
- [20] Caterina Urban, Maria Christakis, Valentin Wüstholtz, and Fuyuan Zhang. Perfectly Parallel Fairness Certification of Neural Networks. *arXiv:1912.02499 [cs]*, December 2019.
- [21] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Efficient Formal Safety Analysis of Neural Networks. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems 31*, pages 6367–6377. Curran Associates, Inc., 2018.
- [22] Shiqi Wang, Huan Zhang, Kaidi Xu, Xue Lin, Suman Jana, Cho-Jui Hsieh, and J. Zico Kolter. Beta-CROWN : Efficient Bound Propagation with Per-neuron Split Constraints for Complete and Incomplete Neural Network Robustness Verification.

Necro ML: Kit de Nécromancie (Démonstration)

Louis Noizet and Alan Schmitt

INRIA Rennes – Bretagne Atlantique

Abstract

On présente Necro ML, un outil pour générer des interpréteurs OCaml à partir de sémantiques en Skel, un langage minimaliste de description de sémantiques. Ces interpréteurs sont modulaires, en exploitant des monades pour interpréter les calculs.

Introduction

Ce travail est initialement basé sur les travaux de Courant *et al.* [3]. En particulier, le langage Skel a été largement amélioré, avec du polymorphisme, des fonctions de première classe, et du filtrage. L'interpréteur OCaml généré est maintenant modulaire pour l'interprétation des branches, ce qui permet d'en changer aisément. La section 1 présente le langage Skel. La section 2 décrit comment un interpréteur est généré. Enfin, la section 3 montre comment les monades d'interprétations peuvent être utilisées pour représenter différents comportements de l'interpréteur. Lors de la démonstration, nous présenterons le fonctionnement de Necro ML et l'usage de différentes monades d'interprétations pour traiter le non-déterminisme.

1 Skel

Skel [2] est un langage statiquement typé pour décrire des sémantiques opérationnelles de langages. Il est à la fois léger et puissant, et une sémantique squelettique peut notamment être extraite vers un générateur OCaml, une formalisation Coq, et un débogueur. Le langage Skel est basé sur ML avec une construction supplémentaire appelée le branchement pour traiter les calculs non-déterministes, et la possibilité de sous-spécifier.

Une des fonctionnalités principales de Skel est la séparation entre *squelettes* et *termes*. Les premiers représentent des calculs, et les seconds des valeurs. Cela est inspiré du λ -calcul computationnel de Moggi [6].

Skel est proche de langages ML comme OCaml ou SML, mais son AST est léger, de manière à pouvoir être manipulé plus simplement. Il est défini dans [7], et contient uniquement 126 lignes de spécification, ce qui permet de gérer et extraire facilement des sémantiques.

2 Necro ML

Necro est un écosystème pour manipuler les sémantiques squelettiques. Il repose sur Necro Lib [7], une bibliothèque OCaml qui définit la syntaxe de Skel, et un ensemble de fonctions pour manipuler les sémantiques squelettiques, y compris la fonction `parse_and_type`, qui lit un fichier et renvoie une sémantique squelettique typée.

Ainsi, n'importe qui a accès à la syntaxe et à des fonctions de base, pour pouvoir créer des outils qui exploitent les sémantiques squelettiques et étendent l'écosystème Necro. En plus de Necro Lib, nous fournissons plusieurs outils. L'un d'eux, celui que nous présentons ici, est Necro ML [5], qui génère des interpréteurs OCaml depuis des sémantiques squelettiques.

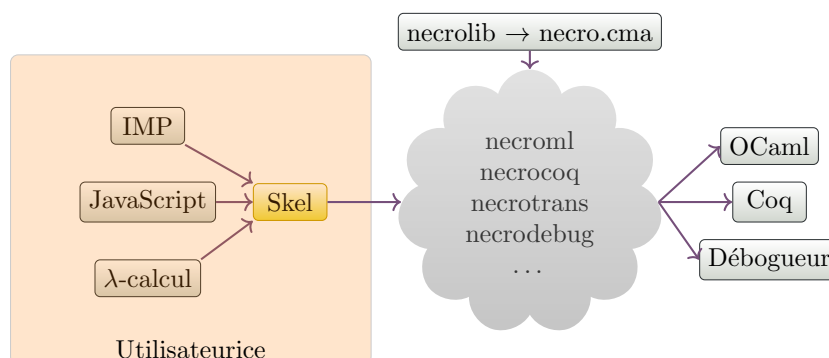


Figure 1: The Necro ecosystem

Comme nous l’avons précisé dans la section 1, il y a dans les sémantiques squelettiques des données non-spécifiées. Cela est traité par Necro ML de manière modulaire. On génère un foncteur `MakeInterpreter`, qui produit un interpréteur quand on lui fournit une implémentation des termes et types non-spécifiés.

Le plongement est principalement superficiel, mais on doit gérer les branches avec prudence, puisque OCaml n’a pas de construction native équivalente. Nous allons donc montrer comment cela est traité dans la section 3.

3 Monade d’interprétation

Comme expliqué plus haut, on ne peut pas utiliser un simple plongement profond, en raison des constructions non-déterministes. À cette fin, on utilise une monade de plongement, ou monade d’interprétation, pour décrire les calculs. Ainsi, les termes de type `'a` sont plongés superficiellement dans des expressions OCaml de type `'a`, tandis que les squelettes de type `'a` sont plongés profondément dans des expressions OCaml de type `'a M.t`. Necro ML propose plusieurs monades d’interprétations, et d’autres peuvent être définies par l’utilisatrice. L’aspect principal est qu’il est très simple de changer la monade d’interprétation sans avoir à réécrire aucun code.

4 Conclusion

Necro ML est un outil flexible pour générer des interpréteurs. Il effectue un plongement superficiel des types, et semi-profond des expressions pour gérer le non-déterminisme. Il a été testé sur des fichiers de taille significative, notamment dans le travail en cours de formalisation de JavaScript [4].

D’autres projets sont intimement liés. D’abord, Necro Debug est une exécution pas-à-pas d’une sémantique squelettique.¹ Il utilise la même approche et les mêmes signatures de modules que Necro ML, ce qui permet de réutiliser l’instanciation des termes non-spécifiés. Ensuite, Ambal *et al.* ont présenté un générateur d’interpréteurs certifiés à l’aide de Necro Coq et du mécanisme d’extraction [1].

¹https://skeletons.inria.fr/debugger/index_while.html

References

- [1] Guillaume Ambal, Sergueï Lenglet, and Alan Schmitt. Certified Abstract Machines for Skeletal Semantics. In *Certified Programs and Proofs*, Philadelphia, United States, January 2022.
- [2] Martin Bodin, Philippa Gardner, Thomas Jensen, and Alan Schmitt. Skeletal Semantics and their Interpretations. *Proceedings of the ACM on Programming Languages*, 44:1–31, 2019. URL: <https://hal.inria.fr/hal-01881863>, doi:10.1145/3290357.
- [3] Nathanaëlle Courant, Enzo Crance, and Alan Schmitt. Necro: Animating Skeletons. In *ML 2019*, Berlin, Germany, Aug 2019.
- [4] Adam Khayam, Louis Noizet, and Alan Schmitt. JSkel: Towards a Formalization of JavaScript’s Semantics, 2022. Submitted for publication.
- [5] Enzo Crance Martin Bodin, Nathanaëlle Courant and Louis Noizet. Necro Ocaml Generator. URL: <https://gitlab.inria.fr/skeletons/necro-ml>.
- [6] Eugenio Moggi. Computational lambda-calculus and monads. *Proceedings. Fourth Annual Symposium on Logic in Computer Science*, 1988.
- [7] Louis Noizet. Necro Library. URL: <https://gitlab.inria.fr/skeletons/necro>.

Auteurs

Abou Samra, Jean	7
Alberti, Michele	288
Baudart, Guillaume	24
Bertholon, Guillaume	43
Blaudeau, Clément	59
Bobot, François	288
Bodin, Martin	7
Boldo, Sylvie	2
Bourke, Timothy	101
Bussone, Grégoire	24
Butte, Elliot	172
Carnier, Denis	285
Charguéraud, Arthur	43
Chihani, Zakaria	288
Cohen, Cyril	3
Contejean, Évelyne	121
Correnson, Arthur	265, 285
Filliâtre, Jean-Christophe	274
Gardelle, Samuel	134
Girard-Satabin, Julien	288
Jensen, Thomas	152
Ledein, Amélie	172
Leroy, Xavier	4
Mandel, Louis	24
McNally, Christopher	285
Miné, Antoine	211
Miquey, Étienne	134
Moawad, Youssef	285
Monat, Raphaël	211
Noizet, Louis	293
Paskevich, Andrei	274
Pesin, Basile	101
Pouzet, Marc	101
Radanne, Gabriel	59
Rébiscoul, Vincent	152
Rémy, Didier	59
Samokish, Andrei	121
Scherer, Gabriel	190
Schmitt, Alan	152, 293
Tasson, Christine	24
Valnet, Milla	211
Wallez, Théophile	243
Zakowski, Yannick	7

Le comité d'organisation des JFLA 2023 remercie chaleureusement ses généreux mécènes.



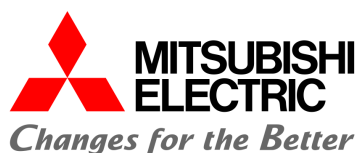
CEA List



Fondation OCaml



GDR GPL



Mitsubishi Electric



Nomadic Labs



OCamlPro



Tarides



TrustInSoft