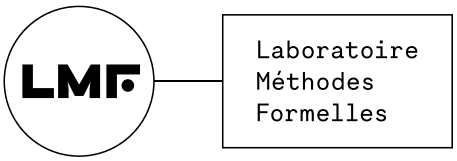
33èmes journées francophones des langages applicatifs

Chantal Keller, Timothy Bourke, Sandrine Blazy, Frédéric Bour, Guillaume Bury, Stefania Dumbrava, Diane Gallois-Wong, Adrien Guatto, David Janin, Marie Kerjean, et al.

33èmes Journées Francophones des Langages Applicatifs

du 28 juin au 1er juillet 2022 au Domaine d'Essendiéras (Périgord)



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 ainsi les aspects les plus théoriques de la conception des langages applicatifs jusqu'à ses applications industrielles. La $33^{\text{ème}}$ édition des JFLA se déroule au Domaine d'Essendiéras, à Saint-Médard-d'Excideuil, dans le Périgord.

Cette année, nous avons sélectionné 10 articles de recherche, 2 articles de recherche courts, et 9 démonstrations. Le programme de ces JFLA aborde de nombreuses thématiques, couvrant un large panel de la programmation à la vérification et la preuve formelle. La programmation fonctionnelle sera à l'honneur, avec l'optimisation de code OCaml et la compilation de programmes OCaml vers FPGA, ainsi que la programmation synchrone, avec l'étude de sémantiques opérationnelles et l'inférence parallèle. Un langage appelé Catala permet de lier législation et programmation. Le système de paquet opam, relié à Software Heritage, permet de conserver les développements OCaml. Côté vérification dynamique, seront présentés Osnap, une bibliothèque de snaphot testing ; Alt-Ergo-Fuzz, un testeur pour le prouveur SMT Alt-Ergo ; ainsi qu'un vérificateur d'assertions arithmétiques formalisé. Concernant la vérification statique, nous pourrons découvrir un système de types permettant d'inférer statiquement des tailles de tableaux ; la vérification de programmes concurrents avec d'une part la formalisation des "futurs" et "promesses" en Viper, et d'autre part Steel, une logique concurrente pour les programmes F* ; et un outil permettant de décrire les scénarios du protocole de consensus de la blockchain Tezos. Du côté des outils de preuve formelle, on pourra découvrir des améliorations de l'assistant de preuve Coq, comme l'automatisation avec les outils trakt et Sniper, l'amélioration des notations pour les nombres grâce à la commande Number Notation, ainsi qu'Actema, une interface graphique et gestuelle pour preuves formelles. Un nouvel outil à but pédagogique de preuve par induction dans des systèmes de transition nommé Mikino sera également présenté. Enfin, concernant les formalisations, seront présentés une vérification de l'arithmétique flottante employée dans un solveur de contraintes ; une formalisation des mathématiques discrètes en Coq combinées à la génération d'un livre ; et une traduction du framework sémantique K vers le framework logique Dedukti.

Cette sélection a été faite par les membres du comité de programme, que nous remercions chaleureusement, à partir des 23 soumissions. Nous tenons aussi à exprimer nos vifs remerciements aux rapporteurs externes au comité de programme : Boubacar Sall, Germán Andrés Delbianco, Guillaume Claret et Nicolas Ayache.

En plus du programme sélectionné, nous aurons le grand plaisir d'écouter un cours invité de Delphine Demange sur l'enseignement de la programmation fonctionnelle, un cours invité de Denis Mérigoux sur Rust, ainsi qu'un exposé invité de Matthias Puech – qui nous fera également l'honneur de donner un concert – sur musique et programmation fonctionnelle.

Nous espérons que cette édition permettra de se retrouver après l'édition précédente à distance, dans l'ambiance studieuse mais néanmoins conviviale qui caractérise les JFLA.

Programmatiquement vôtre,

Chantal Keller et Timothy Bourke.

Comité de programme

Chantal Keller	LMF, Université Paris-Saclay
Timothy Bourke	Inria, ÉNS de Paris
Sandrine Blazy	Irisa, Université Rennes 1
Frédéric Bour	Tarides - Inria
Guillaume Bury	OcamlPro
Stefania Dumbrava	Samovar, ENSIIE, Télécom Sud Paris
Diane Gallois-Wong	Nomadic Labs
Adrien Guatto	IRIF, Université de Paris
David Janin	LaBRI, Université de Bordeaux
Marie Kerjean	LIPN, Université Paris 13
Luc Pellissier	LACL, Université Paris-Est Créteil
Mário Pereira	NOVA-LINCS, Universidade Nova de Lisboa
Alix Trieu	Aarhus University
Yannick Zakowski	LIP, Inria, ÉNS de Lyon

Rapporteurs externes

Boubacar Sall
Germán Andrés Delbianco
Guillaume Claret
Nicolas Ayache

Table des matières

Démonstrations de prototypes

Exposés et cours invités

Si2-FIP: Programmation Fonctionnelle en Licence 1 avec Scala

Cours invité

Delphine Demange

Irisa, Université Rennes 1

Résumé

Nous présentons un retour d'expérience sur un module d'enseignement de la programmation fonctionnelle en Scala, proposé ces dernières années en Licence 1 à l'Université de Rennes 1. Ce module s'adresse à un public grand débutant, qui découvre le paradigme de programmation fonctionnelle, et doit s'efforcer de déconstruire ses habitudes de programmation impérative. De fait, ce public est encore non acquis à une approche méthodique de la programmation.

Dans une première partie, nous décrivons le positionnement du module dans la formation de Licence d'Informatique, ainsi que les besoins et enjeux associés, et les choix pédagogiques qui en ont résulté. Les notions enseignées dans ce module sont : la spécification et la modélisation adéquate de problèmes et des données associées, les types de données algébriques, la récursivité, l'ordre supérieur, le polymorphisme paramétrique, et les types abstraits.

En seconde partie, nous illustrons de manière concrète et pratique, à l'aide d'un mini-projet proposé aux étudiants en fin de module, comment nos choix pédagogiques ont été mis en œuvre. Le projet consiste à programmer une application réactive avec interface graphique, manipulant des images représentées en interne par des arbres quaternaires, et offrant quelques transformations de base.

Nous terminons par quelques éléments de réflexion pédagogique, en tirant les leçons des succès et des écueils rencontrés, et par un temps d'échange autour des besoins et des pratiques pédagogiques de la communauté.

Rust pour le formaliste impatient

Cours invité

Denis Merigoux

Inria

Résumé

Il est devenu impossible d'éviter en ligne les bruyants fans du langage de programmation Rust ces dernières années. En effet, celui-ci a été voté comme langage "le plus adoré" par Stackoverflow depuis 6 années consécutives. En tout cas, Rust a le potentiel pour plaire à la communauté des JFLAs puisqu'il possède de nombreuses fonctionnalités héritées de la programmation fonctionnelle, tout en étant orienté vers la programmation système. Mais alors, est-ce que Rust est vraiment meilleur que C++ ? Est-il possible de faire segfaulter un programme Rust ? Qu'est-ce que l'emprunt et la possession de variables ? Comment vaincre le compilateur Rust et enfin se débarrasser de ses très longs messages d'erreur ?

Pour répondre à toutes ces questions, ce cours vous proposera deux sessions d'une heure trente avec un pré-requis d'un niveau M2 en méthodes formelles, afin d'aller directement au cœur du sujet et de tenter une approche fonctionnelle et formelle à la description du langage. La première session introduira les bases de la syntaxe et des fonctionnalités, notamment les concepts clés d'emprunt de possession qui permettent à Rust de garantir la sûreté mémoire statiquement. La deuxième session sera consacrée à un état de l'art de la formalisation et la vérification de programmes en Rust, dans laquelle des travaux francophones sont bien placés. Enfin, pour celles et ceux qui voudraient entrer dans l'écosystème Rust pour jouer formellement avec les programmes, un tutoriel vous apprendra à vous connecter au compilateur de Rust et récupérer les arbres de syntaxes abstraits et informations de typage dont vous auriez besoin pour vos propres projets.

Pour pouvoir utiliser Rust pendant le cours, vous pouvez l'installer en suivant ces instructions : https://www.rust-lang.org/tools/install.

Développement d'outils audionumériques pour la création électroacoustique

Exposé invité

Matthias Puech

INA GRM

Résumé

En marge des systèmes de types, des preuves formelles et des systèmes de réécriture, je vous inviterai dans cet exposé ludique et largement non-scientifique à adopter un nouveau hobby : la programmation audio temps-réel embarquée.

J'exposerai ici par la pratique mon activité de lutherie électronique, de la caractérisation d'un besoin musical jusqu'à la production : recherche, conception, développement logiciel et électronique, et production industrielle d'instruments et de traitements audio à destination des compositeurs et interprètes de musiques électro-acoustiques. Nous nous attarderons bien sûr — nous sommes aux JFLA — sur leur programmation, embarquée, optimisée et temps réel «dur», et l'attention particulière qu'elle exige : connaissance des caractéristiques d'exécution du processeur, du comportement du compilateur, optimisation du WCET, maîtrise précise des ressources, calculs lors de la compilation, etc.

Après un survol du contexte artistique dans lequel nous nous placerons et un bref rappel de traitement du signal numérique, nous explorerons le chemin de l'information audio à travers un système embarqué typique, des doigts du musicien jusqu'aux haut-parleurs ; nous détaillerons les principes et enjeux de la programmation d'un tel système temps-réel, et les techniques pratiques (ici en C++) qui rendront possible l'implémentation d'un petit instrument autonome.

Articles longs

Inférence parallèle pour un langage réactif probabiliste

Guillaume Baudart,[1,2] Louis Mandel,[3]
Marc Pouzet,[2,1] Reyyan Tekin[1,2]

[1] Inria Paris
[2] École normale supérieure – PSL university
[3] IBM Research

Résumé

ProbZelus est un langage synchrone probabiliste qui permet de décrire des modèles probabilistes réactifs en interaction avec un environnement observable. Des méthodes d'inférences réactives permettent d'apprendre en ligne les distributions associées aux paramètres non-observés du modèle à partir d'observations statistiques. Ce problème n'a, en général, pas de solution analytique simple. Pour obtenir des estimations précises, une méthode classique consiste à analyser les résultats obtenus par de nombreuses exécutions indépendantes du modèle. Ces méthodes sont coûteuses, mais ont l'avantage de pouvoir être massivement parallélisées.

Nous proposons d'utiliser JAX pour paralléliser les méthodes d'inférence réactive de ProbZelus. JAX est une bibliothèque récente qui permet de paralléliser automatiquement du code Python qui peut ensuite être exécuté de manière efficace et transparente sur CPU, GPU, ou TPU.

Dans cet article, nous décrivons un nouveau moteur d'inférence réactive parallèle implémenté en JAX et le nouveau *backend* JAX associé pour ProbZelus. Nous montrons sur des exemples existants que notre nouvelle implémentation surpasse l'implémentation séquentielle originale pour un nombre de particules élevé.

1 Introduction

Les langages synchrones [4] ont été introduits pour concevoir et implémenter des systèmes embarqués temps réel. Ils permettent de décrire une spécification exécutable précise qui est utilisée comme référence pour la simulation, le test et la vérification formelle. Le compilateur génère ensuite le code embarqué qui est *correct par construction*, c'est à dire qu'il préserve la sémantique de la spécification initiale. Le langage synchrone Scade [13] est ainsi utilisé pour le logiciel certifié d'avions et de trains, par exemple.

La plupart des systèmes embarqués évoluent dans un environnement ouvert et incertain qu'il ne perçoivent qu'à travers des capteurs imparfaits et bruités (accéléromètres, caméras ou GPS). Les interactions avec des agents autonomes (animaux, humains, robots) ajoutent encore à cette incertitude. Les langages synchrones classiques ne permettent pas de manipuler ces incertitudes.

La programmation probabiliste est un paradigme de programmation qui a connu un essor important ces dernières années. Les langages de programmation probabilistes permettent de décrire des modèles qui manipulent l'incertitude de manière explicite et proposent des méthodes automatiques pour inférer les paramètres du modèle à partir d'observations statistiques. Ces langages reposent sur la méthode Bayésienne qui permet de raffiner une croyance a priori sur la distribution des paramètres d'un modèle à partir d'observations concrètes [6, 18, 20, 24, 25]. Dans la lignée de ces travaux, nous avons récemment proposé ProbZelus, un langage synchrone probabiliste [2]. ProbZelus combine les constructions d'un langage réactif synchrone (temps logique synchrone et automates hiérarchiques) et les constructions probabilistes (`sample`, `observe` et `infer`) pour concevoir des applications réactives probabilistes [1].

L'exécution d'un programme probabiliste nécessite la résolution d'un problème d'inférence. Ce problème n'a en général, pas de solution analytique simple. Pour obtenir des estimations précises, une méthode classique consiste à analyser les résultats obtenus par de nombreuses exécutions indépendantes du modèle, appelées *particules* [15]. Ces méthodes sont coûteuses mais ont l'avantage de pouvoir être massivement parallélisées.

Le moteur d'inférence de ProbZelus est implémenté en OCaml qui se prête mal à la parallélisation massive. Dans cet article nous proposons d'utiliser JAX pour paralléliser le moteur d'inférence de ProbZelus. JAX est une bibliothèque récente qui permet de compiler du code purement fonctionnel qui peut ensuite être exécuté de manière transparente sur CPU, GPU, ou TPU de manière massivement parallèle [10]. JAX est associé à une impressionnante bibliothèque de calcul numérique qui permet de ré-implementer efficacement le moteur d'inférence. D'autres fonctionnalités, telle que l'auto-différenciation, pourront également être utilisée dans le futur pour implémenter des méthodes d'inférence plus avancées.

Dans cet article, nous montrons comment exécuter des modèles probabilistes réactifs de manière massivement parallèle grâce à JAX. Nous présentons, en particulier, les contributions suivantes : Section 3 un nouveau moteur d'inférence parallèle efficace implémenté en JAX, Section 4 un compilateur de ProbZelus vers JAX , Section 5 une évaluation sur un ensemble de modèles réactifs existants. Le code est disponible à l'adresse suivante : https://github.com/rpl-lab/jfla22-zlax.

2 Contexte

Dans cette section nous rappelons les éléments fondamentaux de la programmation synchrone et de la programmation probabiliste à travers un exemple simple (voir [1] pour une introduction plus complète).

2.1 Programmation réactive probabiliste

Programmation synchrone. ProbZelus est une extension du langage Zelus [1] [9] avec des constructions probabilistes. Zelus est lui-même un descendent de Lustre [19] dont il reprend les principes et le style de programmation *flot de données*. Un programme Zelus est défini par un ensemble de fonctions de suites, appelées *nœuds*. Les entrées et les sorties d'un nœuds sont des suites infinies, appelées des *flots*. Toutes ces suites progressent au même rythme, de manière *synchrone*. Ce type de programmation permet de décrire naturellement les schémas-blocs de l'automatique, la notation classique utilisée dans la conception du logiciel réactif embarqué [21].

Par exemple, le nœud suivant lève une alarme dès que l'entrée Booléenne devient vraie.

```
node watch x = alarm where
  rec automaton
  | Wait → do alarm = false unless x then Ring
  | Ring → do alarm = true done
```

Le nœud `watch` calcule le flot de sortie `alarm` à partir du flot d'entrée `x`. Le comportements de `watch` est défini par un automate à deux états. Dans l'état initial `Wait` la valeur de `alarm` est `false` à tous les instant. Dès que `x` devient `true`, la transition `unless x` est activée, et l'automate passe dans l'état `Ring`. La valeur de `alarm` devient alors `true` à tous les instants. La Figure 1 montre un exemple de trace d'exécution du nœud `watch`.

1. https://zelus.di.ens.fr

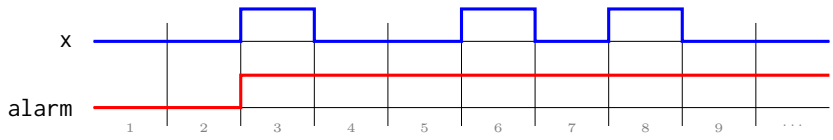


FIGURE 1 – Exemple de chronogramme pour le nœud `edge`.

Programmation probabiliste. L'inférence Bayésienne permet de calculer la distribution d'un paramètre θ sachant une série d'observations $\mathbf{x}$ (distribution *a posteriori* $p(\theta \mid \mathbf{x})$) à partir d'une croyance initiale (distribution *a priori* $p(\theta)$).

$$p(\theta \mid \mathbf{x}) = \frac{p(\theta)\, p(\mathbf{x} \mid \theta)}{p(\mathbf{x})} \qquad \text{(Bayes, 1763)}$$

Les langages de programmation probabilistes introduisent donc trois constructions pour décrire les modèles : `theta = sample(d)` introduit une variable aléatoire `theta` de distribution a priori `d`, $p(\theta)$; `observe(d, x)` mesure la vraisemblance de l'observation `x` par rapport à une distribution `d` (i.e., la valeur de la densité de `d` en `x`), $p(x \mid \theta)$; `infer m x` calcule la distribution a posteriori des valeurs de sortie d'un modèle `m` étant donné les observations `x`, $p(\theta \mid x)$.

Un exemple introductif classique consiste à déterminer le biais d'une pièce à partir d'observations indépendantes. À chaque instant $n \in \mathbb{N}$, on observe le résultat d'un lancé : $x_n = \mathit{True}$ (pile) ou $x_n = \mathit{False}$ (face). On suppose que chacun de ces lancés suit une loi de Bernoulli de paramètre θ : $p(x_n \mid \theta) = \theta$ et $p(\overline{x_n} \mid \theta) = 1 - \theta$. On cherche à estimer le biais θ à partir des observations $(x_n)_{n\in\mathbb{N}}$, i.e., $p(\theta \mid x_0, x_1, x_2, \dots)$.

Le programme suivant implémente ce modèle en ProbZelus.

```
proba coin x = theta where
  rec init theta = sample (uniform_float (0., 1.))
  and () = observe (bernoulli theta, x)
```

Initialement, on ne sait rien sur la pièce, tous les biais sont équivalents. La distribution a priori est donc uniforme sur $[0, 1]$ ($\theta = 0.5$ correspond à une pièce parfaitement équilibrée, $\theta = 0$ est une pièce qui tombe toujours sur face). On suppose que le paramètre θ est constant (mot-clé `init`). À chaque instant on utilise la construction `observe` pour faire l'hypothèse que l'observation `x` suit une loi de Bernoulli de paramètre θ.

2.2 Inférence dans la boucle

La construction `infer` est un nœud d'ordre supérieur qui prend ici en argument le nœud probabiliste `coin` et un flot d'entrées. Il renvoie à chaque instant, la distribution de sortie du modèle. L'inférence est un processus synchrone qui ne s'arrête jamais et qui peut être exécutée *dans la boucle*, i.e., en parallèle, avec d'autres nœuds synchrones déterministes classiques.

Par exemple, le programme suivant combine le modèle `coin` et le nœud `watch` pour signaler une pièce suspecte.

```
node cheater_detector x = cheater where
  rec theta_dist = infer coin x
  and m, s = stats_float theta_dist
  and cheater = watch ((m < 0.2 || 0.8 < m) && (s < 0.01))
```

Le flot `theta_dist` correspond à la distribution estimée du paramètre `theta` sachant les premières observations $x_0, x_1, \dots, x_n$. À chaque instant, on calcule la moyenne et l'écart type de

cette distribution. Le nœud `watch` définit ci-dessus surveille ces valeurs et lève l'alarme `cheater` dès que la condition `(m < 0.2 || m > 0.8) && (s < 0.01)` est vérifiée.

2.3 ProbZelus

La syntaxe, le typage, et la sémantique formelle d'un noyau de ProbZelus sont formellement définit dans [2]. Nous en rappelons ici l'essentiel.

Syntaxe. La syntaxe du noyau de ProbZelus est la suivante. Les constructions manquantes comme les automates hiérarchiques peuvent être compilé vers ce noyau par une série de transformations source à source.

$$
\begin{array}{rcl}
d & ::= & \texttt{let node}\ f\ x\ \texttt{=}\ e \mid \texttt{let proba}\ f\ x\ \texttt{=}\ e \mid d\ d \\
e & ::= & c \mid x \mid \texttt{(}e\texttt{,}e\texttt{)} \mid op\texttt{(}e\texttt{)} \mid f\texttt{(}e\texttt{)} \mid \texttt{last}\ x \mid e\ \texttt{where rec}\ E \\
 & & \mid \texttt{present}\ e\ \texttt{->}\ e\ \texttt{else}\ e \mid \texttt{reset}\ e\ \texttt{every}\ e \\
 & & \mid \texttt{sample(}e\texttt{)} \mid \texttt{observe(}e\texttt{,}e\texttt{)} \mid \texttt{infer(}f\texttt{(}e\texttt{))} \\
E & ::= & x\ \texttt{=}\ e \mid \texttt{init}\ x\ \texttt{=}\ c \mid E\ \texttt{and}\ E
\end{array}
$$

Un programme est une suite de déclaration de fonctions de flots déterministes (`node`) et probabilistes (`proba`). Le corps d'une fonction de flot est défini par une expression. Un expression de base peut être une constante (c), une variable (x), une paire ((e,e)), l'application d'un opérateur primitif (opérateur arithmétique, distribution, etc.), un appel de fonction de flot, un délai ($\texttt{last}\ x$) qui renvoie la valeur de x à l'instant précédent, ou une définition locale ($e\ \texttt{where rec}\ E$). Une équation $x\ \texttt{=}\ e$ définit la variable x à partir de l'expression e et une équation $\texttt{init}\ x\ \texttt{=}\ e$ définit la valeur initiale de x. Une expression peut également être une structure de contrôle : $\texttt{present}\ x\ \texttt{->}\ e_1\ \texttt{else}\ e_2$ active l'expression e_1 ou e_2 selon la valeur de x, $\texttt{reset}\ e_1\ \texttt{every}\ e_2$ réinitialise la valeur des expressions $\texttt{last}\ x$ avec la valeur des équations $\texttt{init}\ x\ \texttt{=}\ e$ correspondantes. Enfin, le noyau déterministe du langage est étendu avec les expressions probabilistes $\texttt{sample(}e\texttt{)}$, $\texttt{observe(}e_1\texttt{,}e_2\texttt{)}$, et $\texttt{infer(}f\texttt{(}e\texttt{))}$.

Ordonnancement. En ProbZelus, dans l'expression $e\ \texttt{where rec}\ E$, E est un ensemble d'équations mutuellement récursives. Au moment de la compilation, les équations sont simplifiées et ordonnées selon leurs dépendances. Les initialisations $\texttt{init}\ x_j\ \texttt{=}\ c_j$ sont regroupées au début et l'équation $x_j = e_j$ doit apparaître après l'équation $x_i = e_i$ si l'expression e_j utilise x_i en dehors d'un opérateur `last`. Le compilateur peut introduire des équations supplémentaires pour relâcher les contraintes d'ordonnancement. Les programmes qui ne peuvent pas être ordonnancés statiquement sont rejetés (cette décision est prise par le compilateur de Zelus à l'aide d'un système de types dédiés qui construit une signature exprimant les dépendances entre entrées/sorties d'un noeud [3]). Dans la suite, on fera donc l'hypothèse que le noyau ProbZelus est ordonnancé. L'expression $e\ \texttt{where rec}\ E$ prend donc la forme suivante :

$$
\begin{array}{l}
e\ \texttt{where rec init}\ x_1\ \texttt{=}\ c_1\ \ldots\ \texttt{and init}\ x_k\ \texttt{=}\ c_k \\
\qquad \texttt{and}\ y_1\ \texttt{=}\ e_1\ \ldots\ \texttt{and}\ y_n\ \texttt{=}\ e_n
\end{array}
$$

Pour simplifier, on suppose aussi que toutes les variables introduites par `init` sont également définies par une équation, i.e., $\{x_i\}_{1..k} \cap \{y_j\}_{1..n} = \{x_i\}_{1..k}$. Il est toujours possible d'ajouter des équations $x_i\ \texttt{=}\ \texttt{last}\ x_i$ si ce n'est pas le cas.

Sémantique. La sémantique idéale de ProbZelus est définie de manière co-itérative [11]. Les expressions déterministes sont caractérisées par un état initial de type S et une fonction de

transition de type $S \to T \times S$ qui prend en argument l'état courant, renvoie une sortie et l'état suivant. Le flot correspondant est obtenu en itérant la fonction de transition à partir de l'état initial. Pour un environnement γ qui associe les noms de variables à leur valeur et une expression déterministe e, on note $[\![e]\!]^{\text{init}}_\gamma$ l'état initial et $[\![e]\!]^{\text{step}}_\gamma$ la fonction de transition. Par exemple, l'accès à une variable et la construction `present` sont définis de la manière suivante :

$$
\begin{array}{lcl}
[\![x]\!]^i_\gamma & = & () \\
[\![x]\!]^s_\gamma & = & \lambda s.\ (\gamma(x), s) \\
[\![\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2]\!]^{\text{init}}_\gamma & = & ([\![e]\!]^{\text{init}}_\gamma, [\![e_1]\!]^{\text{init}}_\gamma, [\![e_2]\!]^{\text{init}}_\gamma) \\
[\![\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2]\!]^{\text{step}}_\gamma & = & \\
\multicolumn{3}{l}{\quad \lambda(s, s_1, s_2).\ \mathit{let}\ v, s' = [\![e]\!]^{\text{step}}_\gamma(s)\ \mathit{in}} \\
\multicolumn{3}{l}{\quad\quad \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ v_1, s'_1 = [\![e_1]\!]^{\text{step}}_\gamma(s_1)\ \mathit{in}\ (v_1, (s', s'_1, s_2))} \\
\multicolumn{3}{l}{\quad\quad\quad \mathit{else}\ \ \mathit{let}\ v_2, s'_2 = [\![e_2]\!]^{\text{step}}_\gamma(s_2)\ \mathit{in}\ (v_2, (s', s_1, s'_2))}
\end{array}
$$

La fonction de transition d'une variable renvoie à chaque instant la valeur stockée dans l'environnement. La construction `present` e `->` e_1 `else` e_2 renvoie la valeur de e_1 quand e est vrai, et la valeur de e_2 dans le cas contraire. L'état contient l'état des trois sous-expressions. La fonction de transition exécute paresseusement l'une ou l'autre branche en fonction de la valeur de e.

Les expressions probabilistes sont définies par un état initial et une fonction de transition de type $S \to \Sigma_{T\times S} \to [0, \infty)$ qui prend en argument l'état courant et renvoie une *mesure* qui associe un score positif à tous les ensembles mesurables de couples (sortie, état suivant). [2] Pour un environnement γ et une expression probabiliste e, on note $\{\!|e|\!\}^{\text{init}}_\gamma$ l'état initial, et $\{\!|e|\!\}^{\text{step}}_\gamma$ la fonction de transition.

L'opérateur `infer` permet d'obtenir une valeur déterministe (une distribution) à partir d'un modèle probabiliste f et d'un flot d'observations e. À chaque instant, cet opérateur calcule une distribution de résultats et une distribution d'états suivants possibles.

$$
\begin{array}{lcl}
[\![\texttt{infer}(f(e))]\!]^{\text{init}}_\gamma & = & \lambda U.\ \delta_{[\![f(e)]\!]^i_\gamma}(U) \\
[\![\texttt{infer}(f(e))]\!]^{\text{step}}_\gamma & = & \lambda\sigma.\ \mathit{let}\ \mu = \lambda U.\ \int_S \sigma(ds) \{\!|f(e)|\!\}^{\text{step}}_\gamma(s)(U)\ \mathit{in} \\
 & & \quad \mathit{let}\ \nu = \lambda U.\ \mu(U)/\mu(\top)\ \mathit{in} \\
 & & \quad (\pi_{1*}(\nu), \pi_{2*}(\nu))
\end{array}
$$

Pour un environnement γ, l'état initial de $[\![\texttt{infer}(f(e))]\!]^{\text{init}}_\gamma$ est la mesure de Dirac sur l'état initial de $f(e)$. La fonction de transition intègre la mesure définit par $f(e)$, notée $\{\!|f(e)|\!\}^{\text{step}}_\gamma$, sur tous les états possibles pour obtenir une mesure μ. Cette mesure est ensuite normalisée pour obtenir une distribution $\nu : T \times S$ `dist` ($\top$ représente l'espace entier). Cette distribution est ensuite séparée en une paire de distribution marginales sur les résultats et les états suivants avec les mesures images par les projections π_1 et π_2.

3 Moteur d'inférence

En général, estimer la distribution a posteriori d'un modèle n'admet pas de solution analytique simple. Les langages de programmation probabiliste reposent donc sur des méthodes d'inférence approximative. Les méthodes de Monte-Carlo consistent à accumuler les résultats d'un grand nombre de simulations indépendantes pour obtenir une estimation précise [15].

2. $\Sigma_{T\times S}$ est la σ-algèbre sur $T \times S$, c'est à dire l'ensemble des ensembles mesurables de $T \times S$.

$$
\begin{aligned}
&\{\!|c|\!\}^{\text{step}}_{\gamma} && = \lambda s, w.\ (c, s, w)\\
&\{\!|x|\!\}^{\text{step}}_{\gamma} && = \lambda s, w.\ (\gamma(x), s, w)\\
&\{\!|\texttt{sample(}e\texttt{)}|\!\}^{\text{step}}_{\gamma} && = \lambda s, w.\ \mathit{let}\ d, s', w' = \{\!|e|\!\}^{\text{step}}_{\gamma}(s, w)\ \mathit{in}\ (\text{draw}(d), s', w')\\
&\{\!|\texttt{observe(}e_1\texttt{,}e_2\texttt{)}|\!\}^{\text{step}}_{\gamma} && = \lambda s, w.\\
& && \quad \mathit{let}\ \mu, s_1, w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma}(s, w)\ \mathit{in}\\
& && \quad \mathit{let}\ v, s_2, w_2 = \{\!|e_2|\!\}^{\text{step}}_{\gamma}(s_1, w_1)\ \mathit{in}\ ((), s_2, w_2 * \mu_{\text{pdf}}(v))\\
&\{\!|\texttt{present}\ e\ \texttt{->}\ e_1\ \texttt{else}\ e_2|\!\}^{\text{step}}_{\gamma} && = \lambda(s, s_1, s_2), w.\\
& && \quad \mathit{let}\ v, s', w' = \{\!|e|\!\}^{\text{step}}_{\gamma}(s, w)\ \mathit{in}\\
& && \quad\ \mathit{if}\ v\ \mathit{then}\ \mathit{let}\ v_1, s_1', w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma}(s_1, w')\ \mathit{in}\ (v_1, (s', s_1', s_2), w_1)\\
& && \quad\quad \mathit{else}\ \ \mathit{let}\ v_2, s_2', w_2 = \{\!|e_2|\!\}^{\text{step}}_{\gamma}(s_2, w')\ \mathit{in}\ (v_2, (s', s_1, s_2'), w_2)\\
&\{\!|\texttt{reset}\ e_1\ \texttt{every}\ e_2|\!\}^{\text{step}}_{\gamma} && = \lambda(s_0, s_1, s_2), w.\\
& && \quad \mathit{let}\ v_2, s_2', w_2 = \{\!|e_2|\!\}^{\text{step}}_{\gamma}(s_2, w)\ \mathit{in}\\
& && \quad \mathit{let}\ v_1, s_1', w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma}(\mathit{if}\ v_2\ \mathit{then}\ (s_0, w_2)\ \mathit{else}\ (s_1, w_2))\ \mathit{in}\\
& && \quad (v_1, (s_0, s_1', s_2'), w_1)\\
&\left\{\!\!\left|\begin{array}{l} e\ \texttt{where}\\ \quad \texttt{rec init}\ x_1\ \texttt{=}\ c_1\ \ldots\\ \quad \texttt{and init}\ x_k\ \texttt{=}\ c_k\\ \quad \texttt{and}\ y_1\ \texttt{=}\ e_1\ \ldots\\ \quad \texttt{and}\ y_n\ \texttt{=}\ e_n \end{array}\right|\!\!\right\}^{\text{step}}_{\gamma} && = \begin{array}{l} \lambda((m_1, \ldots, m_k), (s_1, \ldots, s_n), s), w.\\ \quad \mathit{let}\ \gamma_1 = \gamma[m_1/x_1\texttt{\_last}]\ \mathit{in}\ \ldots\\ \quad \mathit{let}\ \gamma_k = \gamma_{k-1}[m_k/x_k\texttt{\_last}]\ \mathit{in}\\ \quad \mathit{let}\ v_1, s_1', w_1 = \{\!|e_1|\!\}^{\text{step}}_{\gamma_k}(s_1, w)\ \mathit{in}\ \mathit{let}\ \gamma_1' = \gamma_k[v_1/y_1]\ \mathit{in}\ \ldots\\ \quad \mathit{let}\ v_n, s_n', w_n = \{\!|e_n|\!\}^{\text{step}}_{\gamma_{n-1}'}(s_n, w_{n-1})\ \mathit{in}\ \mathit{let}\ \gamma_n' = \gamma_{n-1}'[v_n/y_n]\ \mathit{in}\\ \quad \mathit{let}\ v, s', w' = \{\!|e|\!\}^{\text{step}}_{\gamma_n'}(s)\ \mathit{in}\\ \quad (v, ((\gamma_n'[x_1], \ldots, \gamma_n'[x_k]), (s_1', \ldots, s_n'), s'), w') \end{array}
\end{aligned}
$$

FIGURE 2 – Sémantique opérationnelle des expressions probabilistes.

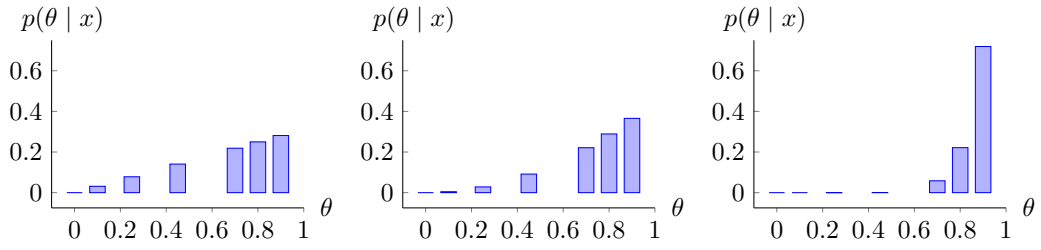
3.1 Échantillonneur

Pour ce type d'algorithme d'inférence, la sémantique opérationnelle d'un modèle probabiliste est un échantillonneur. Les états initiaux ne changent pas, mais la fonction de transition produit un échantillon aléatoire associé à un score qui mesure la qualité de l'échantillon.

La fonction de transition d'une expression probabiliste de type $S \times [0, \infty) \to T \times S \times [0, \infty)$ prend en argument un état et un score et renvoie l'état suivant, la sortie, et le nouveau score. Les fonctions de transition des expressions probabilistes de ProbZelus sont présentées en Figure 2. Les constantes, et l'accès à une variable (initialisée ou non) renvoient la valeur attendue et laissent l'état et le score inchangés. `sample(d)` tire une valeur aléatoire dans la distribution `d` sans modifier le score. `observe(d, x)` multiplie le score par la vraisemblance de l'observation `x` par rapport à la distribution `d` et renvoie une valeur de type `unit`.[3] Pour les autres constructions, l'état contient les états de toutes les sous-expressions. La fonction de transition se contente de calculer l'état suivant et de propager le score en suivant l'ordre des sous-expressions.

L'état d'un ensemble d'équations récursives (ordonnancé) e `where rec` E contient, en plus de l'état des sous-expressions, la valeur des variables locales à l'instant précédent $(m_1, \ldots, m_k)$. La fonction de transition commence par mettre à jour l'environnement γ avec un ensemble de variables fraîches x_i`_last` initialisées avec les valeurs m_i. Cet environnement est ensuite étendu par les définitions des variables y_i en exécutant toutes les sous-expressions tout en propageant le score. Enfin, l'expression e est exécutée dans l'environnement final. L'état suivant contient la nouvelle valeur des variables initialisées qui sera utilisée pour démarrer l'instant suivant.

3. On note D_{pdf} la densité de la distribution D.



(a) instant 1 (b) instant 2 (c) instant 10

FIGURE 3 – Échantillonnage préférentiel avec 7 particules. On observe pile/`true` à tout les instants. Au cours du temps, les valeurs de θ proches de 1 deviennent de plus en plus probables.

3.2 Échantillonnage préférentiel

À partir d'un échantillonneur, la méthode d'inférence la plus simple possible lance N exécutions indépendantes, appelés *particules*. À chaque instant, chaque particule exécute un pas de l'échantillonneur pour calculer un triplet (résultat, état suivant, score). Les scores sont ensuite normalisés pour obtenir une distribution catégorique, i.e., une distribution discrète sur les paires (résultat, état).

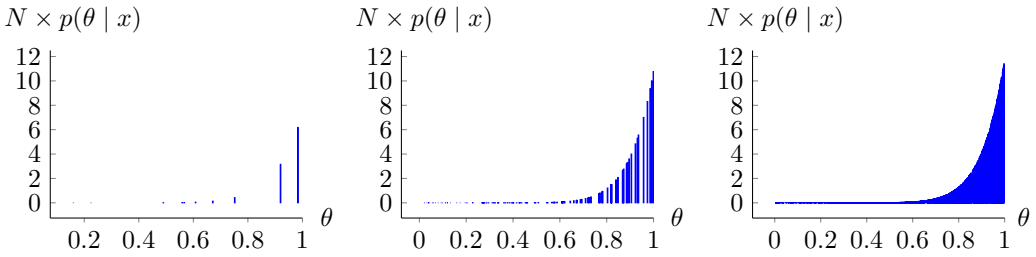
$$
\begin{aligned}
[\![\texttt{infer}(f(e))]\!]_\gamma^{\text{init}} &= [([\![f(e)]\!]_\gamma^{\text{init}}, 1)]_{1\leq i\leq N} \\
[\![\texttt{infer}(f(e))]\!]_\gamma^{\text{step}} &= \lambda s.\ let\ \left[(o_i, s_i', w_i') = let\ s_i, w_i = s[i]\ in\ \{\!|f(e)|\!\}_\gamma^{\text{step}}(s_i, w_i)\right]_{1\leq i\leq N}\ in \\
&\quad let\ \mu = \lambda U.\ \textstyle\sum_{1\leq i\leq N} \overline{w_i'} * \delta_{o_i}(U)\ in \\
&\quad \mu, [(s_i', w_i')]_{1\leq i\leq N}
\end{aligned}
$$

L'état de l'opérateur `infer` est donc un tableau initialisé avec N copies de l'état initial de l'échantillonneur associé au score initial 1. À chaque instant, chaque particule récupère son état courant et son score dans le tableau pour exécuter un pas de l'échantillonneur. Les scores sont ensuite normalisés pour obtenir la distribution de résultats μ, et le tableau est mis à jour pour l'instant suivant. On note $\overline{w_i} = w_i / \sum_{i=1}^N w_i$ les poids normalisés. Comparé à la sémantique idéale de `infer` de la Section 2.3, l'échantillonnage préférentiel approche l'intégrale incalculable par une somme discrète sur le tableau de particules.

Reprenons l'exemple `coin` de la Section 2.1. À l'instant initial, la première équation `init theta = sample (uniform_float (0., 1.))` tire un ensemble de valeurs possibles pour `theta`. Puis, à chaque instant, la première équation ne change plus (opérateur `init`), mais la seconde équation `() = observe (bernoulli theta, x)` met à jour le score pour chacune des valeurs possibles pour `theta` avec la formule suivante $w' = w * Bernoulli(\theta)_{\text{pdf}}(x)$. On peut enfin normaliser ces scores pour obtenir la distribution de sorties à chaque instant.

La Figure 3 illustre une exécution possible dans le cas où les valeurs observées sont toujours pile/`true`. On constate que les valeurs de `theta` les plus proche de 1 sont les plus probables, ce qui correspond bien aux premières observations.

La Figure 4 illustre la précision de l'échantillonnage préférentiel pour un nombre croissant de particules sur l'exemple précédent. On observe que la précision de la distribution obtenue dépend du nombre de particules utilisées. Sur cet exemple simple, 1000 particules suffisent pour obtenir une approximation raisonnable de la valeur théorique : $Beta(1, 10+1)$. Pour des modèles plus compliqués, où les paramètres cherchés sont de plus grande dimension, le nombre de particules nécessaires peut croître très rapidement. On peut cependant exploiter le fait que les particules sont indépendantes les unes des autres pour paralléliser le moteur d'inférence.



(a) 10 particules (b) 100 particules (c) 1000 particules

FIGURE 4 – Précision de l'échantillonnage préférentiel en fonction du nombre de particules N après 10 instants. On observe pile/`true` à tous les instants.

3.3 Parallélisation avec JAX

JAX[4] est une bibliothèque récente qui permet d'exécuter du code écrit dans un sous-ensemble de Python purement fonctionnel sur CPU, GPU ou TPU de manière transparente pour l'utilisateur [10]. Au moment de l'exécution, un compilateur *juste-à-temps* (JAT) spécialise les fonctions Python qui sont ensuite envoyées à XLA,[5] le compilateur haute-performance de Google pour GPU et TPU.

Opérateur `vmap`. JAX offre en particulier un opérateur `vmap` qui permet de vectoriser automatiquement une fonction pour l'exécuter de manière massivement parallèle. Pour les structures de données arborescentes simples, cet opérateur préserve la structure des entrées/sorties et ne vectorise que les feuilles. Par exemple, pour une fonction telle que `f(0) = ((1, 2), 3)` on aura :

```
vmap(f)([0, 0, 0]) = (([1, 1, 1], [2, 2, 2]), [3, 3, 3]).
```

Plus généralement, si les entrées sont décrites par le type `float t_in` et les sorties par le type `float t_out`, le type de `vmap` est le suivant :

```
val vmap: (float t_in → float t_out) → float array t_in → float array t_out
```

Dans notre contexte, cette propriété est fondamentale pour vectoriser l'état des particules qui stocke l'état interne de toutes les sous-expressions sous la forme de tuples imbriqués (cf. Section 2.3). En utilisant l'opérateur `vmap`, on peut ainsi implémenter un nœud ProbZelus d'ordre supérieur `zmap` qui exécute en parallèle N instances d'un nœud de la manière suivante.

```
from jax import vmap
class zmap(Node):
  def __init__(self, f, n):
      self.f = f()
      self.n = n

  def init(self)
      s_init = vmap(lambda _: init(self.f))(np.empty(self.n))
      return s_init
```

4. https://github.com/google/jax
5. https://www.tensorflow.org/xla

```
    def step(self, s, i):
        o, s = vmap(step(self.f))(s, i)
        return s, o
```

Les nœuds ProbZelus implémentent la classe `Node` qui impose la définition des deux méthodes `init` et `step`. L'état initial est obtenu en vectorisant sur un tableau de taille n une fonction qui renvoie l'état initial de `f` quel que soit son argument. La fonction de transition de `zmap` vectorise l'exécution de la fonction de transition de `f` sur l'état courant `s` et un tableau d'entrées `i`. `init` et `step` sont des fonctions génériques qui appellent les méthodes correspondantes.

L'implémentation de l'échantillonnage préférentiel suit le même schéma. L'état initial contient N copies de l'état initial de `f` associées à un score initial de 1. La fonction de transition déconstruit l'état courant en une paire (états, scores) qui permet de vectoriser l'exécution de l'échantillonneur. Les résultats sont ensuite normalisés pour obtenir une distribution.

```
class infer_importance(Node):
    def __init__(self, f, n):
        self.f = f()
        self.n = n

    def init(self):
        s_init = vmap(lambda _: (init(self.f), 1.0))(np.empty(self.n))
        return s_init

    def step(self, s, obs):
        s_in, w_in = s
        o, s_out, w_out = vmap(step(self.f))(s_in, w_in, obs)
        mu = normalize(o, w_out)
        return mu, (s_out, w_out)
```

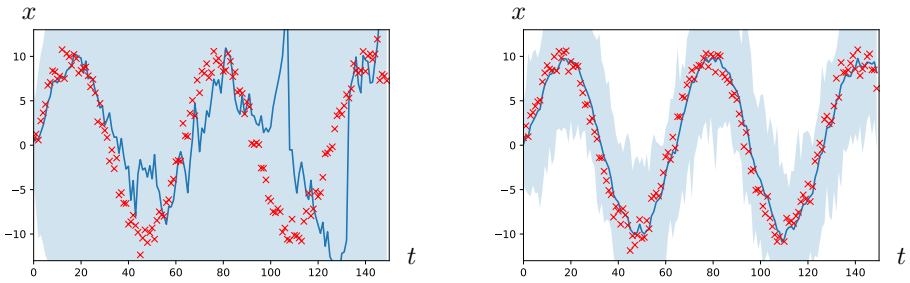
Remarque. Le générateur aléatoire de JAX qui permet d'échantillonner une valeur dans une distribution (utilisé pour la construction `sample`) prend explicitement une source aléatoire en argument : la *clé*. Nous avons ignoré ces clés pour simplifier la présentation. En pratique, les fonctions de transition prennent un argument supplémentaire *key* qui peut être regroupé avec le score pour former un argument unique *proba* qui contient toute l'information nécessaire pour les fonctions de transition probabilistes (cf. Section 4.1).

3.4 Filtre particulaire

L'échantillonnage préférentiel est une méthode relativement efficace pour inférer des paramètres constants à partir d'une série d'observations, comme dans l'exemple de la pièce `coin`. Malheureusement cette méthode montre rapidement ses limites pour des modèles où les paramètres peuvent changer au cours du temps. Par exemple, le modèle suivant implémente un modèle de Markov caché qui estime la position courante `x` à partir d'observations bruitées `obs`.

```
proba hmm obs = x where rec x = sample (gaussian (0. → pre x), speed)
                        and () = observe (gaussian (x, noise), obs)
```

À chaque instant, on suppose que la position courante n'est pas trop loin de la position précédente, i.e., suit une distribution normale centrée sur `0. → pre x` (la position précédente initialisée à 0). On impose également à chaque instant que la position estimée ne soit pas trop loin de l'observation courante, i.e., l'observation `obs` suit une distribution normale centrée sur `x`. `speed` et `noise` sont des constantes globales.



(a) échantillonnage préférentiel (b) filtre particulaire

FIGURE 5 – Résultat de l'inférence pour le modèle `tracker` avec 100 particules sur un sinus bruité pour 150 instants. Les points rouges sont les observations, la ligne bleue est la moyenne, la zone bleue contient les valeurs de toutes les particules (intervalle min/max à chaque instant).

Sur ce type de modèle, l'échantillonnage préférentiel correspond à une marche aléatoire : à chaque instant, la valeur courante de `x` est tirée aléatoirement à partir de la valeur précédente. La probabilité qu'une trajectoire aléatoire correspond à une série d'observations tend rapidement vers 0. Les estimations sont donc inexploitables après 2 ou 3 instants (Figure 5a).

Ré-échantillonnage. Pour résoudre ce problème, un *filtre particulaire* introduit une étape de ré-échantillonnage. Le nombre de particules reste constant au cours de l'exécution, mais à chaque instant, les particules les moins probables sont écartés et les particules les plus probables sont dupliqués. En d'autres termes, le tableaux de particules est recentré sur les observations.

$$
\begin{aligned}
[\![\texttt{infer}(f(e))]\!]_\gamma^{\text{init}} &= [([\![f(e)]\!]_\gamma^{\text{init}}, 1)]_{1\le i\le N} \\
[\![\texttt{infer}(f(e))]\!]_\gamma^{\text{step}} &= \lambda s.\ let\ \left[(o_i, s_i', w_i') = let\ s_i, w_i = s[i]\ in\ \{\!|f(e)|\!\}_\gamma^{\text{step}}(s_i, w_i)\right]_{1\le i\le N}\ in \\
&\quad let\ \mu = \lambda U.\ \textstyle\sum_{1\le i\le N} \overline{w_i'} * \delta_{(o_i,s_i)}(U)\ in \\
&\quad \pi_{1*}(\mu), [(\text{draw}(\pi_{2*}(\mu)), 1)]_{1\le i\le N}
\end{aligned}
$$

La sémantique de l'opérateur `infer` suit de prêt celle de l'échantillonnage préférentiel. À chaque instant, chaque particule exécute un pas de l'échantillonneur. Les scores sont ensuite normalisés pour obtenir une distribution sur les paires (résultat, nouvel état). Comme pour la sémantique idéale présentée en Section 2.3, on sépare ensuite cette distribution pour obtenir une paire de distributions. Le nouvel état est obtenu en ré-échantillonnant N valeurs dans la distribution d'états possibles associé à un score réinitialisé à 1.

L'implémentation de cette méthode d'inférence est très proche du nœud `infer_importance`. On utilise à nouveau l'opérateur `vmap` pour vectoriser l'exécution de l'échantillonneur. La fonction `normalize` renvoie une distribution catégorique qui peut être échantillonnée en utilisant la méthode `sample`.

```
class infer_pf(Node):
    def __init__(self, f, n):
        self.f = f()
        self.n = n

    def init(self):
        s_init = vmap(lambda _: (init(self.f), 1.0))(np.empty(self.n))
        return s_init
```

```
def step(s, obs):
    o, s_out, w_out = vmap(f_step)(s, np.ones(self.n), obs)
    mu = normalize(o, w_out)
    s_out = normalize(s_out, w_out).sample(sample_shape=self.n)
    return mu, s_out
```

La Figure 5b montre les résultats du filtre particulaire sur le modèle `hmm`. On observe qu'on obtient cette fois des résultats satisfaisants pour un modèle probabiliste réactif, où les flots de paramètres peuvent évoluer au cours du temps.

Il reste maintenant à comprendre comment compiler les modèles ProbZelus vers du code Python/JAX qui peut être appelé par les nœuds `infer_importance` et `infer_pf`. Dans la section suivante nous présentons un nouveau *backend* JAX pour ProbZelus.

4 Compiler ProbZelus vers JAX

Traditionnellement, les compilateurs de langages synchrones transforment des programmes à flot de données en programmes impératifs. La mémoire nécessaire à l'exécution du programme est allouée statiquement avant l'exécution. Les mises à jour sont réalisées par effet de bord. Ce schéma de compilation, bien adapté aux langages bas-niveau utilisés pour les systèmes embarqués, garantit une exécution en mémoire bornée. Malheureusement, JAX n'accepte que du code purement fonctionnel, plus facilement parallélisable.

Nous fondons notre travail sur le compilateur Zelus [8] qui a l'architecture suivante. (1) Le programme source est d'abord analysé et réécrit par des transformations successives dans un sous-ensemble du langage source. Parmi les analyses et transformations il y a, par exemple, le typage de données, l'analyse de causalité, la compilation des automates, la mise sous forme normale des équations, l'élimination de sous-expressions communes, ou la suppression de code mort. (2) Les équations mutuellement récursives sont ensuite ordonnancées pour satisfaire les dépendances de données (cf. Section 2.3) et le programme est traduit en OBC, un langage impératif intermédiaire (3) Enfin, le code OBC est compilé en OCaml impératif.

Pour bénéficier le plus possible de cette chaîne de compilation pour le nouveau *backend* vers JAX, nous remplaçons seulement la dernière étape. À partir du code OBC, nous le traduisons dans un langage μF purement fonctionnel que nous pouvons ensuite compiler en JAX. [6]

4.1 Langages intermédiaires

Le langage OBC. Le langage intermédiaire OBC (pour Object Based Code) a été introduit pour compiler un langage synchrone semblable à Lustre vers du code impératif (e.g., C) [5]. Il est utilisé par les compilateurs Heptagon [16], Vélus [7] et Zelus (avec quelques spécificités pour chacun de ces compilateurs). OBC permet de représenter une fonction de flot par un état et un ensemble de fonctions de transition agissant sur cet état et le modifiant en place. La syntaxe du langage étendue avec les constructions probabilistes est la suivante :

6. Dans [2], μF permet d'exprimer la sémantique mais ne correspond pas au code généré.

```
program ::= d*
d       ::= machine proba? m =
              memory (x,...,x)
              instances (o : m,...,o : m, o : infer(m),...,o : infer(m))
              reset() = S
              step(p) returns(p) = S
S       ::= var x in S | x := e | state(x) := e | S ; S | skip
          | match x with | C -> S ... | C -> S | o.reset | p := o.step(e)
          | p := o.pstep(e) | p := sample(e) | observe(e,e)
e       ::= c | x | state(x) | op(e)
p       ::= x | (p,p)
```

Un programme est une suite de déclarations de machines (ou classes). L'annotation optionnelle `proba` indique un modèle probabiliste. Une machine m est composée de quatre champs: (1) `memory` l'état de la fonction de transition, (2) `instances` les machines utilisées dans m, (3) `reset` la méthode de réinitialisation de l'état, et (4) `step` la fonction de transition qui prend une entrée, met à jour l'état et renvoie une sortie. Les instances sont annotées par le type m de la machine ou `infer(`m`)` s'il s'agit d'une instance de l'opérateur `infer`.

Une instructions peut être une déclaration de variable mutable locale (`var` x `in` S), une mise à jour d'une variable locale (x `:=` e), une mise à jour d'une variable d'état (`state(`x`) :=` e), une séquence d'instruction (S `;` S), une instruction sans effets (`skip`), une structure de contrôle (`match`/`with`), ou l'appel des méthodes (`step` or `reset`) d'une instance. OBC est étendu avec les instructions probabilistes `sample`, `observe`, et une méthode `pstep` qui correspond à l'appel de la méthode `step` d'une machine probabiliste. Une expression peut être une constantes (c), l'accès à une variable local (x), l'accès à une variable d'état (`state(`x`)`), ou l'application d'un opérateur primitif.

Le langage μF. μF est un simple langage purement fonctionnel sans ordre supérieur [2]. La syntaxe de μF est définie par la grammaire suivante :

```
program ::= d*
d       ::= val p = e | val m = stream { init = e ; step(p,p) = e }
e       ::= c | x | (e,e) | op(e) | f(e) | match x with | C -> e ... | C -> e
          | let p = e in e | init(m) | unfold(x,e)
          | sample(proba,e) | observe(proba,e,e) | infer(m)
```

Un programme est une suite de déclaration de valeurs et de fonctions de suites (`stream`). Une fonction de suite m est composée d'un état initial (`init`) et d'une fonction de transition (`step`). Une fonction de transition prend en argument un état et une entrée et renvoie une sortie et un nouvel état. Une expression peut être une constante, une variable, une paire, l'application d'un opérateur, un appel de fonction, une conditionnelle, ou une définition locale. L'expression `init(`m`)` renvoie une nouvelle instance m où l'état de l'instance est l'état initial. L'expression `unfold(`x`,`e`)` exécute un pas de l'instance x et renvoie l'élément suivant du flot et une nouvelle instance avec un état mis à jour.

La fonction de transition d'une machine probabiliste ajoute dans ses entrées/sorties une valeur *proba* qui représente l'état du modèle probabiliste, par exemple le score pour les méthodes d'inférence particulaire, ou les clés pour le générateur aléatoire de JAX (cf. Section 3.3). Cette valeur *proba* est passée en argument, mise à jour et renvoyée par les opérateurs probabilistes `sample` et `observe`. Enfin, `infer` correspond à l'instanciation de l'opérateur d'inférence sur un modèle probabiliste.

$$
\begin{array}{rcl}
\mathcal{C}_p(\texttt{skip}) & = & p \\
\mathcal{C}_p(x \texttt{ := } e) & = & \texttt{let } x \texttt{ = } \mathcal{C}(e) \texttt{ in } p \\
\mathcal{C}_p(\texttt{state}(x) \texttt{ := } e) & = & \texttt{let } x \texttt{ = } \mathcal{C}(e) \texttt{ in } p \\
\mathcal{C}_p(\texttt{var } x \texttt{ in } S) & = & \texttt{let } (p,x) \texttt{ = } \mathcal{C}_{(p,x)}(S) \texttt{ in } p \\
\mathcal{C}_p(S_1 \texttt{ ; } S_2) & = & \texttt{let } p \texttt{ = } \mathcal{C}_p(S_1) \texttt{ in } \mathcal{C}_p(S_2) \\
\mathcal{C}_p(\texttt{match } e \texttt{ with} & = & \texttt{match } \mathcal{C}(e) \texttt{ with} \\
\quad \texttt{|} C_1 \texttt{ -> } S_1 \ldots \texttt{|} C_n \texttt{ -> } S_n) & & \quad \texttt{|} C_1 \texttt{ -> } \mathcal{C}_p(S_1) \ \ldots \texttt{|} C_n \texttt{ -> } \mathcal{C}_p(S_n) \\
\mathcal{C}_p(o.\texttt{reset}) & = & \texttt{let } o \texttt{ = init}(m) \texttt{ in } p \qquad \text{si } o : m \\
\mathcal{C}_p(o.\texttt{reset}) & = & \texttt{let } o \texttt{ = infer}(m) \texttt{ in } p \qquad \text{si } o : \texttt{infer}(m) \\
\mathcal{C}_p(p' \texttt{ := } o.\texttt{step}(e)) & = & \texttt{let } (p',o) \texttt{ = unfold}(o,\mathcal{C}(e)) \texttt{ in } p \\
\mathcal{C}_p(p' \texttt{ := } o.\texttt{pstep}(e)) & = & \texttt{let } ((p',proba),o) \texttt{ = unfold}(o,(proba,\mathcal{C}(e))) \texttt{ in } p \\
\mathcal{C}_p(\texttt{sample}(proba,e)) & = & \texttt{let } (p',proba) \texttt{ = sample}(proba,\mathcal{C}(e)) \texttt{ in } p \\
\mathcal{C}_p(\texttt{observe}(proba,e_1,e_2)) & = & \texttt{let } proba \texttt{ = sample}(proba,\mathcal{C}(e)) \texttt{ in } p
\end{array}
$$

FIGURE 6 – Compilation de OBC vers μF.

4.2 Compilation

Compilation de OBC vers μF. La fonction de compilation rend explicite la manipulation de l'état pour transformer le code OBC impératif en code fonctionnel.

Considérons la machine m suivante :

$$
\begin{array}{l}
\texttt{machine } m \texttt{ =} \\
\quad \texttt{memory } (x_1,\ldots,x_n) \\
\quad \texttt{instances } (o_1 : m_1,\ldots,o_t : m_t, o_{t+1} : \texttt{infer}(m_{t+1}),\ldots,o_k : \texttt{infer}(m_k)) \\
\quad \texttt{reset}() \texttt{ = } S \\
\quad \texttt{step}(p) \texttt{ returns}(p) \texttt{ = } S
\end{array}
$$

La compilation d'une telle machine m produit une déclaration de fonction de suites (`stream`) et se décompose en deux étapes: la compilation de la méthode `reset` pour produire le champs `init`, et la compilation de la méthode `step` pour produire le champs `step`.

$$
\begin{array}{rcl}
\mathcal{C}(\texttt{reset}() \texttt{ = } S) & = & \texttt{init = } \mathcal{C}_{x_1,\ldots,x_n,o_1,\ldots,o_k}(S) \\
\mathcal{C}(\texttt{step}(p_{input}) \texttt{ returns}(p_{output})) & = & \\
\multicolumn{3}{r}{\texttt{step}(x_1,\ldots,x_n,o_1,\ldots,o_k,p_{input}) \texttt{ = } \mathcal{C}_{(p_{output},x_1,\ldots,x_n,o_1,\ldots,o_k)}(S)}
\end{array}
$$

Pour machine probabiliste, la variable *proba* est ajouté en entrée/sortie de la fonction `step`.

$$
\begin{array}{rcl}
\mathcal{C}(\texttt{step}(p_{input}) \texttt{ returns}(p_{output})) & = & \\
\multicolumn{3}{r}{\texttt{step}(x_1,\ldots,x_n,o_1,\ldots,o_k,(proba,p_{input})) \texttt{ = } \mathcal{C}_{((p_{output},proba),x_1,\ldots,x_n,o_1,\ldots,o_k)}(S)}
\end{array}
$$

La fonction de compilation des instructions $\mathcal{C}_p(S)$ est définie Figure 6. Cette fonction est paramétrée par l'ensemble p des variables potentiellement mises à jour par l'évaluation de l'instruction S et renvoie la valeur de ces variables. La compilation de `skip` renvoie simplement p. La compilation de l'affectation d'une variable ($\mathcal{C}_p(x \texttt{ := } e)$) respecte l'invariant que la variable x fait parti des variables de p et donc l'expression $\texttt{let } x \texttt{ = } \mathcal{C}(e) \texttt{ in } p$ masque la valeur précédente de x et retourne sa nouvelle valeur. La compilation de la méthode $o.\texttt{reset}$ instancie un nouvel état. Selon la déclaration de l'instance o, l'état est alloué avec l'opérateur `init` ou `infer`. La compilation de l'appel de machines probabilistes et des opérateurs `sample` et `observe` rend explicite l'état probabiliste qui est mis à jour par chacune de ces opérations. La fonction de compilation des expressions $\mathcal{C}$ correspond globalement à l'identité. On peut simplement remarquer que les variables d'état et locales sont traitées de manière similaire en μF: $\mathcal{C}(\texttt{state}(x)) = x$.

Remarque. Cette fonction de compilation génère beaucoup de `let` imbriqués et de copies de variables. Une passe par réécriture source à source est appliquée pour simplifier le code généré.

Compilation de μF vers JAX A partir du code μF purement fonctionnel, il est maintenant possible de générer du code Python compatible avec JAX : les `stream`s sont compilés en classes où les champs `init` et `step` deviennent des méthodes.

Afin d'éviter les problèmes de portée de variables en Python, une passe de compilation réécrit le code μF sans définitions imbriquées avant la traduction vers Python. La compilation est ensuite assez directe. La construction `match` est traduit vers des conditionnelles JAX qui prennent en arguments des fermetures pour contrôler leur évaluation. Enfin, pour permettre la parallélisation automatique, les structures de données doivent être sérialisables. Comme l'état des programmes ProbZelus respecte une structure arborescente simple, on peut utiliser le décorateur `@register_pytree_node_class` pour générer automatiquement les fonctions de sérialisation et déserialisation.

Par exemple, le code généré pour le modèle `coin` de la section Section 2 est le suivant :

```
@register_pytree_node_class
class coin(Node):
    def init(self):
        return {"theta": 42.0, "first": True}

    def step(self, *args):
        (state, (proba, obs)) = args
        def _ft(_):
            (proba, theta) = sample(proba, uniform_float(0.0, 1.0))
            return (proba, {**state, "theta": theta})
        def _ff(_):
            return (proba, state)
        (proba, state) = cond(state["first"], _ft, _ff, None)
        state = {**state, "first": False}
        (proba, ()) = observe(proba, bernoulli(state["theta"], obs)
        return (state, (proba, state["theta"]))
```

5 Évaluation

Nous avons évalué les performances de notre nouveau moteur d'inférence pour ProbZelus sur un ensemble d'exemples qui illustrent plusieurs aspects du langage [2]. Notre évaluation se concentre sur les deux questions suivantes : (**QR1**) Quel est l'impact du nouveau moteur d'inférence sur la précision des estimations ? (**QR2**) La parallélisation automatique permet-elle d'améliorer les performances ?

5.1 Méthode expérimentale

Pour chacun des exemples, nous mesurons le temps d'exécution et la précision obtenu après 500 pas d'exécution avec le moteur d'inférence OCaml sur CPU, et le moteur d'inférence JAX sur GPU. La précision est l'erreur quadratique moyenne (*Mean Square Error* MSE) sur l'ensemble des paramètres du modèle. L'ensemble des expériences a été réalisé sur un serveur Intel Xeon E7 avec 64 cœurs, 128Go de RAM et un GPU Nvidia Quadro M6000 avec 24 Go de RAM.

Les exemples retenus sont les suivants : **Coin** correspond au modèle de la pièce présenté en Section 2 où un agent estime le biais d'une pièce à partir d'une série de lancés. On suppose initialement que le biais suit une loi uniforme sur $[0, 1]$. **Gaussian-Gaussian** estime la moyenne et l'écart type d'une loi normale à partir d'un ensemble d'échantillons. On suppose initialement que la moyenne suit une distribution $\mathcal{N}(0, 10)$, et que l'écart type suit une distribution $\mathcal{N}(0, 1)$. **Kalman** correspond au modèle de la Section 3.4 où un agent qui estime sa position à partir d'observations bruitées. À chaque instant, on suppose que la position courante suit une distribution normale autour de la position précédente, et que l'observation courante suit une distribution normale autour de la position courante.

5.2 QR1 : Précision

Les résultats de précision sont présentés en Figure 7. On constate sur l'ensemble des exemples considérés que les deux moteurs d'inférence sont équivalents. Comme attendu, pour tous les exemples, l'erreur diminue quand le nombre de particules augmente avant d'atteindre un plateau lorsque l'estimation se rapproche de la distribution théorique. Les différences mineures sont dues aux divergences d'implémentation des générateurs aléatoires OCaml et JAX.

5.3 QR2 : Performance

Les résultats de performance sont présentés en regard des résultats de précisions en Figure 8. Les performances du moteur d'inférence OCaml font apparaître le compromis classique temps de calcul/précision : plus le nombre de particules est élevé, plus le temps de calcul est important, mais plus l'estimation est précise.

Les performances du moteur d'inférence JAX restent quasiment constantes jusqu'à 100 000 particules. La compilation vers du code GPU optimisé est coûteuse et n'est amortie que pour un nombre élevé de particules (autour de 10 000 pour nos exemples). En revanche, une fois le code compilé, JAX peut facilement exécuter un très grand nombre de particules en parallèle à peu de frais. Pour tous les modèles, les limites du GPU sont atteintes vers 200 000 particules. On retrouve alors des performances qui se dégradent linéairement avec le nombre de particules, mais qui restent significativement supérieurs à celle de l'implémentation en OCaml.

6 Travaux connexes et conclusion

La parallélisation des algorithmes d'inférence pour la programmation probabiliste est un sujet de recherche dynamique. La plupart des langages probabiliste modernes [6, 23, 14, 25] s'appuient sur des systèmes de calcul parallèle efficaces popularisés par l'apprentissage profond. Notre travail est cependant le premier à appliquer ces techniques à un langage réactif comme ProbZelus. Il existe également plusieurs travaux sur la parallélisation de langages synchrones [17, 22, 12]. Mais aucun ne traite les problèmes spécifiques de la programmation probabiliste.

Dans cet article nous avons montré comment utiliser JAX pour paralléliser l'inférence probabiliste sur les modèles réactifs programmés avec ProbZelus. Notre implémentation permettra d'experimenter avec d'autres fonctionnalité de JAX comme l'auto-différentiation qui est un composant clé de plusieurs méthodes d'inférence plus avancées.

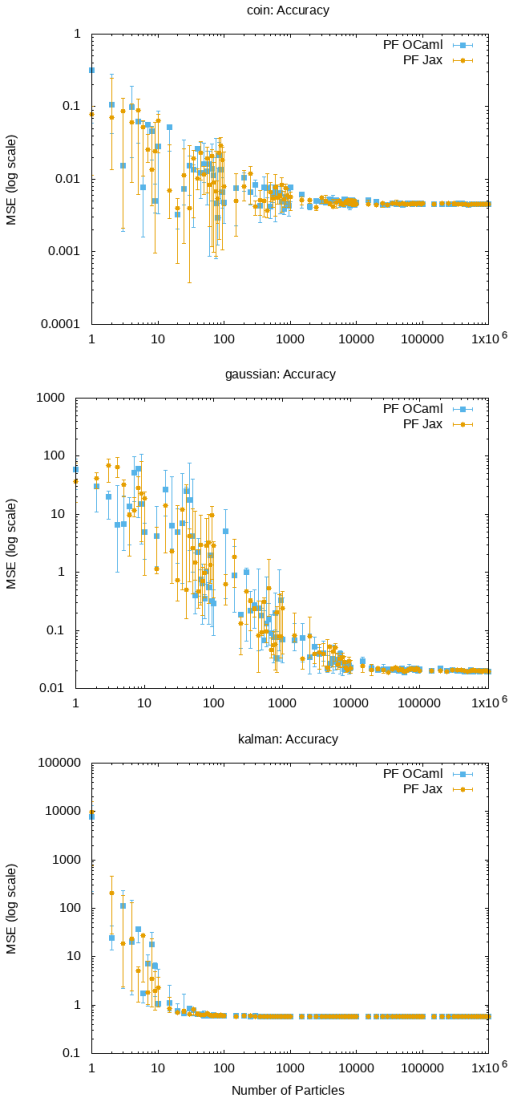


FIGURE 7 – Précision en fonction du nombre de particules.

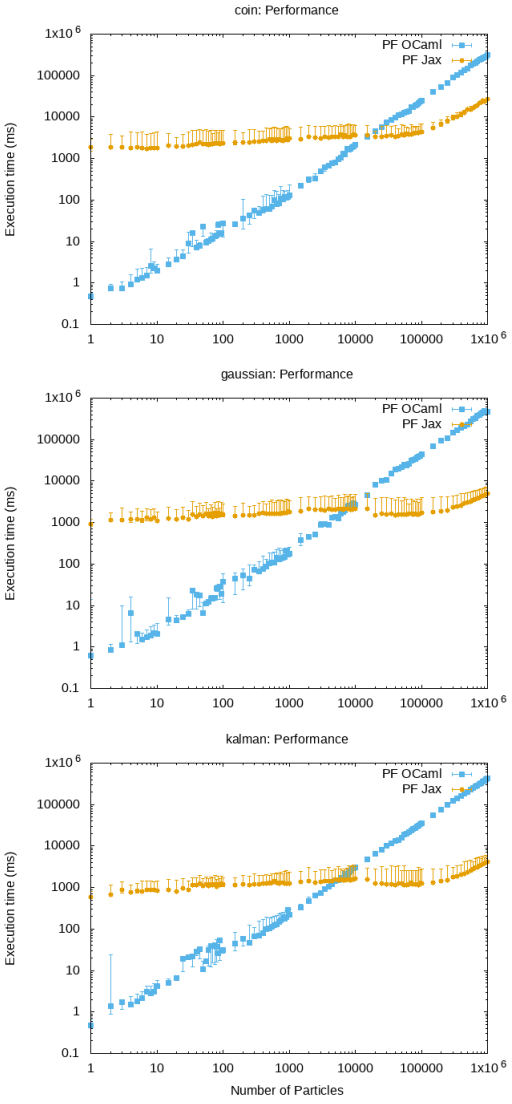


FIGURE 8 – Performance en fonction du nombre de particules.

Références

[1] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Programmation d'applications réactives probabilistes. In *JFLA*, Gruissan, France, 2020.

[2] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive probabilistic programming. In *PLDI*. ACM, 2020.

[3] Albert Benveniste, Timothy Bourke, Benoît Caillaud, Bruno Pagano, and Marc Pouzet. A type-based analysis of causality loops in hybrid systems modelers. In *HSCC*, pages 71–82. ACM, 2014.

[4] 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.

[5] Dariusz Biernacki, Jean-Louis Colaço, Grégoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *LCTES*, pages 121–130. ACM, 2008.

[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, Lélio Brun, and Marc Pouzet. Mechanized semantics and verified compilation for a dataflow synchronous language with reset. *Proc. ACM Program. Lang.*, 4(POPL):44:1–44:29, 2020.

[8] Timothy Bourke and Marc Pouzet. *Zélus, a Hybrid Synchronous Language*. École normale supérieure, September 2013. Distribution at: `zelus.di.ens.fr`.

[9] Timothy Bourke and Marc Pouzet. Zélus: a synchronous language with ODEs. In *HSCC*, pages 113–118. ACM, 2013.

[10] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018.

[11] 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.

[12] Albert Cohen, Léonard Gérard, and Marc Pouzet. Programming parallelism with futures in lustre. In *EMSOFT*, pages 197–206. ACM, 2012.

[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] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A modular memory optimization for synchronous data-flow languages: application to arrays in a lustre compiler. In *LCTES*, pages 51–60. ACM, 2012.

[17] Alain Girault. A survey of automatic distribution method for synchronous programs. In F. Maraninchi, M. Pouzet, and V. Roy, editors, *SLAP*, ENTCS, 2005.

[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] Lawrence M. Murray and Thomas B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46:29–43, 2018.

[21] Bruno Pagano. Le Langage Scade 6 pour les systèmes embarqués De la conception à la compilation. Séminaire du cours de G. Berry au Collège de France, 2013.

[22] Claire Pagetti, Julien Forget, Frédéric Boniol, Mikel Cordovilla, and David Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discret. Event Dyn. Syst.*, 21(3):307–338, 2011.

[23] Du Phan, Neeraj Pradhan, and Martin Jankowiak. Composable effects for flexible and accelerated probabilistic programming in numpyro. *arXiv preprint arXiv:1912.11554*, 2019.

[24] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank D. Wood. Design and implementation of probabilistic programming language Anglican. In *IFL*, pages 6:1–6:12. ACM, 2016.

[25] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *ICLR (Poster)*. OpenReview.net, 2017.

Formalisation d'un vérificateur efficace d'assertions arithmétiques à l'exécution

Thibaut Benjamin[1] Félix Ridoux[1,2] Julien Signoles[1] *

[1] Université Paris-Saclay, CEA, List, Palaiseau, France
prenom.nom@cea.fr
[2] École Normale Supérieure de Rennes, Rennes, France
felix.ridoux@ens-rennes.fr

Résumé

La vérification d'assertions à l'exécution est une technique consistant à vérifier la validité d'annotations formelles pendant l'exécution d'un programme. Bien qu'ancienne, cette technique reste encore peu étudiée d'un point de vue théorique. Cet article contribue à pallier ce manque en formalisant un vérificateur d'assertions à l'exécution pour des propriétés arithmétiques entières. La principale difficulté réside dans la modélisation d'un générateur de code pour les propriétés visées qui génère du code à la fois correct et efficace. Ainsi, le code généré repose sur des entiers machines lorsque le générateur peut prouver qu'il est correct de le faire et sur une bibliothèque spécialisée dans l'arithmétique exacte, correcte mais moins efficace, dans les autres cas. Il utilise pour cela un système de types dédié. En outre, la logique considérée pour les propriétés inclue des constructions d'ordre supérieur. L'article présente également une implémentation de ce générateur de code au sein d'E-ACSL, le greffon de Frama-C dédié à la vérification d'assertions à l'exécution, ainsi qu'une première évaluation expérimentale démontrant empiriquement l'efficacité du code généré.

1 Introduction

Contexte. La vérification d'assertions à l'exécution (abbrégée en RAC [1] par la suite) est une technique de vérification formelle consistant à vérifier la validité d'annotations formelles (typiquement, des assertions) pendant l'exécution d'un programme [7]. RAC peut être vue comme une *technique de compilation* générant du code exécutable ou du code-octet à partir d'annotations formelles, soit directement pendant la compilation, soit indirectement en générant du code source qui est ensuite traduit en code exécutable à l'aide d'un compilateur classique.

Comparée aux méthodes formelles permettant de vérifier statiquement des propriétés formelles, comme la vérification de modèles [5], la vérification déductive [11] ou l'interprétation abstraite [21], RAC apporte des garanties moins fortes, car elle ne raisonne pas sur l'ensemble des traces d'exécutions du programme mais uniquement sur celle en cours d'exécution (ou, éventuellement, celles déjà exécutées). En contrepartie, elle est plus légère à mettre en œuvre pour l'utilisateur, en ne requérant en particulier aucune expertise particulière.

Bien qu'aussi ancienne que ses alternatives statiques, RAC a été nettement moins étudiée que celles-ci d'un point de vue théorique [23]. Par exemple, les auteurs de Spec#, un langage de spécification formelle pour C# permettant à la fois l'usage de RAC et de vérification déductive, indiquent que leur « vérificateur à l'exécution est simple » [2], sans fournir plus de détails sur son fonctionnement, tout en précisant néanmoins qu'il n'est pas suffisament efficace en pratique [3] [2].

*Le premier et le troisième auteurs ont été financés par le programme de Recherche et Innovation Horizon 2020 de l'Union Européenne sous l'accord numéro N° 883242, projet ENSURESEC.

1. pour *Runtime Assertion Checking*.
2. "The run-time checker is straightforward".
3. "the run-time overhead is prohibitive".

Voilà en effet tout l'enjeu de RAC : générer du code permettant de vérifier des propriétés à l'exécution, à la fois *correctement*, *i.e.*, en ne se trompant pas sur les verdicts de validité des propriétés vérifiées, et *efficacement*, *i.e.*, en minimisant les surcoûts à l'exécution, aussi bien en temps qu'en mémoire, induits par les vérifications effectuées.

Contributions. Cet article étudie ce problème en se focalisant sur les propriétés arithmétiques entières. Celles-ci sont en effet intéressantes à plus d'un titre : d'un point de vue utilisateur, elles sont fondamentales car inévitables en pratique, tandis que, d'un point de vue théorique, elles soulèvent des problèmes intéressants. En effet, les langages de spécification formelle modernes permettent exprimer ces propriétés dans $\mathbb{Z}$, l'ensemble des entiers relatifs, alors que les langages exécutables cibles reposent sur des représentations machines bornées (typiquement, des intervalles finis d'entiers au lieu de $\mathbb{Z}$). Pour RAC, il en résulte une tension entre correction et efficacité : la première requiert l'utilisation d'une bibliothèque spécialisée permettant de modéliser fidèlement $\mathbb{Z}$ et son arithmétique (par exemple, GMP [4] en C), au prix d'un surcoût non négligeable à l'exécution, tandis que la seconde requiert l'utilisation directe des représentations machines bornées, au risque d'être incorrecte.

Pour surmonter cette difficulté, nous utilisons ici un système de types dédié permettant de reposer, d'une manière correcte, sur une représentation machine bornée, lorsqu'il infère un type suffisamment précis. Ce système de types est antérieur à nos travaux [12, 13] mais a été étendu ici aux conditions ternaires et à trois nouveaux termes arithmétiques d'ordre supérieur représentant une somme $\Sigma_{i=1}^{n} f(i)$, un produit $\Pi_{i=1}^{n} f(i)$, et un dénombrement d'éléments d'un ensemble d'entiers satisfaisant une certaine propriété P. L'apport principal de nos travaux réside néanmoins en la formalisation de la phase de génération de code en présence d'un tel système, ce qui n'avait encore jamais été réalisé. Nous présentons également l'implantation qui en a été faite dans E-ACSL [24], le vérificateur d'annotations à l'exécution de Frama-C [3], une plateforme ouverte pour l'analyse de programmes C, ainsi qu'une première évaluation de l'efficacité du code généré pour les constructions d'ordre supérieur. En résumé, nos contributions sont les suivantes :

— une extension à l'ordre supérieur du système de types d'E-ACSL [12, 13] ;
— une formalisation du générateur de code reposant sur ce système de types ;
— une présentation de l'implémentation de ce générateur de code dans E-ACSL ;
— une évaluation de l'efficacité du code généré pour les constructions d'ordre supérieur.

Travaux connexes. Y. Cheon [6] s'est, le premier, intéressé à la formalisation d'un vérificateur à l'exécution lié à JML [14], un langage de spécification formelle pour Java. Il ne démontre néanmoins aucun résultat. En outre, il ne s'est pas intéressé à l'arithmétique, étant donné que, à l'époque de ces travaux (le tout début des années 2000), l'arithmétique de JML était exactement celle de Java : la traduction était donc exactement la fonction identité. On peut néanmoins mentionner le fait que notre notion de macros, présentée section 5.1 et sur laquelle se fonde notre traduction de l'arithmétique entière, est proche de sa notion de contexte introduite pour générer le code des constructions indéfinies de JML, comme $1/0$.

Par la suite, H. Lehner [15] a formalisé en Coq une large partie de la sémantique de JML, tout en définissant et prouvant correct un algorithme pour le RAC d'une construction complexe (la clause `assignable` de JML, spécifiant les zones mémoires potentiellement écrites par une fonction), mais cette construction est indépendante des constructions arithmétiques.

Plus récemment, plusieurs travaux ont eu lieu autour d'E-ACSL. Nous avons déjà mentionné ceux liés à son système de types permettant de générer du code efficace pour l'arithmétique entière [12], étendue aux rationnels [13]. Ils n'abordent cependant pas le problème de la génération de code. G. Petiot a le premier, formalisé une transformation de programme proche d'E-ACSL

4. https://gmplib.org/

incluant l'arithmétique entière exacte [18], mais sans s'intéresser au problème d'optimisation. D. Ly s'est également intéressé à une telle transformation, mais centrée sur les propriétés mémoires [16]. En particulier, tous les entiers sont bornés.

Enfin, très récemmment, C. Pascutto et J. C. Filliâtre ont proposé Ortac, un vérificateur à l'exécution de propriétés formelles pour des programmes OCaml [9]. Il repose sur un mécanisme similaire à celui d'E-ACSL pour générer du code efficace pour l'arithmétique. Néanmoins, le mécanisme de génération de code n'est pas détaillé. *A fortiori*, il n'est pas formalisé.

Par ailleurs, W. Dietz, P. Li, J. Regher et V. Adve [8] proposent de tester à l'exécution si une opération entière provoque un débordement. Leur méthode pourrait être combinée à notre système de types lorsque ce dernier ne permet pas de conclure si le calcul peut ou non être fait avec des entiers machines bornés. Il est néanmoins à noter que leur solution repose explicitement sur LLVM, alors que notre système est indépendant du compilateur et du système sous-jacent.

Plan. D'abord, la section 2 présente un exemple complet, tandis que la section 3 introduit la syntaxe et la sémantique de nos langages d'étude. Ensuite, la section 4 présente notre système de types et la section 5 formalise le générateur de code. Enfin la section 6 présente sa mise en œuvre dans E-ACSL et une évaluation empirique de l'efficacité du code généré.

2 Exemple complet

Cette section introduit un exemple afin d'illustrer concrètement le problème. Considérons une machine 64-bit exécutant un programme C contenant trois variables entières a, b et n, et supposons que l'on veuille vérifier à l'exécution que la somme des carrés des nombres entre a et b est inférieure à n, autrement dit que $\Sigma_{k=a}^{b} k^2 < n$.

Il est possible pour cela d'utiliser le langage de spécification ACSL [4] afin d'écrire à l'endroit adéquat du programme l'assertion `/*@ assert \sum(a, b, \lambda integer k; k * k)< n; */` et d'utiliser le greffon E-ACSL [24] de Frama-C [3] afin de transformer l'annotation en code C à compiler avec le reste du programme. En supposant que n soit un entier de type `int` codé sur quatre octets, la Figure 1 présente des versions, très légèrement simplifiées à des fins didactiques, des deux traductions possibles effectuées par E-ACSL, en fonction du type des variables a et b (`char` codé sur un octet à gauche et `int` codé sur quatre octets à droite).

On peut remarquer que les deux versions du code généré suivent la même structure, à savoir d'abord des déclarations et initialisations des mêmes variables temporaires (sauf pour les types et pour les variables $\_\_n$ et $\_\_eq$ présentes à droite et pas à gauche), ensuite une boucle permettant d'itérer sur tous les nombres entre a et b et calculant la somme $\_\_sum$ de leurs carrés, enfin une comparaison de cette somme avec n. Néanmoins, les types des variables générées et les opérations associées diffèrent fortement.

En effet, dans la version de gauche pour laquelle a et b sont de type `char`, et donc avec des valeurs comprises entre -128 et 127, E-ACSL utilise un système de types dédié pour inférer automatiquement que la somme des carrés entre a et b est nécessairement comprise entre $256 \times (-128) \times 127 = -4\,161\,536$ et $256 \times (-128)^2 = 4\,194\,560$ car le nombre maximal de valeurs entre a et b inclus est 256 et que le plus petit (*resp.* grand) carré calculé est -128×127 (*resp.* $(-128)^2$) : il s'agit d'une sur-approximation des résultats possibles, qui peut sembler importante mais qui est néanmoins suffisante pour conclure que tous les calculs peuvent être effectués dans le type `int`, dont les valeurs sont comprises entre -2^{31} et $2^{31} - 1$, sans aucun risque de dépassement arithmétique. En conséquence, le code généré repose sur ce type de données.

Néanmoins, lorsque les variables a et b sont de type `int` comme dans la version de droite, et en suivant le même raisonnement, E-ACSL infère automatiquement que la somme des carrés est, cette fois, nécessairement comprise entre $2^{32} \times (-2^{31}) \times (2^{31} - 1) = -2^{94} + 2^{62}$ et $2^{32} \times (-2^{31})^2 =$

```c
// code si a et b sont de type char:
int __k;
int __one;
int __cond;
int __lambda;
int __sum;
__one = 1;
__cond = 0;
__lambda = 0;
__sum = 0;
__k = (int)a;
while (1) {
  __cond = __k > (int)b;
  if (__cond) break;
  else {
    __lambda = __k * __k;
    __sum += __lambda;
    __k += __one;
  }
}
__e_acsl_assert(__sum < n);
```

```c
// code si a et b sont de type int:
long __k;
long __one;
int __cond;
__mpz_t __lambda;
__mpz_t __sum;
__mpz_t __n;
int __eq;
__one = 1;
__cond = 0;
__gmpz_init_set_si(__lambda,0L);
__gmpz_init_set_si(__sum,0L);
__k = (long)a;
while (1) {
  __cond = __k > (long)b;
  if (__cond) break;
  else {
    { __mpz_t __1;
      __gmpz_init_set_si(__1,__k *
          __k);
      __gmpz_set(__lambda, __1);
      __gmpz_clear(__1); }
    __gmpz_add(__sum,__sum,__lambda);
    __k += __one;
  }
}
__gmpz_init_set_si(__n,(long)n);
__eq = __gmpz_cmp(__sum,__n);
__e_acsl_assert(__eq < 0);
__gmpz_clear(__lambda);
__gmpz_clear(__sum);
__gmpz_clear(__n);
```

FIGURE 1 – Code généré par E-ACSL pour vérifier que la somme des carrés entre a et b est inférieure à n en supposant, à gauche (resp. droite), que a et b sont de type `char` (resp. `int`).

2^{94}, ce qui inclut des entiers bien au-delà de ceux représentables par une machine 64-bit. Par conséquent, afin de ne pas risquer un dépassement arithmétique, E-ACSL choisit de reposer sur des entiers GMP en précision arbitraire, de type `__mpz_t`, pour générer le code. Ainsi, toutes les opérations sur ces entiers sont effectués *via* des appels de fonctions définies dans la bibliothèque GMP, sans compter que chaque entier GMP doit être désalloué après utilisation *via* un appel à la fonction `__gmpz_clear`. On peut aussi noter que le type utilisé pour l'indice de boucle `__k` est `long` car E-ACSL infère qu'à l'issue de la dernière itération de la boucle, cette variable peut valoir, dans le pire des cas, $2^{31} - 1 + 1 = 2^{31}$, ce qui n'est pas représentable dans le type `int` mais l'est dans le type `long` dont la plus grande valeur est $2^{63} - 1$.

Ainsi, E-ACSL génère du code reposant sur des entiers machines efficaces lorsque son système de types lui indique qu'il est toujours correct de le faire, alors qu'il utilise des entiers GMP, un ordre de magnitude plus lent, lorsque son système de types estime qu'il existe une possibilité de dépasser le plus petit ou le plus grand des entiers représentables.

3 Langages d'étude

Cette section introduit la syntaxe (section 3.1) et la sémantique (section 3.2) des langages qui serviront de support à notre formalisation. Il s'agit de versions simplifiées des langages C et ACSL, intégrant également la bibliothèque GMP.

⟨*statement*⟩ ::= ⟨*variable*⟩ '`=`' ⟨*expression*⟩ affectation
| ⟨*statement*⟩ '`;`' ⟨*statement*⟩ séquence
| '`if(`' ⟨*expression*⟩ '`)`' ⟨*statement*⟩ '`else`' ⟨*statement*⟩ condition
| '`while(`' ⟨*expression*⟩ '`)`' ⟨*statement*⟩ boucle
| '`/*@ assert`' ⟨*predicate*⟩ '`; */`' assertion logique
| '`assert(`' ⟨*expression*⟩ '`)`' assertion programmatique
| ⟨*mpz statement*⟩ appel à GMP

⟨*expression*⟩ ::= ⟨*integer*⟩
| ⟨*variable*⟩ variable entière
| ⟨*expression*⟩ ◇ ⟨*expression*⟩ ◇ ∈ {'`+`', '`-`', '`*`', '`/`'}
| ⟨*expression*⟩ ◁ ⟨*expression*⟩ ◁ ∈ {'`<`', '`<=`', '`>`', '`>=`', '`==`', '`!=`'}

(a) Syntaxe du langage mini-C.

⟨*predicate*⟩ ::= '`\true`'
| '`\false`'
| ⟨*term*⟩ ◁ ⟨*term*⟩ ◁ ∈ {'`<`', '`<=`', '`>`', '`>=`', '`==`', '`!=`'}
| '`!`' ⟨*predicate*⟩
| ⟨*predicate*⟩ '`||`' ⟨*predicate*⟩

⟨*term*⟩ ::= ⟨*integer*⟩
| ⟨*variable*⟩ variable mini-C
| ⟨*binder*⟩ variable logique
| ⟨*term*⟩ ◇ ⟨*term*⟩ ◇ ∈ {'`+`', '`-`', '`*`', '`/`'}
| ⟨*predicate*⟩ '`?`' ⟨*term*⟩ '`:`' ⟨*term*⟩ terme conditionel
| '`\sum(`' ⟨*term*⟩ '`,`' ⟨*term*⟩ '`, \lambda`' ⟨*binder*⟩ '`;`' ⟨*term*⟩ '`)`'
| '`\product(`' ⟨*term*⟩ '`,`' ⟨*term*⟩ '`, \lambda`' ⟨*binder*⟩ '`;`' ⟨*term*⟩ '`)`'
| '`\numof(`' ⟨*term*⟩ '`,`' ⟨*term*⟩ '`, \lambda`' ⟨*binder*⟩ '`;`' ⟨*predicate*⟩ '`)`'

(b) Syntaxe du langage mini-ACSL.

⟨*mpz statement*⟩ ::= '`mpz_init(`' ⟨*variable*⟩ '`)`' allocation d'un `mpz`
| '`mpz_set_int(`' ⟨*variable*⟩ '`,`' ⟨*expression*⟩ '`)`' affectation d'un entier mini-C dans un `mpz`
| '`mpz_set_mpz(`' ⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`)`' affectation d'un `mpz` dans un autre
| '`mpz_clear(`' ⟨*variable*⟩ '`)`' dé-allocation d'un `mpz`
| '`mpz_add(`' ⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`)`' addition de deux `mpz`
| '`mpz_sub(`' ⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`)`' soustraction de deux `mpz`
| '`mpz_mul(`' ⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`)`' multiplication de deux `mpz`
| '`mpz_div(`' ⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`)`' division de deux `mpz`
| ⟨*variable*⟩ = '`mpz_cmp(`'⟨*variable*⟩ '`,`' ⟨*variable*⟩ '`)`' comparaison de deux `mpz`

(c) Syntaxe du langage mini-GMP.

FIGURE 2 – Syntaxe des langages d'étude.

3.1 Syntaxe

La syntaxe de nos langages d'étude est présentée dans la Figure 2. Ces langages, au nombre de trois, sont appelés mini-C, mini-ACSL et mini-GMP et sont inter-dépendants. Le premier, mini-C, est le langage de programmation et correspond à un sous-ensemble du langage C. Les instructions (*statements*) sont constituées d'affectations et des structures de contrôle les plus classiques, étendues aux assertions logiques et programmatiques ainsi qu'aux appels à GMP. Les assertions logiques correspondent aux assertions ACSL et sont celles qui doivent être traduites en code C pour être exécutées, tandis que les assertions programmatiques sont justement les assertions exécutables vers lesquelles sont traduites les assertions logiques. Elles correspondent aux appels à `__e_acsl_assert` dans la Figure 1 (lignes 21 à gauche et 28 à droite).

Les expressions, quant à elles, sont limitées aux opérateurs arithmétiques et aux comparateurs logiques classiques. Le seul type supporté est `int`. L'extension aux autres types d'entiers bornés du C, comme `unsigned long`, ne pose aucun problème théorique, mais complique inutilement le propos. L'extension aux nombres flottants, considéré dans [13], n'est pas étudiée ici, quoique supportée en pratique (voir section 6).

Le langage mini-ACSL introduit les termes et les prédicats utilisés par les assertions logiques. Il définit une logique propositionnelle dans laquelle les termes peuvent contenir des conditions ternaires, des entiers en précision arbitraire, des variables venant du monde programmatique mini-C ou liées à des lambda-expressions. Par la suite, les premières seront nommées *variables* et les dernières *lieurs*. Les lieurs sont présents dans les trois constructions spécifiques `\sum(a,b,f)`, `\product(a,b,f)` et `\numof(a,b,p)` dénotant respectivement la somme $\sum_{k=a}^{b} f(k)$, le produit

$\prod_{k=a}^{b} f(k)$ et le nombre d'élements dans l'intervalle $[a, b]$ satisfaisant le prédicat p, autrement dit un dénombrement. Par convention, et pour distinguer aisément les variables des lieurs, les premières seront notées x et les secondes seront notées ξ, éventuellement suffixées d'un indice. On peut également noter que les connecteurs logiques sont limités à la négation et à la disjonction, ce qui est suffisant pour encoder les autres. L'extension à une logique du premier ordre, considérée dans les travaux passés [12, 13], n'est pas étudiée ici pour simplifier le propos.

Le langage mini-GMP introduit les appels à la bibliothèque GMP. Les variables en argument des fonctions sont des variables GMP modélisant des entiers en précision arbitraire. La variable stockant l'entier résultant de la fonction `mpz_cmp` est une variable mini-C. Les variables GMP sont des pointeurs qui doivent être alloués et désalloués *via*, respectivement, à des appels à `mpz_init` et `clear`. Ces variables peuvent être inialisées *via* à un appel à `mpz_set_int` ou `mpz_set_mpz` en fonction du type de l'expression en argument (type `int` ou entier GMP). Les opérations autorisées sur les entiers GMP sont les quatre opérations arithmétiques usuelles et la comparaison avec `mpz_cmp`.

3.2 Sémantique

Les sémantiques dynamiques des trois langages présentés à la section précédente sont définies à l'aide d'une sémantique opérationnelle à grand pas, décrite dans la suite de cette section.

Notations. Etant donné deux ensembles X et Y, on note $f : X \rightharpoonup Y$ pour indiquer que f est une fonction partielle de X vers Y, que l'on voit comme une fonction $f : X \to Y \uplus \bot$, où l'élément $\bot$ signifie que la fonction n'est pas définie. On note $\mathrm{cod}(f)$, le co-domaine de f. Étant donné une fonction $f : X \to Y$ et deux éléments $x \in X, y \in Y$, on note $f\{x \leftarrow y\}$ la fonction qui coïncide avec f en tout point, sauf en x où l'on a $f\{x \leftarrow y\}(x) = y$. Par ailleurs, on note $\mathbb{B} = \{V, F\}$, l'ensemble des valeurs de vérité.

Une police de caractères monospace est utilisée pour représenter les opérateurs et comparateurs syntaxiques (par exemple `+` et `<` pour l'addition et l'inégalité stricte mini-C), et une police proportionnelle pour représenter les opérateurs et comparateurs sémantiques correspondant (par exemple $+$ et $<$ pour les opérateurs correspondants aux précédents). Un opérateur (*resp.* comparateur) générique est noté $\diamond$ (*resp.* $\triangleleft$) et sa version sémantique correspondante est notée $\dot{\diamond}$ (*resp.* $\dot{\triangleleft}$). Par extension, la constante entière correspondante à une constante syntaxique entière z de nos langages sera notée $\dot{z}$.

Variables, lieurs, valeurs et environnements. L'ensemble infini dénombrable des variables programmatiques de nos langages est noté $\mathcal{V}$, tandis que celui des lieurs logiques est noté $\mathcal{L}$. En outre, on note `Int` l'ensemble des valeurs possibles d'une variable de type `int` et `Mpz` l'ensemble des valeurs possibles d'une variable de type mpz. L'ensemble des valeurs est ainsi $\mathbb{V} \triangleq \texttt{Int} \uplus \texttt{Mpz}$, c'est-à-dire l'union disjointe de ces deux ensembles de valeurs. De plus, étant donné une constante entière syntaxique z de mini-C ou mini-ACSL, on note z^{int} et z^{mpz} les valeurs entières correspondant à z encodée respectivement dans le type `int` et dans le type `mpz`, de manière à ce qu'un élément de $\mathbb{V}$ soit nécessairement de la forme z^{int} ou z^{mpz}. On appelle $\mathrm{unwrap} : \mathbb{V} \to \mathbb{Z}$ la fonction qui renvoie l'entier codé par une valeur. Cette fonction vérifie donc les propriétés $\mathrm{unwrap}(n^{\mathsf{mpz}}) = n$ et $\mathrm{unwrap}(n^{\mathsf{int}}) = n$.

Enfin, un environnement programmatique, noté Δ, est une fonction partielle des variables vers les valeurs, c'est-à-dire $\Delta : \mathcal{V} \rightharpoonup \mathbb{V}$. Symmétriquement, un environnement logique, noté Λ, est une fonction partielle des lieurs vers les entiers, c'est-à-dire $\Lambda : \mathcal{L} \rightharpoonup \mathbb{Z}$. Par la suite, on appelle *environnement étendu* un couple Δ, Λ constitué d'un environnement programmatique et d'un environnement logique.

$$\frac{}{\Delta,\,\Lambda \models_p \texttt{\textbackslash true} \Rightarrow V} \qquad \frac{}{\Delta,\,\Lambda \models_p \texttt{\textbackslash false} \Rightarrow F} \qquad \frac{\Delta,\,\Lambda \models_p p \Rightarrow b}{\Delta,\,\Lambda \models_p \texttt{!}p \Rightarrow \neg b} \qquad \frac{\Delta,\,\Lambda \models_p p_1 \Rightarrow V}{\Delta,\,\Lambda \models_p p_1 \texttt{ || } p_2 \Rightarrow V}$$

$$\frac{\Delta,\,\Lambda \models_p p_1 \Rightarrow F \quad \Delta,\,\Lambda \models_p p_2 \Rightarrow b}{\Delta,\,\Lambda \models_p p_1 \texttt{ || } p_2 \Rightarrow b} \qquad \frac{\Delta,\,\Lambda \models_t t_1 \Rightarrow v_1 \quad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2}{\Delta,\,\Lambda \models_p t_1 \lhd t_2 \Rightarrow v_1 \dot{\lhd} v_2}$$

(a) Sémantique des prédicats.

$$\frac{}{\Delta,\,\Lambda \models_t z \Rightarrow \dot{z}} \qquad \frac{\Delta(x) = z^{\mathsf{int}}}{\Delta,\,\Lambda \models_t x \Rightarrow z} \qquad \frac{\Lambda(\xi) = z}{\Delta,\,\Lambda \models_t \xi \Rightarrow z} \qquad \frac{\Delta,\,\Lambda \models_t t_1 \Rightarrow v_1 \quad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2}{\Delta,\,\Lambda \models_t t1 \diamond t2 \Rightarrow v_1 \dot{\diamond} v_2}$$

$$\frac{\Delta,\,\Lambda \models_p p \Rightarrow V \quad \Delta,\,\Lambda \models_t t_1 \Rightarrow v_1}{\Delta,\,\Lambda \models_p p \,?\, t_1 : t_2 \Rightarrow v_1} \qquad \frac{\Delta,\,\Lambda \models_p p \Rightarrow F \quad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2}{\Delta,\,\Lambda \models_p p \,?\, t_1 : t_2 \Rightarrow v_2}$$

$$\frac{\Delta,\,\Lambda \models_t t_1 \Rightarrow v_1 \qquad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2 \qquad \forall k \in [v_1, v_2],\, \Delta,\,\Lambda\{\xi \leftarrow k\} \models_t t_3 \Rightarrow v_3^k}{\Delta,\,\Lambda \models_t \texttt{\textbackslash sum}(t_1,\, t_2,\, \texttt{\textbackslash lambda}\,\xi;\, t_3) \Rightarrow \sum_{k=v_1}^{v_2} v_3^k}$$

$$\frac{\Delta,\,\Lambda \models_t t_1 \Rightarrow v_1 \qquad \Delta,\,\Lambda \models_t t_2 \Rightarrow v_2 \qquad \forall k \in [v_1, v_2],\, \Delta,\,\Lambda\{\xi \leftarrow k\} \models_t t_3 \Rightarrow v_3^k}{\Delta,\,\Lambda \models_t \texttt{\textbackslash product}(t_1,\, t_2,\, \texttt{\textbackslash lambda}\,\xi;\, t_3) \Rightarrow \prod_{k=v_1}^{v_2} v_3^k}$$

$$\frac{\Delta,\,\Lambda \models_t \texttt{\textbackslash sum}(t_1,\, t_2,\, \texttt{\textbackslash lambda}\,\xi;\, p \,?\, 1 : 0) \Rightarrow v}{\Delta,\,\Lambda \models_t \texttt{\textbackslash numof}(t_1,\, t_2,\, \texttt{\textbackslash lambda}\,\xi;\, p) \Rightarrow v}$$

(b) Sémantique des termes.

FIGURE 3 – Sémantique du langage mini-ACSL.

Sémantique statique. Dans un souci de simplicité, les systèmes de types des langages mini-C, mini-GMP et mini-ACSL, c'est-à-dire leurs sémantiques statiques, ne sont pas décrits. On suppose néanmoins les programmes en entrée bien typés.

Sémantique des prédicats et des termes. La Figure 3a présente la sémantique des prédicats en utilisant un jugement $\Delta, \Lambda \models_p \texttt{pred} \Rightarrow b$ exprimant le fait que, dans l'environnement étendu (Δ, Λ), le prédicat `pred` s'évalue en une valeur de vérité $b \in \mathbb{B}$. De la même manière, la Figure 3b présente la sémantique des termes, en utilisant un jugement $\Delta, \Lambda \models_t \texttt{term} \Rightarrow v$ exprimant le fait que, dans l'environnement étendu Δ, Λ, le terme `term` est évalué en l'entier $v \in \mathbb{Z}$.

La plupart de ces règles sont classiques et ne méritent pas d'explication particulière. Les deux opérateurs d'ordre supérieur `\sum` et `\product` sont évalués vers leurs équivalents mathématiques, tandis que l'opérateur de dénombrement est encodé sémantiquement comme une somme. En outre, on peut noter qu'on ignore ici, à des fins simplificatrices, le problème des valeurs indéfinies, qui résulteraient par exemple de l'évaluation de termes comme `1/0`.

Sémantique des expressions. La Figure 4a présente la sémantique des expressions en utilisant un jugement noté $\Delta \models_e \texttt{expr} \Rightarrow v$, qui exprime le fait que l'expression `expr` dans l'environnement Δ est évaluée en la valeur $v \in \mathbb{V}$. Là encore, ces règles sont usuelles, à ceci près qu'elles modélisent les débordements arithmétiques. Pour ce faire, $\min_{\texttt{int}}$ et $\max_{\texttt{int}}$ désignent respectivement le plus petit et le plus grand entier qui que l'on peut coder avec un `int`, c'est-à-dire respectivement -2^{31} et $2^{31} - 1$ en considérant l'exemple de la section 2.

Sémantique des instructions. La Figure 4b présente les règles de sémantique pour les instructions en utilisant un jugement noté $\Delta_1 \models_s \texttt{instr} \Rightarrow \Delta_2$, exprimant le fait que l'instruction `instr` est évaluée en un environnement Δ_2 à partir d'un environnement Δ_1. Cette sémantique est standard pour un tel fragment du langage C, même si elle contient une sémantique des assertions et du langage mini-GMP permettant de manipuler le type `mpz`. Concernant les assertions, on peut noter que la sémantique est dite bloquante [10], c'est-à-dire qu'une assertion logique (respectivement programmatique) ne peut être exécutée que si son prédicat en argument est évalué à V (*resp.*, son expression en argument est évaluée à une valeur entière non nulle).

$$\frac{\texttt{min\_int} \leq \dot{z} \leq \texttt{max\_int}}{\Delta \models_e z \Rightarrow \dot{z}^{\,\mathsf{int}}} \qquad \frac{\Delta(x) = z^{\,\mathsf{int}}}{\Delta \models_e x \Rightarrow z^{\,\mathsf{int}}}$$

$$\frac{\Delta \models_e e_1 \Rightarrow {z_1}^{\,\mathsf{int}} \quad \Delta \models_e e_2 \Rightarrow {z_2}^{\,\mathsf{int}} \quad \texttt{min\_int} \leq z_1 \dot{\diamond}\, z_2 \leq \texttt{max\_int}}{\Delta \models_e e_1 \diamond e2 \Rightarrow (z_1 \dot{\diamond}\, z_2)^{\,\mathsf{int}}}$$

$$\frac{\Delta \models_e e_1 \Rightarrow {z_1}^{\,\mathsf{int}} \quad \Delta \models_e e_2 \Rightarrow {z_2}^{\,\mathsf{int}} \quad z_1 \dot{\triangleleft}\, z_2}{\Delta \models_e e_1 \triangleleft e_2 \Rightarrow 1^{\,\mathsf{int}}} \qquad \frac{\Delta \models_e e_1 \Rightarrow {z_1}^{\,\mathsf{int}} \quad \Delta \models_e e_2 \Rightarrow {z_2}^{\,\mathsf{int}} \quad \neg(z_1 \dot{\triangleleft}\, z_2)}{\Delta \models_e e_1 \triangleleft e_2 \Rightarrow 0^{\,\mathsf{int}}}$$

(a) Sémantique des expressions.

$$\frac{\Delta \models_e e \Rightarrow z^{\,\mathsf{int}}}{\Delta \models_s x = e \Rightarrow \Delta\{\mathtt{x} \leftarrow z^{\,\mathsf{int}}\}} \qquad \frac{\Delta_1 \models_s s_1 \Rightarrow \Delta_2 \quad \Delta_2 \models_s s_2 \Rightarrow \Delta_3}{\Delta_1 \models_s s_1; s_2 \Rightarrow \Delta_3}$$

$$\frac{\Delta \models_e e \Rightarrow 0^{\,\mathsf{int}} \quad \Delta \models_s s_2 \Rightarrow \Delta_2}{\Delta \models_e \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \Rightarrow \Delta_2} \qquad \frac{\Delta \models_e e \Rightarrow z^{\,\mathsf{int}} \quad z^{\,\mathsf{int}} \neq 0 \quad \Delta \models_s s_1 \Rightarrow \Delta_1}{\Delta \models_e \texttt{if}\,(e)\ s_1\ \texttt{else}\ s_2 \Rightarrow \Delta_1}$$

$$\frac{\Delta \models_e e \Rightarrow 0^{\,\mathsf{int}}}{\Delta \models_s \texttt{while}\,(e)\ s \Rightarrow \Delta} \qquad \frac{\Delta \models_e e \Rightarrow z^{\,\mathsf{int}} \quad z \neq 0 \quad \Delta \models_s s \Rightarrow \Delta_2 \quad \Delta_2 \models_s \texttt{while}\,(e)\ s \Rightarrow \Delta_3}{\Delta \models_s \texttt{while}\,(e)\ s \Rightarrow \Delta_3}$$

$$\frac{\Delta, \emptyset \models_p p \Rightarrow V}{\Delta \models_s \texttt{/*@ assert}\ p;\ \texttt{*/} \Rightarrow \Delta} \qquad \frac{\Delta \models_e e \Rightarrow z^{\,\mathsf{int}} \quad z \neq 0}{\Delta \models_s \texttt{assert}(e); \Rightarrow \Delta}$$

$$\frac{\Delta(x) = \bot}{\Delta \models_s \texttt{mpz\_init}(x); \Rightarrow \Delta\{x \leftarrow 0^{\,\mathsf{mpz}}\}} \qquad \frac{\Delta(x) \neq \bot}{\Delta \models_s \texttt{mpz\_clear}(x); \Rightarrow \Delta\{x \leftarrow \bot\}}$$

$$\frac{\Delta(x) \neq \bot \quad \Delta \models_e e \Rightarrow z^{\,\mathsf{int}}}{\Delta \models_s \texttt{mpz\_set\_int}(x,\ e); \Rightarrow \Delta\{x \leftarrow z^{\,\mathsf{mpz}}\}} \qquad \frac{\Delta(x) \neq \bot \quad \Delta(y) = z^{\,\mathsf{mpz}}}{\Delta \models_s \texttt{mpz\_set\_mpz}(x,\ y); \Rightarrow \Delta\{x \leftarrow z^{\,\mathsf{mpz}}\}}$$

$$\frac{\Delta(x) \neq \bot \quad \Delta(y) = {v_y}^{\,\mathsf{mpz}} \quad \Delta(z) = {v_z}^{\,\mathsf{mpz}}}{\Delta \models_s \texttt{mpz\_}\diamond(x,\ y,\ z); \Rightarrow \Delta\{x \leftarrow (v_y \dot{\diamond}\, v_z)^{\,\mathsf{mpz}}\}} \qquad \frac{\Delta(y) = {v_y}^{\,\mathsf{mpz}} \quad \Delta(z) = {v_z}^{\,\mathsf{mpz}}}{\Delta \models_s x = \texttt{mpz\_cmp}(y,\ z); \Rightarrow \Delta\{x \leftarrow (\mathrm{compare}(v_y,\ v_z))^{\,\mathsf{int}}\}}$$

(b) Sémantique des instructions.

FIGURE 4 – Sémantique des langages mini-C et mini-GMP.

Concernant les instructions mini-GMP, un appel à `mpz_init` permet d'allouer une nouvelle variable `mpz` initialisée à 0, tandis que `mpz_clear` effectue l'opération inverse, à savoir désallouer son argument. Notons que nous modélisons le fait qu'une variable x est allouée dans l'environnement Δ par la propriété $\Delta(x) \neq \bot$. La fonction `mpz_set_int` (*resp.* `mpz_set_mpz`) permet d'affecter un entier de type `int` (*resp.* `mpz`) à un entier `mpz` correctement alloué, tandis que `mpz_add`, `mpz_sub`, `mpz_mul`, `mpz_div` et `mpz_cmp` permettent de, respectivement, additionner, soustraire, multiplier, diviser et comparer. Les deux premières opérations stockent leur résultat respectif dans un troisième entier `mpz` préalablement alloué. La sémantique de la comparaison, quant à elle, repose sur une fonction compare : $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ définie par

$$\mathrm{compare}(x,\ y) = \begin{cases} 1 & \text{si } x > y\,; \\ 0 & \text{si } x = y\,; \\ -1 & \text{si } x < y. \end{cases}$$

Cette sémantique est plus précise que celle indiquée par la documentation de GMP, qui se contente de spécifier le signe et non la valeur exacte. Par la suite, nous n'utilisons cette fonction que pour faire des tests à 0. Cette sur-spécification n'a donc aucune incidence.

Il est important de remarquer que cette sémantique des entiers `mpz` permet de simuler le fait que le type `mpz` soit équivalent à un pointeur [5], sans avoir besoin d'intégrer explicitement les pointeurs (ou les tableaux) dans nos langages d'étude. Ce modèle simplifié des entiers `mpz`

5. En réalité, il s'agit d'un tableau à un unique élément, ce qui est sémantiquement équivalent à un pointeur si on s'affranchit des subtilités du typage du C.

est néanmoins suffisant pour exprimer, outre les propriétés sémantiques sur les valeurs entières, les propriétés relative à la bonne formation des programmes par vis-à-vis de l'allocation, y compris l'absence de fuite mémoire si chaque entier alloué avec `mpz_init` est finalement libéré en appelant `mpz_clear`. Il s'agit alors de vérifier que, à la fin de l'exécution, le programme atteint un environnement Δ tel que, pour toute variable x de type `mpz`, on a $\Delta(x) = \bot$.

4 Système de types pour une génération de code efficace

Cette section présente le système de types permettant de décider si le code généré repose sur des entiers et une arithmétique machine bornée, efficaces mais potentiellement incorrects, ou sur des entiers et une arithmétique en précision arbitraire, inefficaces mais nécessairement corrects. Ce système repose sur une inférence d'intervalles permettant de déterminer un ensemble d'entiers consécutifs (autrement dit, un intervalle) le plus petit possible contenant nécessairement l'ensemble des valeurs d'un terme. Le jugement d'évaluation est noté $\Gamma \vDash t : I$ et spécifie que les valeurs du terme t sont incluses dans l'intervalle I dans l'environnement Γ. On note également $\mathbb{I}_\Gamma(t)$ l'intervalle I résultant de cette évaluation. Si l et u sont les valeurs respectivement minimale et maximale d'un intervalle, ce dernier est noté $[l; u]$. L'union de deux intervalles $[l_1; u_1]$ et $[l_2; u_2]$ est l'intervalle $[l_1; u_1] \sqcup [l_2; u_2] \triangleq [\min(l_1, l_2); \max(u_1, u_2)]$. Par convention, un intervalle $[l; u]$ avec $l > u$ est l'intervalle vide.

La Figure 5 introduit le système de règles d'inférence définissant $\Gamma \vDash t : I$. Les cinq premières règles, déjà présentées dans [12], introduisent les cas des constantes entières, des variables arithmétiques et logiques et des opérateurs arithmétiques. On peut en particulier noter que l'intervalle d'une variable programmatique correspond à celui défini par la plus petite et la plus grande valeur représentable dans son type (nécessairement `int` dans notre contexte) : de ce fait, notre inférence d'intervalle s'abstrait du contexte du programme et reste locale à l'assertion dans laquelle le terme est évalué. La sixième règle correspond au cas de l'opérateur de condition ternaire. L'intervalle résultant est l'union des intervalles issus des deux branches de la condition. En pratique, et même si ce n'est pas présenté ici, il est possible d'optimiser ce cas en fonction de la forme de la condition. Par exemple, si cette dernière est le prédicat $\xi \geq 0$, l'intervalle associé à ξ dans l'environnement peut être réduite à ses valeurs positives lorsqu'on évalue t_1 et à ses valeur strictement négative lorsqu'on évalue t_2. Effectuer de telles réductions sur les conditions est standard, notamment en interprétation abstraite [21].

Les neuf dernières règles gèrent le cas des termes d'ordre supérieur. Les trois premières d'entre elles correspondent au cas de la somme. Dans chaque cas, l'intervalle du terme t_3 est calculé dans l'environnement Γ étendu à la variable ξ à laquelle est associé l'intervalle $[l_1; u_2]$, correspondant aux nombres inclus entre la valeur minimale de la borne inférieure et la valeur maximale de la borne supérieure. On peut noter que cet intervalle est vide si $l_1 > u_2$, c'est-à-dire si la borne inférieure est nécessairement strictement plus grande que la borne supérieure. La borne inférieure (*resp.* supérieure) de l'intervalle résultant est la borne inférieure (*resp.* supérieure) de l'intervalle du corps t_3 du terme t, multipliée par la cardinalité minimale (*resp.* *maximale*) de l'intervalle associé à ξ, modulo les problèmes de signe qui génèrent les trois différents cas en fonction du fait que l'intervalle inféré pour t_3 soit positif, négatif, ou puisse contenir 0. La cardinalité d'un intervalle est introduite par l'opérateur δ défini comme suit :

$$\delta(a, b) = \begin{cases} a - b + 1 & \text{si } a \geq b \\ 0 & \text{sinon.} \end{cases}$$

Considérons de nouveau l'exemple de la section 2, et en particulier le terme $\Sigma_{k=a}^{b} k^2$ avec a et b deux variables programmatiques de type `int`. Leur intervalle est donc le même, à savoir $[\min_{\texttt{int}}; \max_{\texttt{int}}]$, c'est-à-dire -2^{31} et $2^{31}-1$ en considérant des `int` représentés sur quatre octets.

$$\frac{}{\Gamma \vDash z : [z;z]} \quad \frac{}{\Gamma \vDash x : [\min_{\texttt{int}}; \max_{\texttt{int}}]} \quad \frac{\xi \in \Gamma}{\Gamma \vDash \xi : \Gamma(\xi)}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad (\diamond \neq / \text{ ou } 0 \notin [l_2;u_2])}{\Gamma \vDash t_1 \diamond t_2 : [\min(l_1 \diamond l_2, l_1 \diamond u_2, u_1 \diamond l_2, u_1 \diamond u_2); \max(l_1 \diamond l_2, l_1 \diamond u_2, u_1 \diamond l_2, u_1 \diamond u_2)]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad 0 \in [l_2;u_2]}{\Gamma \vDash t_1 / \ t_2 : [\min(l_1, -u_1); \max(-l_1, u_1)]} \quad \frac{\Gamma \vDash t_1 : I_1 \quad \Gamma \vDash t_2 : I_2}{\Gamma \vDash p \,?\, t_1 : t_2 : I_1 \sqcup I_2}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad 0 \leq l_3}{\Gamma \vDash \texttt{\textbackslash sum}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [l_3 \times \delta(l_2,u_1); u_3 \times \delta(u_2,l_1)]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad l_3 < 0 \leq u_3}{\Gamma \vDash \texttt{\textbackslash sum}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [l_3 \times \delta(u_2,l_1); u_3 \times \delta(u_2,l_1)]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad u_3 < 0}{\Gamma \vDash \texttt{\textbackslash sum}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [l_3 \times \delta(u_2,l_1); u_3 \times \delta(l_2,u_1)]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad 0 \leq l_3}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [{l_3}^{\delta(l_2,u_1)}; {u_3}^{\delta(u_2,l_1)}]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad l_3 < 0 \leq u_3 \quad |l_3| \geq u_3}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [{u_3}^{\delta(u_2,l_1)}; {u_3}^{\delta(u_2,l_1)}]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad l_3 < 0 \leq u_3 \quad |l_3| < u_3}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [l_3 \times {u_3}^{\delta(u_2,l_1)-1}; {u_3}^{\delta(u_2,l_1)}]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad u_3 < 0 \quad \delta_u = \delta(u_2,l_1) \neq 1}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [-((-l_3)^{\delta_u}); (-l_3)^{\delta_u - \delta_u \%2}]}$$

$$\frac{\Gamma \vDash t_1 : [l_1;u_1] \quad \Gamma \vDash t_2 : [l_2;u_2] \quad \Gamma\{\xi \leftarrow [l_1;u2]\} \vDash t_3 : [l_3;u_3] \quad u_3 < 0 \quad \delta_u = \delta(u_2,l_1) = 1}{\Gamma \vDash \texttt{\textbackslash product}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ t_3) : [-((-l_3)^{\delta_u}); (-l_3)^{\delta_u}]}$$

$$\frac{\Gamma \vDash \texttt{\textbackslash sum}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ p \,?\, 1 : 0) : I}{\Gamma \vDash \texttt{\textbackslash numof}(t_1,\ t_2,\ \texttt{\textbackslash lambda}\,\xi;\ p) : I}$$

FIGURE 5 – Inférence d'intervalles.

L'intervalle associé à la variable logique k est donc aussi $[-2^{31}; 2^{31}-1]$, ce qui permet d'inférer que celui de k^2 est $[-2^{62}+2^{31}; 2^{62}]$ [6]. On applique donc la deuxième des trois règles de la somme pour conclure que l'intervalle résultant est $[(-2^{62}+2^{31}) \times \delta(2^{31}-1, -2^{31}), 2^{62} \times \delta(2^{31}-1, -2^{31})]$, c'est-à-dire $[-2^{94}+2^{62}; 2^{94}]$ car, ici, le résultat des deux applications de δ est exactement 2^{32}.

Les cinq règles pour le produit sont duales de celles pour la somme, mais reposent sur des puissances en lieu et place de produits pour les bornes minimale et maximale des intervalles résultants. En outre, deux cas sont optimisés *via* deux règles dédiées. Le premier survient lorsque l'intervalle du lambda-terme peut contenir 0 et que sa valeur minimale est plus petite en valeur absolue que sa valeur maximale, par exemple l'intervalle $[-3;5]$. Dans ce cas, si le nombre d'itérations maximal est n, alors une borne inférieure correcte est $-3 \times 5^{n-1}$, ce qui est meilleur que $(-5)^n$, résultant du calcul par défaut. Le second cas optimise un produit lorsque son lambda-terme est nécessairement négatif avec un nombre d'itérations impair plus grand que 1 : on sait alors que le résultat, négatif, n'est pas une borne maximale (positive) et, donc, on peut itérer une fois de moins. La dernière règle introduit l'intervalle de l'opérateur de dénombrement en le réduisant au cas de la somme à laquelle il est équivalent.

Maintenant que l'inférence d'intervalle a été présentée, nous pouvons nous concentrer sur le système de types proprement dit, introduit par deux jugements d'évalution $\Gamma \vdash t : \tau_1 \hookleftarrow \tau_2$ et $\Gamma \vdash_p p : \tau_1 \hookleftarrow \tau_2$ pour les termes et les prédicats, respectivement, et signifiant informellement

6. On peut noter au passage la perte de précision sur la multiplication, liée à la non-prise en compte des relations entre ses arguments, et plus précisément du fait qu'il s'agit du cas particulier d'un carré.

$$\frac{\Gamma \vDash z : I}{\Gamma \vdash z : \theta(I)} \quad \frac{}{\Gamma \vdash x : \texttt{int}} \quad \frac{}{\Gamma \vdash \xi : \theta(\Gamma(x))} \quad \frac{\Gamma \vdash t : \tau' \quad \tau' \preccurlyeq \tau}{\Gamma \vdash t : \tau}$$

$$\frac{\Gamma \vdash t_1 : \tau' \quad \Gamma \vdash t_2 : \tau' \quad \tau = \theta(\mathbb{I}_\Gamma(t_1 \diamond t_2))}{\Gamma \vdash t_1 \diamond t_2 : \tau \hookleftarrow \tau'} \quad \frac{\Gamma \vdash_p p : \texttt{int} \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash p\,?\,t_1 : t_2 : \tau}$$

$$\frac{\Gamma \vdash t_1 : \tau' \quad \Gamma \vdash t_2 : \tau' \quad I = \mathbb{I}_\Gamma(t_1) \sqcup \mathbb{I}_\Gamma(t_2) \quad \tau' = \theta(I) \quad \Gamma\{\xi \leftarrow I\} \vdash t_3 : \tau \quad \texttt{name} \in \{\texttt{sum}, \texttt{product}, \texttt{numof}\}}{\Gamma \vdash \backslash\texttt{name}(t_1,\ t_2,\ \backslash\texttt{lambda}\,\xi;\ t_3) : \tau \hookleftarrow \tau'}$$

$$\frac{}{\Gamma \vdash_p \backslash\texttt{true} : \texttt{int}} \quad \frac{}{\Gamma \vdash_p \backslash\texttt{false} : \texttt{int}} \quad \frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau \quad \tau = \theta(\mathbb{I}_\Gamma(t_1) \sqcup \mathbb{I}_\Gamma(t_2))}{\Gamma \vdash_p t_1 \lhd t_2 : \texttt{int} \hookleftarrow \tau} \quad \frac{\Gamma \vdash_p p_1 : \texttt{int} \quad \Gamma \vdash_p p_2 : \texttt{int}}{\Gamma \vdash_p p_1 \,||\, p_2 : \texttt{int}}$$

FIGURE 6 – Système de types.

≪ dans l'environnement Γ, le résultat de la traduction du terme t (*resp.* du prédicat p). peut être de type τ_1 et l'opération arithmétique associée doit être effectuée avec le type τ_2 ≫. Cette distinction entre types du résultat et de l'opération est nécessaire pour typer précisément les opérateurs monotones décroissants, comme la division, pour lesquels le résultat peut être petit (et donc tenir dans un `int`), même si les arguments (et notamment le dénominateur pour la division) sont grands (ce qui nécessite de calculer la division en précision arbitraire). Elle peut néanmoins être omise lorsque les deux types sont les mêmes, auquel cas on écrit juste τ au lieu de $\tau \hookleftarrow \tau$. En outre, on muni notre algèbre de types, restreinte aux types `int` et `mpz` dans cet article, d'une relation d'ordre notée $\preccurlyeq$: en complément de son caractère réflexif, elle permet de spécifier que le type `int` est plus petit que le type `mpz`.

La Figure 6 introduit les règles des deux jugements de typage. Elles utilisent notamment un type $\theta(I)$ correspondant au plus petit type pouvant représenter l'ensemble des valeurs de l'intervalle I. Il s'agit donc du type `int` lorsque I est inclus dans l'intervalle $[\min_{\texttt{int}}; \max_{\texttt{int}}]$ et du type `mpz` sinon. Ces jugements de typage sont similaires à, quoique plus simples que, ceux déjà introduits dans les travaux passés [12, 13], à l'exception des sixième et septième règles, qui permettent respectivement de déduire le type d'une condition à partir du type de ses sous-termes, et de typer les quantificateurs étendus à partir du type de leur corps, en prenant garde à introduire correctement la variable liée dans l'environnement Γ. On peut noter que la quatrième règle est une règle standard de subsomption introduisant du sous-typage [19]. Grâce à elle, notre système de types peut déduire que tout terme de type `int` peut être utilisé dans un contexte où un terme de type `mpz` est attendu. Ainsi, l'arbre ci-dessous utilise cette règle (à l'extrême droite) pour utiliser la variable n dans un contexte `mpz` dans la dérivation associée à l'exemple de la section 2, dans le cas où a, b et n sont de type `int`, et en notant Γ_k l'environnement dans lequel k est associé à l'intervalle $[-2^{31}; 2^{31} - 1]$.

$$\dfrac{\dfrac{\vdash a : \texttt{int} \quad \vdash b : \texttt{int} \quad \dfrac{\Gamma_k \vdash k : \texttt{int} \quad \Gamma_k \vdash k : \texttt{int} \quad \texttt{mpz} = \theta([-2^{94} + 2^{62}; 2^{94}])}{\Gamma_k \vdash k * k : \texttt{mpz} \hookleftarrow \texttt{int}}}{\vdash \backslash\texttt{sum}(a,\ b,\ \backslash\texttt{lambda}\,k;\ k * k) : \texttt{mpz} \hookleftarrow \texttt{int}} \quad \dfrac{\vdash n : \texttt{int}}{\vdash n : \texttt{mpz}}}{\vdash_p \backslash\texttt{sum}(a,\ b,\ \backslash\texttt{lambda}\,k;\ k * k) \leq n : \texttt{int} \hookleftarrow \texttt{mpz}}$$

Par la suite, on suppose que notre typeur a été exécuté sur le programme d'entrée afin d'inférer le type de chaque terme et qu'on dispose ainsi d'une fonction $\mathcal{T}(t)$ nous le fournissant.

5 Génération de code

Cette section présente la génération de code qui repose sur le système de types pour décider s'il faut utiliser des entiers et des opérations machines ou GMP. Nous restreignons cette présentation aux cas de la somme et du produit, celui du dénombrement étant un cas particulier de somme et les autres générant du code moins complexe.

Cadre d'étude. La génération de code est effectuée dans un environnement, appelé *environnement de traduction* et prenant la forme d'une fonction $\Omega : \mathcal{L} \rightharpoonup \mathcal{V}$, qui associe à chaque lieur du programme source une variable fraîche dans le programme généré. La génération de code pour un terme (*resp.* prédicat) est ainsi une fonction $[\![\cdot, \cdot]\!]$ qui associe, à un terme t (*resp.* un prédicat p), dans un environnement Ω, un morceau de code généré. Ce morceau de code est une séquence d'instructions conclue par une affectation du résultat de l'évaluation dans une variable fraîche distinguée. Cette séquence d'instruction peut éventuellement manipuler d'autres variables fraîches, contenant notamment des résultats d'évaluation intermédiaire. Afin de simplifier notre analyse, on gère séparément l'initialisation et la libération des variables de type `mpz` et la transmission du résultat. On se focalise donc sur la liste des instructions générées, hors initialisation et libération, que l'on note $[\![\Omega, t]\!]_{\text{code}}$. Par ailleurs, on note $[\![\Omega, t]\!]_{\text{decl}}$ la liste des variables fraîches de type `mpz` utilisées dans le programme $[\![\Omega, t]\!]_{\text{code}}$, et l'on note $[\![\Omega, t]\!]_{\text{res}}$ la variable fraîche distinguée contenant le résultat de l'évaluation. La façon dont les variables fraîches sont générées et déclarées (non initialisées) dans le programme relève du détail d'implémentation. On suppose donc avoir accès à autant de variables fraîches que nécessaire, chacune correctement déclarée mais non initialisée au préalable. L'accès à une telle variable x est noté $\bar{x}$ pour rappeler cet état de fait.

On définit la fonction init_var (*resp.* init_vars) qui, étant donné une variable (*resp.* un ensemble fini de variables) de type `mpz`, renvoie le code correspondant à sa (*resp.* leur) initialisation de la façon suivante :

$$\text{init\_var}(\mathtt{z}) = \mathtt{mpz\_init}(\mathtt{z});$$
$$\text{init\_vars}(\{\mathtt{z}_1; \ldots; \mathtt{z}_n\}) = \text{init\_var}(\mathtt{z}_1) \ldots \text{init\_var}(\mathtt{z}_n)$$

On définit de manière duale clear_vars(V) qui renvoie le code désallouant un ensemble V de variables de type `mpz`. Ainsi le code généré lors d'une traduction est-il le suivant :

$$[\![\Omega, t]\!] = \text{init\_vars}([\![\Omega, t]\!]_{\text{decl}})\,;\ [\![\Omega, t]\!]_{\text{code}}\,;\ \ \text{clear\_vars}([\![\Omega, t]\!]_{\text{decl}})\,;$$

5.1 Définitions de macros dédiées

La prise en compte combinée des types `int` et `mpz` au moment de la génération de code introduit une explosion combinatoire du nombre de cas à considérer. Considérons par exemple le terme `\product(`t_1`, `t_2`, \lambda x;`t_3`)`. Le type renvoyé par la fonction $\mathcal{T}$ pour ce terme peut-être `int` ou `mpz`, et il en va de même pour chacun des sous-termes t_1, t_2 et t_3. On a donc un total de $2 \times 2 \times 2 \times 2 = 16$ combinaisons différentes à traiter. Une approche trop directe de la génération de code serait donc très fastidieuse, d'autant plus lorsqu'on souhaite la formaliser.

Pour éviter cela, on définit un ensemble de *macros*, introduit à la Figure 7, permettant de générer une instruction en fonction des types considérés. La macro *operation* _assignment est spécialisable pour chacune des quatre opérations arithmétiques élémentaires. Dans ces définitions, Ω est l'environnement de traduction, τ est le type de la variable résultat `v`, et t est un terme arbitraire. Ces macros contiennent des cas indéfinis (marqués par l'instruction `assert false`), qui sont prohibitifs. On s'appuie sur notre traduction de code et sur le système de type, pour s'assurer de ne jamais appeler de macro dans l'un de ces cas prohibitifs.

Lemme 1 (Correction des définitions de macros)**.** *L'ensemble des macros définit à la Figure 7 génère du code ayant une sémantique bien typée et bien définie, c'est-à-dire plus précisément :*

1. *La macro* int_assignment$(\tau, \mathtt{v}, \mathtt{z})$ *assigne à la variable* $\mathtt{v}$ *de type* τ *un entier* $\mathtt{z}$ *de type* `int`*.*

$$\frac{\min_{int} \leq \dot{\mathtt{z}} \leq \max_{int}}{\Delta \models_s \text{int\_assignment}(\tau, \mathtt{v}, \mathtt{z}) \Rightarrow \Delta\{\mathtt{v} \leftarrow \dot{\mathtt{z}}^{\tau}\}}$$

```
int_assignment(τ, v, z) :=
  match τ with :
    case int :
      v = z;
    case mpz :
      mpz_set_int(v, z);

var_assignment(Ω, τ, v, t) :=
  match τ, 𝒯(t) with:
    case int, int :
      v = ⟦Ω, t⟧res;
    case mpz, int :
      mpz_set_int(v, ⟦Ω, t⟧res);
    case mpz, mpz :
      mpz_set_mpz(v, ⟦Ω, t⟧res);
    case int, mpz :
      assert false;
```

```
operation _assignment(Ω, τ, v, t) :=
  match τ, 𝒯(t) with :
    case int, int :
      v = v ◇ ⟦Ω, t⟧res;
    case mpz, mpz :
      mpz_op(v, v, ⟦Ω, t⟧res);
    case mpz, int :
      var_assignment(Ω, mpz, x̄, t);
      mpz_op(v, v, x̄);
    case int, mpz :
      assert false;
```

```
condition(Ω, c, τ, v, t) :=
  match τ, 𝒯(t) with :
    case int, int :
      c = v <= ⟦Ω, t⟧res;
    case mpz, mpz :
      c = mpz_cmp(v, ⟦Ω, t⟧res);
      c = c ≤ 0
    case mpz, int
      var_assignment(Ω, mpz, x̄, t);
      c = mpz_cmp(v, x̄);
      c = c ≤ 0
    case int, mpz
      assert false
```

FIGURE 7 – *Macros* specification.

2. *La macro* var_assignment$(\Omega, \tau, \mathtt{v}, t)$ *assigne à la variable* $\mathtt{v}$ *de type* τ *la traduction du terme* t *dans l'environnement de traduction* Ω.

$$\frac{\mathcal{T}(t) \preccurlyeq \tau}{\Delta \models_s \text{var\_assignment}(\Omega, \tau, \mathtt{v}, t) \Rightarrow \Delta\{\mathtt{v} \leftarrow (\text{unwrap}(\Delta[\![\Omega, t]\!]_{\text{res}}))^{\tau}\}}$$

3. *la macro operation* _assignment$(\Omega, \tau, \mathtt{v}, t)$ *assigne à la variable* $\mathtt{v}$ *de type* τ *le résultat de l'opération appliquée sur* $\mathtt{v}$ *et la traduction du terme* t*, en supposant que* m *et* M *désignent respectivement* $-\infty$ *et* $+\infty$ *dans le cas où* $\tau =$ ***mpz****, et* $\min_{int}$*et* $\max_{int}$*dans le cas où* $\tau =$ ***int****.*

$$\frac{\Delta(\mathtt{v}) = v^{\tau} \qquad m \leq v \diamond \text{unwrap}(\Delta([\![\Omega, t]\!]_{\text{res}})) \leq M}{\Delta \models_s \textit{operation}\ \text{\_assignment}(\Omega, \tau_v, \mathtt{v}, t) \Rightarrow \Delta\{\mathtt{v} \leftarrow v \diamond (\text{unwrap}\,(\Delta([\![\Omega, t]\!]_{\text{res}})))^{\tau}\}}$$

4. *la macro condition*$(\Omega, \mathtt{c}, \tau, \mathtt{v}, t)$ *assigne à la variable* $\mathtt{c}$ *une valeur différente de 0 lorsque la valeur de* $\mathtt{v}$ *de type* τ *est plus petite que celle de la traduction de* t*, et 0 sinon.*

$$\frac{\Delta(\mathtt{v}) = v^{\tau} \quad z \neq 0 \quad v \leq \text{unwrap}(\Delta([\![\Omega, t]\!]_{\text{res}}))}{\Delta \models_s \text{condition}(\Omega, \tau, \mathtt{v}, t) \Rightarrow \Delta\{\mathtt{c} \leftarrow z^{int}\}} \qquad \frac{\Delta(\mathtt{v}) = v^{\tau} \quad v > \text{unwrap}([\![\Omega, t]\!]_{\text{res}})}{\Delta \models_s \text{condition}(\Omega, \tau, \mathtt{v}, t) \Rightarrow \Delta\{\mathtt{c} \leftarrow 0^{int}\}}$$

Notons que tous ces lemmes reposent sur les hypothèses de bonne formation de notre programme, qui assurent que dès lors que l'une des macros est appelée avec comme paramètre une variable v, alors cette variable a été correctement initialisée précédemment, ce qui implique que $\Delta(\mathtt{v})$ soit correctement défini.

5.2 Traduction des annotations en code C

Il nous reste maintenant à définir les morceaux de code correspondant à $[\![\Omega, t]\!]_{\text{code}}$ et $[\![\Omega, t]\!]_{\text{res}}$ (on peut en effet calculer $[\![\Omega, t]\!]_{\text{decl}}$ à partir de $[\![\Omega, t]\!]_{\text{code}}$). La Figure 8 les définit pour les termes mini-ACSL correspondant à une somme (colonne de gauche) et à un produit (colonne de droite). Les autres construction sont omises mais leur traduction est nettement plus simple que les deux présentées ici, tout particulièrement une fois les macros de la section 5.1 introduites. Nous faisons ici l'hypothèse que la partie non présentée est correcte.

Intéressons nous maintenant à la traduction d'une somme $t =$ `\sum(t1,t2,\lambda x;t3)`, sachant que la traduction d'un produit suit exactement le même principe. La somme est traduite

$[\![\Omega, \backslash\texttt{sum (t1, t2, \lambda x; t3)}]\!]_{\text{code}} =$
  $[\![\Omega, \texttt{t1}]\!]_{\text{code}}$;
  $[\![\Omega, \texttt{t2}]\!]_{\text{code}}$;
  $[\![\Omega, 1]\!]_{\text{code}}$;
  var_assignment$(\Omega, \tau_k, \texttt{k}, \texttt{t1})$
  int_assignment$(\tau, \texttt{sum}, 0)$
  condition$(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2})$
  `while` $(\bar{c})$ {
    $[\![\Omega_k, \texttt{t3}]\!]_{\text{code}}$;
    add_assignment$(\Omega_k, \tau, \texttt{sum}, \texttt{t3})$
    add_assignment$(\Omega, \tau_k, \texttt{k}, 1)$
    condition$(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2})$
  }

$[\![\Omega, \backslash\texttt{sum(t1, t2, \lambda x; t3)}]\!]_{\text{res}} =$
  sum

$[\![\Omega, \backslash\texttt{product (t1, t2, \lambda x; t3)}]\!]_{\text{code}} =$
  $[\![\Omega, \texttt{t1}]\!]_{\text{code}}$;
  $[\![\Omega, \texttt{t2}]\!]_{\text{code}}$;
  $[\![\Omega, 1]\!]_{\text{code}}$;
  var_assignment$(\Omega, \tau_k, \texttt{k}, \texttt{t1})$
  int_assignment$(\tau, \texttt{product}, 1)$
  condition$(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2})$
  `while` $(\bar{c})$ {
    $[\![\Omega_k, \texttt{t3}]\!]_{\text{code}}$;
    mult_assignment$(\Omega_k, \tau, \texttt{product}, \texttt{t3})$
    add_assignment$(\Omega, \tau_k, \texttt{k}, 1)$
    condition$(\Omega, \bar{c}, \tau_k, \texttt{k}, \texttt{t2})$
  }

$[\![\Omega, \backslash\texttt{product(t1, t2, \lambda x; t3)}]\!]_{\text{res}} =$
  product

FIGURE 8 – Formalization of code generation.

en un itérateur sur toutes les valeurs entre `t1` et `t2`, *via* une boucle, afin d'effectuer l'opération `t3`. La variable `k` est la variable de contrôle de la boucle modélisant le lieur `x`. Son type est donné par $\tau_k = \mathcal{T}(\texttt{t1}) \sqcup \mathcal{T}(\texttt{t2+1})$, car `k` doit pouvoir varier librement entre `t1` et `t2+1`, qui est sa valeur en sortie de boucle. Les opérations dépendantes de `t3` dans le corps de la boucle doivent être effectuées dans l'environnement $\Omega_k = \Omega\{x \leftarrow k\}$ pour prendre en compte cette liaison. Le lecteur attentif pourra noter que le type τ_k est différent du type τ' calculé pour le lieur par le typeur à la Figure 6. En effet, τ' n'est utilisé que pour typer `t3` et ses bornes, dont les valeurs sont indépendantes de la valeur finale de `k`, à savoir `t2+1`. Dans le cas particulier où la valeur de `t2` est $\max_{\texttt{int}}{}^{\texttt{int}}$, τ_k est donc nécessairement `mpz`, alors que τ' pourrait être `int` si les termes `t1` et `t3` demeurent des entiers suffisament petits en valeur absolue. Enfin, signalons que la traduction de la constante 1 comme un terme est un artefact technique permettant de bénéficier de la définition de $[\![\Omega, 1]\!]_{\text{res}}$ dans le corps de add_assignment. La variable de type `int` ainsi générée, constamment égale à 1, est de toute façon *inlinée* par n'importe quel compilateur.

5.3 Propriétés

Nous souhaitons maintenant analyser la sémantique de notre fonction de traduction, et montrer qu'elle satisfait les bonnes propriétés que l'on attend d'un analyseur à l'exécution.

Définition 1. On dit qu'un environnement sémantique Δ_1 est inclus dans un environnement sémantique Δ_2, et l'on note $\Delta_1 \subseteq \Delta_2$, si, pour toute variable $x \in \mathcal{V}$, $\Delta_1(x) = \Delta_2(x)$ ou $\Delta_1(x) = \bot$.

Théorème 1 (Correction du générateur de code)**.** *Soit trois contextes sémantique $\Delta, \Delta_1, \Delta_2$, un contexte logique Λ et un contexte de traduction Ω. Si les variables définies dans Δ et Λ concordent avec celles définies dans Δ_1, alors la valeur de la variable résultat issue de l'évaluation de la traduction du terme t dans l'environnement résultant de cette même traduction est égale à celle de t. Plus formellement, la règle suivante est admissible :*

$$\frac{\Delta \subseteq \Delta_1 \qquad \Lambda = \text{unwrap} \circ \Delta_1 \circ \Omega \qquad \Delta, \Lambda \models_t t \Rightarrow v}{\Delta_1 \models_e [\![\Omega, t]\!] \Rightarrow \Delta_2 \qquad \Delta_2([\![\Omega, t]\!]_{\text{res}}) = v^{\mathcal{T}(t)}}$$

Théorème 2 (Transparence du générateur de code)**.** *Dans le contexte résultant de l'exécution de la traduction d'un terme, toute variable du programme initiale garde la même valeur que*

dans le contexte précédent l'exécution de cette traduction. Plus formellement, la règle suivante est admissible.

$$\frac{\Delta_1 \models_s [\![\Omega, t]\!] \Rightarrow \Delta_2}{\forall x \in \mathcal{V}, x \notin \mathrm{cod}(\Omega) \Rightarrow \Delta_1(x) = \Delta_2(x)}$$

6 Implémentation et évaluation

Cette section introduit d'abord Frama-C et E-ACSL, avant de présenter une évaluation expérimentale des travaux présentés dans cet article menée avec ces outils.

Frama-C et E-ACSL. Le mécanisme de génération de code et le système de types présentés dans cet article sont implémentés au sein du greffon E-ACSL [24] de la plateforme Frama-C [3] pour l'analyse de code C. E-ACSL [24] est un vérificateur d'assertions à l'exécution qui prend en entrée un programme C étendu avec des annotations ACSL [4] et transforme ces dernières en un code C (typiquement, ceux de la Figure 1), afin de vérifier leur validité à l'exécution. E-ACSL supporte l'intégralité du langage C pris en charge par Frama-C, ainsi qu'un très large fragment du langage ACSL [22], plus large que le spectre couvert par cet article. On peut en particulier mentionner les propriétés mémoires, les quantifications bornées, les casts, les flottants et les nombres rationnels, les propriétés portant sur plusieurs points de programme (par exemple, pour une fonction f, la post-condition `ensures G == \old(G)+1` spécifie que la valeur de `G` en sortie de f est la valeur de `G` en entrée, incrémentée de 1) et les fonctions et prédicats définis par l'utilisateur dont, expérimentalement, ceux récursifs. À part les propriétés mémoires formellement étudiées par D. Ly [16], les autres n'ont encore jamais été formalisées dans le cadre d'E-ACSL, même si les propriétés multi-états sont discutés dans [23].

Aux détails d'implémentation et à la prise en compte des autres constructions près, la traduction effectuée par E-ACSL pour le périmètre considéré dans l'article est fidèle à notre présentation. Elle utilise la bibliothèque GMP pour définir le type `mpz` et les opérations associées. Notre sémantique de mini-GMP est conforme à la spécification en langue naturelle de GMP. E-ACSL fait en outre l'hypothèse que l'implémentation de GMP est correcte vis-à-vis de sa spécification, en se contentant de la vérifier par tests. On peut néanmoins signaler que R. Rieu-Helft s'est déjà intéressé à la preuve formelle des algorithmes implémentés dans GMP [20].

Évaluation expérimentale. Pour évaluer les performances du code généré, nous avons mesuré expérimentalement son temps d'exécution sur des exemples. Afin de quantifier l'impact de l'optimisation lié au typage, nous avons comparé ces temps à ceux obtenus en désactivant cette optimisation, et en n'utilisant que des entiers GMP. Cette évaluation a été menée à partir d'une version de développement de Frama-C, commit Git 8db71ab1, disponible en ligne[7], sur une machine Linux Ubuntu dotée d'une architecture 64-bit, d'un processeur Intel Xeon 2.80 GHz 12 cœurs et de 64Go de RAM. Les résultats présentés ici ont été obtenus, en utilisant l'exécutable Hyperfine et en fixant le nombre minimum d'exécutions à 100. Afin de simuler un nombre variable d'assertions, chacune a été dans une boucle de N itérations. Par ailleurs les assertions sont des sommes ou des produits itérant sur un intervalle de taille R. Les résultats de cette évaluation expérimentale sont présentés Figure 9

On peut remarquer que, pour les faibles valeurs de N et R, la version optimisée est peu utile car les calculs sont instantanés. Néanmoins, lorsque ces valeurs croissent, la différence devient vite significative : la version non optimisée ne passe pas à l'échelle. L'impact des optimisations du système de types est meilleure pour les sommes que pour les produits, en particulier

7. https://git.frama-c.com/pub/frama-c

R \ N	100	1000	10000	100000
100	2.2	2.2	2.1	2.6
1000	2.0	1.7	2.7	2.0
10000	2.5	2.6	1.9	2.0

$\sum_{i=1}^{R} i = \frac{R(R+1)}{2}$, avec optimisation

R \ N	100	1000	10000	100000
100	1.7	4.3	27.5	269.4
1000	4.8	21.3	207.5	2064
10000	20.4	200.5	1978	-

$\sum_{i=1}^{R} i = \frac{R(R+1)}{2}$, sans optimisation

R \ N	100	1000	10000	100000
100	3.4	4.5	18.8	183
1000	7.2	62.5	620.7	6202
10000	654.2	6590	-	-

$\prod_{i=1}^{R} i \geq R$, avec optimisation

R \ N	100	1000	10000	100000
100	3.4	4.5	32.5	319
1000	7.9	73.9	734.0	7214
10000	668	6665	-	-

$\prod_{i=1}^{R} i \geq R$, sans optimisation

FIGURE 9 – Evaluation de performances (en ms).

pour les grandes valeurs R, ce qui s'explique par le fait que les produits génèrent plus rapidement des valeurs entières très grands, requérant nécessairement des entiers GMP pour assurer leur correction, dès lors que le nombre de multiplications à effectuer est important. On peut néanmoins observer un gain peu important mais néanmoins significatif lorsque le programme contient beaucoup d'annotations, ce qui s'explique par le fait que le système de types permet d'utiliser les entiers machine pour les opérations sur les bornes du produit, même si l'opération interne au lambda-terme ne peut pas être optimisé. Pour tester cette hypothèse, nous avons effectué un test similaire (non détaillé ici), mais en utilisant des bornes non représentables dans le type `int` et en itérant $N = 10000$. L'écart entre la version optimisée et celle non optimisée devient alors non significatif, confortant notre hypothèse.

Au-delà de cette évaluation expérimentale, E-ACSL a déjà été utilisé avec succès sur des cas réels, par exemple pour évaluer des propriétés numériques de systèmes réactifs synchrones [25], ou encore garantir des propriétés de sécurité sur une bibliothèque cryptographique [1] ou du code utilisé dans l'avionique [17]. Dans tous ces cas, le système de types était toujours actif, même si son impact sur l'efficacité du code généré n'a pas été spécifiquement mesuré.

7 Conclusion

Cet article a présenté une formalisation d'un générateur de code optimisé pour vérifier à l'exécution des propriétés arithmétiques entières, y compris en présence d'opérateurs d'ordre supérieur comme une somme arbitraire sur un intervalle donné. L'optimisation est guidé par un système de types dédié. Ce générateur est implémenté dans le greffon E-ACSL de Frama-C. Les premières évaluations montrent que cette optimisation est indispensable au passage à l'échelle.

Les perspectives incluent la formalisation du générateur de code pour d'autres constructions que celles présentées dans l'article, comme les nombres rationnels ou les fonctions/prédicats récursifs/inductifs. Ces derniers gagneraient aussi à être typés plus finement qu'ils ne le sont aujourd'hui. Des optimisations additionnelles pourraient en outre être implémentées pour, par exemple, limiter le nombre de variables GMP allouées ou mémoiser des calculs GMP intermédiaires effectués plusieurs fois. Enfin, la formalisation pourrait également être assistée, par exemple à l'aide de Coq, afin de pouvoir extraire un générateur de code prouvé correct.

8 Remerciements

Les auteurs remercient les relecteurs anonymes pour leurs précieux retours.

Références

[1] Gergö Barany and Julien Signoles. Hybrid Information Flow Analysis for Real-World C Code. In *Tests and Proofs (TAP)*, July 2017.

[2] Mike Barnett, Manuel Fähndrich, K. Rustan M. Leino, Peter Müller, Wolfram Schulte, and Herman Venter. Specification and Verification : The Spec# Experience. *Communications of the ACM*, 2011.

[3] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The Dogged Pursuit of Bug-Free C Programs : The Frama-C Software Analysis Platform. *Communications of the ACM*, 2021.

[4] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification.* https://frama-c.com/download/acsl.pdf.

[5] Bernhard Beckert, Michael Kirsten, Jonas Klamroth, and Mattias Ulbrich. Modular Verification of JML Contracts Using Bounded Model Checking. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, 2020.

[6] Yoonsik Cheon. *A runtime assertion checker for the Java Modeling Language.* PhD thesis, Iowa State University, 2003.

[7] Lori. A. Clarke and David S. Rosenblum. A Historical Perspective on Runtime Assertion Checking in Software Development. *SIGSOFT Software Engineering Notes*, 2006.

[8] Will Dietz, Peng Li, John Regehr, and Vikram Adve. Understanding Integer Overflow in C/C++. In *International Conference on Software Engineering (ICSE)*, 2012.

[9] Jean-Christophe Filliâtre and Clément Pascutto. Ortac : Runtime Assertion Checking for OCaml (tool paper). In *International Conference on Runtime Verification (RV)*, 2021.

[10] Alain Giorgetti, Julien Groslambert, Jacques Julliand, and Olga Kouchnarenko. Verification of class liveness properties with Java Modeling Language. *Journal of IET Software*, 2008.

[11] Reiner Hähnle and Marieke Huisman. *Deductive Software Verification : From Pen-and-Paper Proofs to Industrial Tools.* 2019.

[12] Arvid Jakobsson, Nikolai Kosmatov, and Julien Signoles. Rester statique pour devenir plus rapide, plus précis et plus mince. In *Journées Francophones des Langages Applicatifs (JFLA)*, 2015.

[13] Nikolai Kosmatov, Fonenantsoa Maurica, and Julien Signoles. Efficient Runtime Assertion Checking for Properties over Mathematical Numbers. In *International Conference on Runtime Verification (RV)*, 2020.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *JML : A Notation for Detailed Design.* 1999.

[15] Hermann Lehner. *A Formal Definition of JML in Coq and its Application to Runtime Assertion Checking.* PhD thesis, ETH Zurich, 2011.

[16] Dara Ly, Nikolai Kosmatov, Frédéric Loulergue, and Julien Signoles. Verified Runtime Assertion Checking for Memory Properties. In *International Conference on Tests and Proofs (TAP)*, 2020.

[17] Dillon Pariente and Julien Signoles. Static Analysis and Runtime Assertion Checking : Contribution to Security Counter-Measures. In *Symposium sur la Sécurité des Technologies de l'Information et des Communications (SSTIC)*, June 2017.

[18] Guillaume Petiot, Bernard Botella, Jacques Julliand, Nikolai Kosmatov, and Julien Signoles. Instrumentation of annotated C programs for test generation. In *International Conference on Source Code Analysis and Manipulation (SCAM)*, 2014.

[19] Benjamin Pierce. *Types and Programming Languages.* MIT Press, 2002.

[20] Raphaël Rieu-Helft. A Why3 Proof of GMP Algorithms. *Journal of Formalized Reasoning*, 2019.

[21] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis : An Abstract Interpretation Perspective.* 2020.

[22] Julien Signoles. *E-ACSL. Implementation in Frama-C Plug-in E-ACSL.* http://frama-c.com/

download/e-acsl/e-acsl-implementation.pdf.

[23] Julien Signoles. The E-ACSL Perspective on Runtime Assertion Checking. In *International Workshop on Verification and mOnitoring at Runtime EXecution (VORTEX)*, 2021.

[24] Julien Signoles, Nikolai Kosmatov, and Kostyantyn Vorobyov. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper. In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*, September 2017.

[25] Franck Védrine, Maxime Jacquemin, Nikolai Kosmatov, and Julien Signoles. Runtime Abstract Interpretation for Numerical Accuracy and Robustness. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, January 2021.

Vers une traduction de $\mathbb{K}$ en Dedukti

Amélie Ledein[1]*, Valentin Blot[1], Catherine Dubois[2]

[1] Laboratoire Méthodes Formelles, Inria, Université Paris-Saclay
[2] Samovar, ENSIIE

$\mathbb{K}$ est un *framework* sémantique permettant de décrire formellement des sémantiques de langages de programmation. C'est aussi un environnement qui offre différents outils pour aider à la programmation avec les langages spécifiés dans le formalisme. Il est par exemple possible d'exécuter des programmes ou encore de vérifier certaines propriétés sur ceux-ci à l'aide de l'outil KProver. $\mathbb{K}$ repose sur une logique du 1er ordre munie d'une application entre formules et d'opérateurs de point fixe, d'égalité, de typage ainsi que d'un opérateur similaire à l'opérateur "next" des logiques temporelles. Ce dernier opérateur permet d'encoder la sémantique des programmes par la réécriture.

Dedukti est un *framework* logique permettant l'interopérabilité des preuves entre différents outils de preuve formelle. Il possède des plugins d'import et d'export pour des systèmes de preuve aussi divers que Coq, PVS ou encore Isabelle/HOL. Il repose sur le $\lambda\Pi$-calcul modulo théorie, une extension de la théorie des types par l'ajout de règles de réécriture dans la relation de conversion. La flexibilité de ce *framework* logique permet d'encoder de nombreuses théories comme la logique du 1er ordre ou la théorie des types simples.

Dans cet article, nous présentons KaMeLo, un outil de traduction de $\mathbb{K}$ vers Dedukti. Cet outil a pour objectif, à plus long terme, de permettre la vérification des preuves faites au sein de $\mathbb{K}$, et la réutilisation de sémantiques formelles de langages de programmation ainsi que des propriétés sur leurs programmes au sein de nombreux systèmes de preuve, via Dedukti.

1 Introduction

Les méthodes formelles ont pour principal objectif l'obtention d'une plus grande confiance dans les programmes informatiques. Mais avant même de pouvoir vérifier un programme, celui-ci doit être écrit dans un langage de programmation dont il est indispensable de connaître précisément la syntaxe et la sémantique, et donc de disposer, dans un premier temps, d'une formalisation de la sémantique du langage de programmation utilisé pour écrire le programme que nous souhaitons vérifier. De nombreux outils permettent d'écrire des sémantiques formelles, comme par exemple Centaur [12], ASF+SDF [29], Ott [27], Sail [7], Lem [23] ou encore $\mathbb{K}$ [6]. Dans cet article, nous nous intéressons uniquement à ce dernier, puisque actuellement, il existe un grand nombre de sémantiques de langage de programmation écrites en $\mathbb{K}$, comme celles de Java [11], C [20] ou encore JavaScript [24]. Une fois la sémantique d'un langage spécifiée, $\mathbb{K}$ offre la possibilité d'exécuter un programme écrit dans ce langage, mais également la possibilité de vérifier certaines propriétés - exprimées sous la forme de propriétés d'atteignabilité - sur ce programme, à l'aide du prouveur automatique KProver [28].

Nous nous intéressons à la traduction de sémantiques écrites en $\mathbb{K}$ vers Dedukti, un *framework* logique permettant l'interopérabilité des preuves entre différents outils de preuve formelle. Cela permettrait d'exécuter au sein de Dedukti un programme écrit avec le langage formalisé, de vérifier les preuves établies par le KProver, voire faire cette preuve avec Dedukti, si le KProver a échoué, et aussi de vérifier formellement des propriétés sur le langage formalisé avec $\mathbb{K}$.

*L'auteure est financée par Digicosme.

Cet article s'intéresse à la traduction d'une sémantique écrite en 𝕂 vers Dedukti, afin de pouvoir exécuter dans Dedukti des programmes écrits dans le langage décrit par cette sémantique. 𝕂 offre de nombreuses facilités pour écrire une sémantique, comme par exemple des attributs pour spécifier une stratégie d'évaluation. La sémantique écrite par l'utilisateur est traduite dans le langage intermédiaire nommé Kore, qui sera notre format d'entrée pour la traduction vers Dedukti. Nos contributions, présentées dans cet article résident, tout d'abord, dans l'étude systématique du langage intermédiaire Kore. En effet, aucun article n'a été encore publié sur ce sujet, et très peu de documentation existe à ce jour. Nous avons donc échangé avec l'équipe 𝕂 pour conforter notre compréhension de Kore. De plus, nous formalisons la transformation effectuée par KaMeLo, un outil en cours de développement permettant de traduire une sémantique écrite en 𝕂 dans Dedukti. La correction de la traduction n'est pas démontrée dans cet article. Informellement, notre traduction cherche à assurer que le programme exécuté dans le framework 𝕂 et le programme exécuté dans Dedukti ont le même comportement - par exemple, calculent la même valeur ou donnent le même état final - si le langage décrit est déterministe.

La vérification des objets de preuve générés par le KProver, ainsi que l'encodage des fondements théoriques de 𝕂 dans ceux de Dedukti ne sont pas traités dans cet article et feront l'objet de travaux futurs. La traduction présentée ici est néanmoins nécessaire pour exécuter un programme et sera réutilisée pour la vérification des preuves.

Après une brève présentation du λΠ-calcul modulo théorie (Section 2) et de son implémentation Dedukti (Section 3), nous présentons la logique sous-jacente de 𝕂, la Matching Logic (Section 4), ainsi que le framework 𝕂 plus en détails, en formalisant un langage impératif nommé IMP (Section 5). Nous expliquons ensuite comment une sémantique 𝕂 est traduite dans Kore qui est une théorie de la Matching Logic (Section 6). Enfin, nous présentons l'outil KaMeLo qui effectue la traduction de Kore vers Dedukti (Section 7).

Dans la suite, les mots-clefs 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`, le langage 𝕂 par une `couleur orange`, le langage de Kore par une `couleur rouge`, et le langage de la Matching Logic par une `couleur verte`. Celles-ci facilitent la lecture, mais ne sont pas nécessaires à la compréhension.

2 Le λΠ-calcul modulo théorie

Le λΠ-calcul modulo théorie, abrégé $\lambda\Pi\equiv_{\mathcal{T}}$ par la suite, est un *framework* logique, c'est-à-dire un *framework* permettant de définir des théories, introduit par Cousineau et Dowek [17]. Celui-ci est une extension du λΠ-calcul, avec une notion primitive de calcul définie à l'aide de règles de réécriture [18]. Actuellement, de nombreuses théories ont pu être encodées dans ce framework comme, par exemple, le Calcul des Constructions [10]. La syntaxe, ainsi que les règles de typage[1] qui régissent le $\lambda\Pi\equiv_{\mathcal{T}}$ sont disponibles en figure 1, où le jugement de typage $\Gamma \vdash t : A$ signifie que le terme t a le type A sous le contexte Γ. Le jugement spécifique $\Gamma \vdash A : \mathbf{Type}$ indique que A est un type sous le contexte Γ. Nous considérons également une signature Σ, ainsi qu'un ensemble de règles de réécriture $\mathcal{R}$. Dans ce cadre, toute règle de réécriture $l \hookrightarrow r \in \mathcal{R}$ vérifie $Var(r) \subseteq Var(l)$, où $Var(p)$ est l'ensemble des variables de p, et l'utilisation d'une telle règle de réécriture nécessite que les instanciations $l\sigma$ et $r\sigma$ de ses membres gauche et droit soient toutes deux bien typées, avec le même type ($\Gamma \vdash l\sigma : A$ et $\Gamma \vdash r\sigma : A$ pour un certain type A).

1. D'après l'isomorphisme de Curry-Horward, ces règles constituent également un système de preuve.

Syntaxe	s	$:=$	**Type** $\mid$ **Kind**	*sorte*
	t	$:=$	$s \mid c \mid x \mid t\ t \mid \lambda(x:t).t \mid \Pi(x:t).t$	*terme*
	Γ	$:=$	$\emptyset \mid \Gamma, x:t$	*contexte*
		avec	c une constante appartenant à Σ,	
			x une variable	

Typage

$$\text{(sort)}\ \frac{}{\Gamma \vdash \textbf{Type} : \textbf{Kind}} \qquad \text{(const)}\ \frac{\Gamma \vdash A : \textbf{Type}}{\Gamma \vdash c : A}\ (c:A) \in \Sigma \qquad \text{(var)}\ \frac{\Gamma \vdash A : \textbf{Type}}{\Gamma \vdash x : A}\ (x:A) \in \Gamma$$

$$\text{(app)}\ \frac{\Gamma \vdash f : \Pi(x:A).B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x \backslash a\}} \qquad \text{(abs)}\ \frac{\Gamma \vdash \Pi(x:A).B : s \quad \Gamma, x:A \vdash b : B}{\Gamma \vdash \lambda(x:A).b : \Pi(x:A).B}$$

$$\text{(prod)}\ \frac{\Gamma \vdash A : \textbf{Type} \quad \Gamma, x:A \vdash B : s}{\Gamma \vdash \Pi(x:A).B : s} \qquad \text{(conv)}\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ A \equiv_{\beta\mathcal{R}} B$$

$$(\equiv_{reduc})\ \frac{}{\Gamma \vdash (\lambda(x:A).t)\ u \equiv t\{x \backslash u\}} \qquad (\equiv_{rule})\ \frac{\Gamma \vdash l\sigma : A \quad \Gamma \vdash r\sigma : A}{\Gamma \vdash l\sigma \equiv r\sigma}\ l \hookrightarrow r \in \mathcal{R}$$

où $s \in \{\textbf{Type}\ ; \textbf{Kind}\}$, $B\{x \backslash a\}$ désigne la substitution de a à x dans B et $\equiv_{\beta\mathcal{R}}$ est la fermeture réflexive, transitive, symétrique et contextuelle de $\equiv$, générée par les règles $\equiv_{reduc}$ et $\equiv_{rule}$.

Figure 1 – Syntaxe et typage du $\lambda\Pi\equiv_{\mathcal{T}}$ avec une signature Σ et des règles de réécriture $\mathcal{R}$

Notons que dans la règle de conversion (conv), la relation d'équivalence dépend non seulement de la β-réduction mais aussi du système de réécriture $\mathcal{R}$. De plus, afin d'avoir la décidabilité du typage, la condition $A \equiv_{\beta\mathcal{R}} B$ dans la règle (conv) doit être décidable, ce qui est assuré lorsque les systèmes de réécriture considérés sont confluents et terminent. Enfin, les contextes peuvent contenir des éléments mal formés et l'ordre des éléments n'a pas d'importance. La bonne formation des éléments du contexte n'est assurée que lors de leur utilisation, dans la règle (var). Cette présentation a été démontrée équivalente aux présentations usuelles dans [19].

3 Un framework pour la logique : Dedukti

Dedukti [9, 2, 3] est un *framework* logique basé sur le $\lambda\Pi\equiv_{\mathcal{T}}$. Ces fondements logiques permettent d'encoder au sein de Dedukti des logiques très expressives comme le Calcul des Constructions Inductives, en les définissant comme des théories du $\lambda\Pi\equiv_{\mathcal{T}}$. Cela permet d'assurer une interopérabilité des preuves entre différents outils formels comme Coq ou PVS (Figure 2).

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.

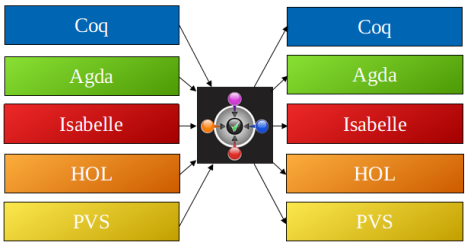


Figure 2 – L'approche de Dedukti, ayant une philosophie similaire à celle de $\mathbb{K}$, mais pour les preuves

Organisation d'un développement en Dedukti. La notion de module n'existe qu'au niveau d'un fichier. Par conséquent, un fichier peut être vu comme un module, mais il n'est pas possible de déclarer un module à l'intérieur d'un fichier. Cependant, il est possible d'importer un ou plusieurs fichiers dans un même fichier, avec les mots-clef `require` et/ou `open`, voire de le renommer avant de l'utiliser par la suite, avec le mot-clef `as`.

Typage et symboles dans Dedukti. La syntaxe du $\lambda\Pi\equiv_{\mathcal{T}}$ est directement accessible dans Dedukti : `TYPE` (**Kind** n'étant pas accessible à l'utilisateur car inféré par le système), λ (abstraction), Π (produit dépendant), mais aussi $\rightarrow$ qui est utilisé lorsque le produit est non-dépendant. La signature peut être définie à partir de symboles constants (`constant`), c'est-à-dire qui ne peuvent pas être réduits par une règle de réécriture. Si la déclaration d'un symbole est faite avec le mot-clef `symbol` seul, le symbole est dit défini, sans propriété particulière.

Règles de réécriture dans Dedukti. Dans Dedukti, une règle de réécriture s'écrit `rule` $LHS \hookrightarrow RHS$ dans laquelle les variables sont notées `$x`, `$y`, etc. Il est possible d'y utiliser un joker (`_`) à gauche lorsqu'une variable n'est pas utilisée dans la partie droite. Pour des raisons d'efficacité, il est possible de déclarer plusieurs règles à la fois à l'aide du mot-clef `with`. 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. Toutes les règles de réécriture définies dans Dedukti constituent un unique ensemble : il n'y a pas de notion de priorité entre ces différentes règles, lorsque nous cherchons la prochaine règle qui s'applique.

4 La Matching Logic

Le framework 𝕂 a d'abord été développé pour faciliter l'écriture de sémantiques, avec l'objectif de ne garder que le meilleur des différents styles de définition sémantique. Ensuite, la recherche de fondements théoriques pour ce framework a donné lieu à différentes logiques qui s'améliorent les unes les autres [26, 28, 16, 15]. Ici, nous nous inspirons de la présentation de la Matching Logic faite à ISR 2021 [5] et de l'article [14]. Ainsi une définition sémantique en 𝕂 peut être considérée comme une théorie de la Matching Logic. D'autres formalisations de la sémantique de 𝕂 ont été proposées dans la littérature. Par exemple, Li et Gunter proposent une formalisation de la sémantique de 𝕂 en Isabelle/HOL [21, 22]. De plus, il existe des travaux sur la définition de la sémantique de 𝕂 en 𝕂 [1], et plus récemment, la Matching Logic a été formalisée à l'aide de Metamath [13].

Syntaxe. La Matching Logic est une logique non typée du 1er ordre qui suit la philosophie « terms as formulae », c'est-à-dire telle qu'il n'y a pas de distinction entre les termes et les formules. En effet, cette logique manipule des *patterns*. Soit Σ un ensemble de symboles constants, nommé également signature. L'ensemble des Σ-patterns est défini à l'aide de huit constructeurs : les variables d'élément (`x`), les variables d'ensemble (`X`), les symboles (σ), l'application ($\varphi_1\ \varphi_2$), les connecteurs propositionnels ($\bot$ et $\rightarrow$), le quantificateur existentiel ($\exists x.\varphi$) et l'opérateur de plus petit point fixe ($\mu X.\varphi$, où il n'y a pas d'occurrences négatives de X dans φ).

Notations. Il est possible d'introduire quelques notations, sans étendre l'expressivité de la logique. Nous avons regroupé différentes notations usuelles à la figure 3, en notant $\varphi[\psi/v]$ la substitution du pattern ψ à v, une variable d'élément ou d'ensemble, dans le pattern φ.

Sémantique. Le nom de la logique est étroitement lié à la notion de *pattern-matching*, d'où le vocabulaire et la sémantique associés. Intuitivement, un pattern φ est interprété par l'ensemble des éléments qu'il *matche*. La sémantique associée à chaque constructeur de pattern est donc étroitement liée à des opérations usuelles sur les ensembles. Nous notons Σ-modèle, un triplet $(M, @_M, \{\sigma_M\}_{\{\sigma\in\Sigma\}})$ où M est un ensemble support non vide, $@_M$ est l'interprétation du pattern applicatif, de type $M \times M \rightarrow \mathcal{P}(M)$ et σ_M est l'interprétation des symboles telle que

Constructeur	Sémantique
$\mathtt{x}$	$\lvert x\rvert_{M,\rho} = \{\rho(x)\}$
$\mathtt{X}$	$\lvert X\rvert_{M,\rho} = \rho(X)$
σ	$\lvert\sigma\rvert_{M,\rho} = \sigma_M$
$\varphi_1\ \varphi_2$	$\lvert\varphi_1\ \varphi_2\rvert_{M,\rho} = \bigcup_{\substack{a_1\in\lvert\varphi_1\rvert_{M,\rho}\\ a_2\in\lvert\varphi_2\rvert_{M,\rho}}} a_1\ @\ a_2$
$\bot$	$\lvert\bot\rvert_{M,\rho} = \emptyset$
$\to$	$\lvert\varphi_1\ \to\ \varphi_2\rvert_{M,\rho} = M \setminus (\lvert\varphi_1\rvert_{M,\rho} \setminus \lvert\varphi_2\rvert_{M,\rho})$
$\exists x.\varphi$	$\lvert\exists x.\varphi\rvert_{M,\rho} = \bigcup_{a\in M} \lvert\varphi\rvert_{M,\rho[a/x]}$
$\mu X.\varphi$	$\lvert\mu X.\varphi\rvert_{M,\rho} = \mathbf{lfp}(A \mapsto \lvert\varphi\rvert_{M,\rho[a/X]})$

Notation	Définition
$\neg\varphi_1$	$\varphi_1\ \to\ \bot$
$\top$	$\neg\bot$
$\varphi_1\ \vee\ \varphi_2$	$\neg\varphi_1\ \to\ \varphi_2$
$\varphi_1\ \wedge\ \varphi_2$	$\neg(\neg\varphi_1\ \vee\ \neg\varphi_2)$
$\varphi_1\ \leftrightarrow\ \varphi_2$	$(\varphi_1\ \to\ \varphi_2)\ \wedge\ (\varphi_2\ \to\ \varphi_1)$
$\forall x.\varphi$	$\neg\exists x.\neg\varphi$
$\nu X.\varphi$	$\neg\mu X.\neg\varphi[\neg X/X]$

Figure 3 – Syntaxe, notation et sémantique de la Matching Logic

$\forall\sigma \in \Sigma.\ \sigma_M \subseteq M$. Cette définition montre que la Matching Logic a une interprétation ensembliste, alors que la logique des prédicats a une interprétation fonctionnelle, ce qui en fait un cas particulier, en considérant qu'il y a une bijection entre a et $\{a\}$. Les sémantiques, notées $\lvert\ .\ \rvert_{M,\rho}$, pour les constructeurs sont données à la figure 3, où ρ une valuation telle que $\rho(x) \in M$ pour toute variable d'élément x et $\rho(X) \subseteq M$ pour toute variable d'ensemble X. et **lfp** est une fonction de plus petit point fixe. Les sémantiques pour les notations peuvent être retrouvées par le calcul.

Théories. Tout comme le $\lambda\Pi\equiv_{\mathcal{T}}$, la Matching Logic est une logique permettant de raisonner modulo une théorie. Il est donc tout à fait possible d'ajouter à la Matching Logic la théorie de l'égalité à l'aide du symbole $\lceil\_\rceil$ (*definedness symbol*), la théorie des sortes à l'aide du symbole $[\![\_]\!]$ (*inhabitant symbol*) ou encore la théorie de la réécriture à l'aide de la sorte $State$ et du symbole $\bullet$ (*one-path next symbol*). La sémantique de ce dernier symbole repose sur le fait que les règles de réécriture définissent une relation binaire $\mathcal{R}$ entre des états, qui sont de type $State$, c'est-à-dire qu'elles définissent un système de transition.
Les définitions et sémantiques nécessaires sont présentées en figure 4, où $[\![s]\!]_M$ est l'ensemble support de s dans M.

Théorie	Symbole	Sémantique	Axiome
Égalité	$\lceil\varphi\rceil$	$\lvert\lceil\varphi\rceil\rvert = M$ *si* $\lvert\varphi\rvert \neq \emptyset$, $\lvert\lceil\varphi\rceil\rvert = \emptyset$ *sinon*	$\forall x.\ \lceil x\rceil$
Sortes	$[\![s]\!]$	$\lvert[\![s]\!]\rvert = [\![s]\!]_M$	
Réécriture	$State$	$State$ est une sorte donc $\lvert[\![State]\!]\rvert = [\![State]\!]_M$	
	$\bullet\varphi$	$\lvert\bullet\varphi\rvert = \{s \in [\![State]\!] \mid \exists t \in \lvert\varphi\rvert$ *tel que* $s \hookrightarrow t \in \mathcal{R}\}$	

$\lfloor\varphi\rfloor \equiv \neg\lceil\neg\varphi\rceil \quad \varphi_1 \subseteq \varphi_2 \equiv \lfloor\varphi_1 \to \varphi_2\rfloor \quad \varphi_1 = \varphi_2 \equiv \lfloor\varphi_1 \leftrightarrow \varphi_2\rfloor \quad x \in \varphi \equiv \lceil x \wedge \varphi\rceil \quad \neg_s\varphi \equiv (\neg\varphi) \wedge [\![s]\!]$

$\forall x : s.\varphi \equiv \forall x.(x \in [\![s]\!]) \to \varphi \quad \varphi_1 \hookrightarrow \varphi_2 \equiv \varphi_1 \to \bullet\varphi_2 \quad \diamond\varphi \equiv \mu X.\varphi \vee \bullet X \quad \varphi_1 \hookrightarrow^* \varphi_2 \equiv \varphi_1 \to \diamond\varphi_2$

Figure 4 – Extensions de la Matching Logic

Ces trois théories constituent la théorie nommée Kore, où $State$ est le type des configurations.

Système de preuve. Le système de preuve de la Matching Logic est présenté en figure 5, où C, C_1, C_2 sont des contextes applicatifs, respectant la grammaire $C ::= \Box \mid C\ \varphi \mid \varphi\ C$.
A travers l'axiome (Prop 3), nous constatons que la Matching Logic est une logique classique.

FOL Reasoning

$$\frac{}{\varphi \to (\psi \to \varphi)} \text{ (Prop 1)}$$

$$\frac{}{(\varphi \to (\psi \to \theta)) \to ((\varphi \to \psi) \to (\varphi \to \theta))} \text{ (Prop 2)}$$

$$\frac{}{((\varphi \to \bot) \to \bot) \to \varphi} \text{ (Prop 3)}$$

$$\frac{\varphi_1 \quad \varphi_1 \to \varphi_2}{\varphi_2} \text{ (Modus Ponens)}$$

$$\frac{}{\varphi[y/x] \to \exists x.\varphi} \text{ (∃-Quantifier)}$$

$$\frac{\varphi_1 \to \varphi_2 \quad (when\ x \notin FV(\varphi_2))}{(\exists x.\varphi_1) \to \varphi_2} \text{ (∃-Generalization)}$$

Technical rules

$$\frac{}{\exists x.x} \text{ (Existence)}$$

$$\frac{}{\neg(C_1[x \wedge \varphi] \wedge C_2[x \wedge \neg\varphi])} \text{ (Singleton)}$$

Fixpoint Reasoning

$$\frac{\varphi_1 \to \varphi_2}{C[\varphi_1] \to C[\varphi_2]} \text{ (Framing)}$$

$$\frac{\varphi}{\varphi[\psi/X]} \text{ (Set Variable Substitution)}$$

$$\frac{}{\varphi[(\mu X.\varphi/X)] \to \mu X.\varphi} \text{ (PreFixpoint)}$$

$$\frac{\varphi[\psi/X] \to \psi}{\mu X.\varphi \to \psi} \text{ (Knaster-Tarski)}$$

Frame Reasoning

$$\frac{}{C[\bot] \to \bot} \text{ } (Propagation_\bot)$$

$$\frac{}{C[\varphi_1 \vee \varphi_2] \to C[\varphi_1] \vee C[\varphi_2]} \text{ } (Propagation_\vee)$$

$$\frac{(when\ x \notin FV(C))}{C[\exists x.\varphi] \to \exists x.C[\varphi]} \text{ } (Propagation_\exists)$$

Figure 5 – Système de preuve de la Matching Logic

5 Un framework pour la sémantique : 𝕂

Le framework 𝕂 [6] permet de définir des sémantiques formelles de langages de programmation et de générer automatiquement des outils pour un langage à partir de sa sémantique formelle (Figure 6). Créé par Grigore Rosu en 2003 à l'Université de Champaign-Urbana (USA - Illinois), 𝕂 est aujourd'hui maintenu par l'entreprise Runtime Verification.
Actuellement, les principaux backends sont le backend LLVM pour l'exécution, et le backend Haskell pour l'exécution symbolique.

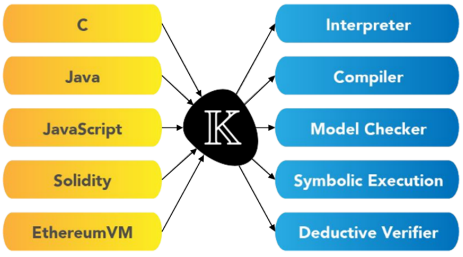


Figure 6 – L'approche de 𝕂

Il est possible de considérer 𝕂 comme une sur-couche orientée notation au-dessus de la logique de réécriture, spécialisée et optimisée pour écrire les définitions de langages et de modèles de programmation complexes. Pour illustrer les fonctionnalités de 𝕂, nous allons montrer comment écrire la sémantique d'un petit langage impératif que nous nommons IMP. Pour une présentation plus complète, le lecteur peut se rapporter à [25].

Organisation d'un développement en 𝕂. Contrairement à Dedukti, la notion de fichier est disjointe de la notion de module puisqu'un fichier peut contenir plusieurs modules (`module`/`endmodule`). De plus, il est possible d'importer des fichiers dans un autre fichier (`requires`), ou encore d'importer un ou plusieurs modules dans un autre module (`imports`).

5.1 Définir la syntaxe d'un langage dans 𝕂

Définir la syntaxe de IMP dans 𝕂 s'apparente à l'écriture d'une grammaire BNF, comme le montre la partie gauche de la colonne « Syntaxe » de la figure 7. La partie droite de cette même colonne contient des attributs, que nous expliquons au fur et à mesure. Dans la suite, un symbole terminal désignera un symbole, pouvant être *mixfix*, écrit entre guillemets, comme par exemple _"+"_, ou encore "`while`"_"`do`"_. Tout ce qui n'est pas entre guillemets sera donc un symbole non terminal.
Pour rendre la syntaxe précédente analysable, il est possible d'utiliser des attributs permettant de préciser la précédence des symboles (`prec(nb)`), ainsi que l'associativité (`left`, `right`, `non-assoc`), ou encore d'ajouter des parenthèses au langage, à l'aide de l'attribut `bracket`. Il sera alors possible de générer un analyseur syntaxique pour IMP.

5.2 Définir la sémantique d'un langage dans 𝕂

Pour définir la sémantique dynamique associée à chaque élément de la syntaxe, il est possible d'utiliser ce qui a déjà été défini dans la bibliothèque standard, mais aussi des configurations, des règles de réécriture et des attributs. La sémantique de IMP est définie à la colonne « Sémantique » de la figure 7.

Syntaxe		Sémantique	n°
`syntax` **AExp** ::= `Int` | `Id`		`rule` ⟨⟨ $x \curvearrowright s$ ⟩$_k$ ⟨ m $x \mapsto i$ ⟩$_{env}$⟩ `=>` ⟨⟨ $i \curvearrowright s$ ⟩$_k$ ⟨ m $x \mapsto i$ ⟩$_{env}$⟩	1
| **AExp** "/" **AExp**	[`left`, `strict`]	`rule` i_1/i_2 `=>` i_1 $/_{Int}$ i_2 `requires` $i_2 \neq_{Int} 0$	2
> **AExp** "+" **AExp**	[`left`, `strict`]	`rule` $i_1 + i_2$ `=>` i_1 $+_{Int}$ i_2	3
| "(" **AExp** ")"	[`bracket`]		
`syntax` **BExp** ::= `Bool`			
| **AExp** "<" **AExp**	[`seqstrict`]	`rule` $i_1 < i_2$ `=>` i_1 $<_{Int}$ i_2	4
| "`not`" **BExp**	[`strict`]	`rule` `not` t `=>` $\neg_{Bool}$ t	5
> **BExp** "`and`" **BExp**	[`left`, `strict(1)`]	`rule` $true$ `and` b `=>` b	6
| "(" **BExp** ")"	[`bracket`]	`rule` $false$ `and` _ `=>` $false$	7
`syntax` **Stmt** ::= "{" "}"		`rule` { } `=>` .	8
| "{" **Stmt** "}"		`rule` { s } `=>` s	9
| `Id` "=" **AExp** ";"	[`strict(2)`]	`rule` ⟨⟨ $x = i;$ $\curvearrowright$ s ⟩$_k$ ⟨ m $x \mapsto$ _ ⟩$_{env}$⟩ `=>` ⟨⟨ s ⟩$_k$ ⟨ m $x \mapsto i$ ⟩$_{env}$⟩	10
| "`if`" **BExp** "`then`" **Stmt** "`else`" **Stmt**	[`strict(1)`]	`rule` `if` $true$ `then` s `else` _ `=>` s	11
		`rule` `if` $false$ `then` _ `else` s `=>` s	12
| "`while`" **BExp** "`do`" **Stmt**		`rule` `while` b `do` s `=>` `if` b `then` {s `while` b `do` s} `else` { }	13
> **Stmt** **Stmt**	[`left`]	`rule` s_1 s_2 `=>` s_1 $\curvearrowright$ s_2	14
`syntax` **Ids** ::= `List{Id, ","}`		`rule` `var` $.Ids$; s `=>` s	15
`syntax` **Pgm** ::= "`var`" **Ids** ";" **Stmt**		`rule` ⟨⟨ `var` x, xl; s ⟩$_k$ ⟨ m ⟩$_{env}$⟩ `=>` ⟨⟨ `var` xl; s ⟩$_k$ ⟨ m $x \mapsto 0$ ⟩$_{env}$⟩ `requires` $x \notin m$	16

Figure 7 – Syntaxe et sémantique de notre langage IMP

Ce qui est natif dans 𝕂, la bibliothèque standard. La bibliothèque standard de 𝕂[2] définit des sortes courantes comme `Id`, `Int` ou encore `Bool` (avec les constantes $true$ et $false$), mais aussi des opérateurs usuels suffixés par une sorte, comme l'égalité sur les entiers (`==Int`), la diségalité sur les entiers (`=/=Int`) ou encore la conjonction booléenne (`andBool`). De plus, cette bibliothèque définit les sortes particulières pour les listes (`List`), les ensembles (`Set`), les multi-ensembles (`Bag`) et les dictionnaires (`Map`). Par convention, les éléments neutres pour la concaténation commencent par `.` dans 𝕂. Par exemple, `.Map` correspond à un dictionnaire vide, tandis que `.Ids` correspond à une liste vide de type `Ids`.

2. Disponible ici : https://kframework.org/k-distribution/include/kframework/builtin/.

Pour finir, la sorte `K` indique qu'un élément de type `K` est calculatoire : tout ce qui est défini dans 𝕂 est de type `K`. Les *K computations* sont un cas particulier de contenu calculatoire, donc de type `K`, particulièrement utiles pour définir des stratégies d'évaluation. Formellement, une K computation est une liste, potentiellement imbriquée et ayant pour constructeurs $\curvearrowright$ et `.`, de calculs à réaliser de manière séquentielle. Par exemple, la K computation x = 10 ; $\curvearrowright$ y = 42 ; modélise le fait que nous affectons d'abord une valeur à la variable x avant de modifier y.

Configurations. Formellement, une configuration est un multi-ensemble de cellules étiquetées, potentiellement imbriquées. Plus concrètement, une configuration contient toutes les informations sémantiques nécessaires pour exécuter un programme : nous pouvons voir une configuration comme un état du programme, qui stocke par exemple le programme à exécuter, les valeurs courantes des variables ou encore la pile. Pour le langage IMP, nous nous contentons d'une configuration à deux cellules, l'une étiquetée par k contenant le programme à exécuter, l'autre étiquetée par *env* contenant les valeurs courantes des variables. Pour définir une telle configuration en 𝕂, il suffit de spécifier la configuration initiale souhaitée, comme par exemple `configuration` <k>`$PGM` :**Pgm**</k> <env>`.Map`</env> qui indique que le programme à exécuter (`$PGM`) doit être mis dans la cellule k et que l'environnement est initialement vide. Dans la suite de cet article, et en accord avec les notations usuelles utilisées dans les articles sur 𝕂, nous allégeons la notation pour décrire une configuration, comme $\langle\langle\ x = 10;\ \rangle_k\ \langle\ x \mapsto 0\ \rangle_{env}\rangle$.

Règles de réécriture. Dans 𝕂, une règle de réécriture, notée `rule` *LHS* `=>` *RHS*, est du 1er ordre et s'applique sur des configurations. Par exemple, la règle suivante signifie que la variable x est évaluée à i puisque la valeur associée à x dans l'environnement, ici la cellule *env*, est i : `rule` $\langle\langle\ x \curvearrowright s\ \rangle_k\ \langle\ m\ \ x \mapsto i\ \rangle_{env}\rangle$ `=>` $\langle\langle\ i \curvearrowright s\ \rangle_k\ \langle\ m\ \ x \mapsto i\ \rangle_{env}\rangle$ (règle n°1 - Figure 7).

Comme dans Dedukti, les règles de réécriture de 𝕂 peuvent être non-linéaires, et ne s'appliquent pas forcément en tête de terme, c'est-à-dire sur une configuration entière. C'est le cas de la règle n°3 (Figure 7) qui donne la sémantique de l'opérateur "+". Dans ce cas de figure, le reste de la configuration est inféré par 𝕂 lors de la compilation en Kore. Cette écriture permet de faire des raisonnements locaux, mais permet surtout d'être plus modulaire. Si l'utilisateur souhaite qu'une règle de réécriture puisse être appliquée n'importe où comme dans Dedukti, il lui suffit d'ajouter l'attribut `anywhere` à cette règle. De plus, ce qui n'est pas utilisé dans le membre droit peut être omis à l'aide d'un joker, comme aux règles n°7, 10 et 12 (Figure 7).

Contrairement à Dedukti, une règle de réécriture peut être conditionnelle dans 𝕂, comme le montre la règle n°2 (Figure 7) qui définit la sémantique de la division.

Définir une stratégie d'évaluation. Il existe deux attributs pour définir une stratégie d'évaluation, c'est-à-dire préciser l'ordre dans lequel les sous-expressions sont évaluées : `strict`, si elle s'effectue de manière non déterministe, et `seqstrict` si elle s'effectue de manière déterministe, de gauche à droite par défaut. Il est également possible de restreindre la liste des arguments qui doivent être évalués avant l'évaluation de l'instruction globale en donnant une liste de nombres, chacun indiquant la position d'un symbole non terminal. Par exemple, pour définir un ≪ et paresseux ≫, nous pouvons utiliser l'attribut `strict(1)`, ainsi que les deux règles suivantes : `rule` *true* `and` b `=>` b et `rule` *false* `and _ =>` *false* (règles n°11 et n°12 - Figure 7).

Enfin, pour spécifier des stratégies d'évaluation plus complexes, il est également possible d'utiliser des contextes (`context`), comme cela se fait classiquement, mais également des squelettes de contextes (`context alias`), permettant de générer automatiquement des contextes, plutôt que d'écrire systématiquement toujours les mêmes contextes. Nous verrons à la section 6.2 une dernière méthode pour définir une stratégie d'évaluation, à l'aide des K computations.

D'autres attributs. Il existe de très nombreux attributs[3], mais à notre connaissance, il n'existe pas de liste d'attributs exhaustive totalement documentée. Nous présentons, ci-dessous, les derniers attributs qui nous intéressent dans le cadre de ce travail.

Tout symbole est soit un symbole de constructeur (`constructor`), soit un symbole de fonction (`function`), mais pas les deux. Par défaut, un symbole est un symbole de constructeur. Nous verrons que la différence entre ces deux attributs influence la traduction de 𝕂 vers Kore, comme le fait que le reste de la configuration est inféré uniquement si le symbole de tête a l'attribut `constructor`. De plus, il est possible de préciser qu'un symbole est injectif (`injective`) ou encore total (`functional`). Enfin, un symbole peut être manipulé modulo associativité (`assoc`), commutativité (`comm`), unité[4] (`unit`), idempotence (`idem`). Ces quatre derniers attributs permettent de faire de la réécriture modulo ACUI.

Les règles de réécriture, par défaut, ne s'appliquent pas dans un ordre particulier. Cependant, l'attribut `owise` indique qu'une règle ne s'applique que lorsque aucune autre ne s'applique, et l'attribut `priority(`*`nb`*`)` permet de préciser l'ordre d'application des règles.

6 De 𝕂 à la Matching Logic : Kore

Le langage Kore est un langage intermédiaire entre le langage de 𝕂 et le langage de la Matching Logic, permettant ainsi de minimiser l'écart entre le langage qui décrit la sémantique, celui de 𝕂, et le langage qui constitue les fondements logiques de 𝕂, la Matching Logic. La compilation d'une définition sémantique en 𝕂 génère un fichier au format Kore. Notre traduction de 𝕂 en Dedukti, présentée à la section 7, utilise ce format Kore intermédiaire, et non le fichier source écrit en 𝕂. Nous supposons donc correcte la traduction de 𝕂 vers Kore. Cette section explique comment s'effectue cette traduction entre 𝕂 et la Matching Logic, explications qui résultent d'une étude des fichiers Kore générés et de discussions avec des membres de l'équipe de développement, puisque aucun document formalisant cette étape n'a été publié. La formalisation partielle de cette traduction est disponible à la figure 8.

Organisation d'un développement en Kore. Un fichier Kore est composé d'une succession de modules (`module`/`endmodule`), commençant éventuellement par des déclarations d'importation (`import`), elles-mêmes suivies de déclarations commençant par l'un des mot-clefs suivants : `sort`, `hooked-sort`, `symbol`, `hooked-symbol`, `alias`/`where`, `axiom`. Quelle que soit la hiérarchie des fichiers qui constitue la sémantique d'un langage, le processus de compilation vers Kore va tout fusionner dans un unique module. Quatre modules[5] sont également importés par défaut lors de la traduction de 𝕂 vers Kore : `BASIC-K` qui contient les sortes `SortK` et `SortKItem`, correspondant aux sortes `K` et `KItem` dans 𝕂 ; `KSEQ` qui contient les constructeurs des K computations `kseq` et `dotk`, notés $\curvearrowright$ et `.` dans 𝕂 ; `INJ` qui introduit le symbole `inj` pour modéliser des injections et enfin, `K` qui contient principalement les modules précédents. Le module `K` est importé dans le module qui contient la sémantique.

Notion de typage. Lors de la traduction de 𝕂 à Kore, tous les types sont précisés à l'aide d'injections (`inj`), informations de typage que nous utilisons parfois, comme à la section 7.3.

6.1 Traduction de la syntaxe dans Kore

La traduction de 𝕂 vers Kore procède de deux manières différentes selon la forme de `B` dans une déclaration `syntax A ::= B`, comme le montre la figure 8[6]. Si `B` n'a pas de symbole

3. Une liste non exhaustive est présente à la fin de cette page https://kframework.org/USER_MANUAL/.
4. On parle aussi d'*identité*, c'est-à-dire modulo les éléments neutres.
5. Les modules `BASIC-K`, `KSEQ`, `INJ` et `K` sont disponibles ici : https://github.com/kframework/kore/blob/master/src/main/kore/prelude.kore, ou au chemin "/usr/include/kframework/builtin/prelude.md".
6. Dans cette figure, de nombreuses déclarations introduisent des accolades laissées vides. En effet, de très rares cas nécessitent de traduire les paramètres de type mis entre accolades.

Module et importation (I_j^h est de la forme `imports` I_j^h [ ]), $k \geq 1$ et $\forall j.\ n_j \geq 0$

|| `module` I_1^1 ... $I_{n_1}^1$ M_1 `endmodule` ... `module` I_1^k ... $I_{n_k}^k$ M_k `endmodule` $||_{\texttt{kore}}$ =
  || Prelude $||_{\texttt{aux}}$ `module import K` [ ] || M_1 $||_{\texttt{kore}}$ ... || M_k $||_{\texttt{kore}}$ `endmodule` [ ]
|| Prelude $||_{\texttt{aux}}$ = Modules `BASIC-K`, `KSEQ`, `INJ` et `K`

Syntaxe (A est un symbole non terminal)

|| `syntax` A : := B [Attr] $||_{\texttt{kore}}$ =
  || `sort` A{} [ ] $||_{\texttt{aux}}$
  || `sort` B{} [ ] $||_{\texttt{aux}}$
  `axiom` $\varphi_{B \subseteq A}$ [`subsort`]
Si B n'a pas de symbole terminal.

|| `syntax` A : := B [Attr] $||_{\texttt{kore}}$ =
  || `sort` A{} [ ] $||_{\texttt{aux}}$
  || `symbol` B{}$(T_1, ..., T_m)$: A [Attr] $||_{\texttt{aux}}$
Si B est de la forme $X_1\ X_2\ ...\ X_n$ avec $n \geq 1$ et X_i un symbole, et $\forall i \in [1..m].\exists j.\ X_j = T_i$ avec T_i un symbole non terminal.

|| `sort` A{} [ ] $||_{\texttt{aux}}$ =
  `sort` A{} [ ]
  || $Proj_A$ et $Pred_A$ $||_{\texttt{aux}}$
Si A n'est pas dans la bibliothèque de 𝕂.

|| `sort` A{} [ ] $||_{\texttt{aux}}$ =
  `hooked-sort` A{} [ ]
  || $Proj_A$ et $Pred_A$ $||_{\texttt{aux}}$
Si A est dans la bibliothèque de 𝕂.

|| $Proj_A$ et $Pred_A$ $||_{\texttt{aux}}$ =
  || `symbol` project$A${}(...) : A [`projection`] $||_{\texttt{aux}}$
  || `symbol` is$A${}(...) : `SortBool` [`predicate`] $||_{\texttt{aux}}$
Si A ≠ `SortKConfigVar`.

|| `symbol` B{}$(T_1, ..., T_m)$: A [x, ...] $||_{\texttt{aux}}$ =
  `symbol` B {}$(T_1, ..., T_m)$: A [x, ...]
  `axiom` φ_x [x, ...]
Si x ∈ {`assoc`, `comm`, `unit`, `idem` `functional`, `constructor`, `injective`, `projection`, `predicate`, `initializer` }.

Configuration

|| `configuration` C $||_{\texttt{kore}}$ =
  || `sort` Sort$cell${} [ ] $||_{\texttt{aux}}$
  || `symbol` $cell${} [`cell`, ...] $||_{\texttt{aux}}$
  || `symbol` Init$cell${} [`initializer`] $||_{\texttt{aux}}$
  || `sort SortKConfigVar` {} [ ] $||_{\texttt{aux}}$
Où $cell \in C$.
Si `$PGM` est une valeur d'initialisation.

Règle de réécriture

|| `rule` LHS `=>` RHS `requires` $Cond$ [$Attr$] $||_{\texttt{kore}}$ =
  `alias` A{}$(T_1, ..., T_n) : T_e$ `where` A{}$(N_1 : T_1, ..., N_n : T_n)$
    := $\overline{Cond}^{\varphi}$ $\wedge$ $mgCONF(\overline{LHS}^{\varphi})$ [ ]
  `axiom` A{}$(N_1 : T_1, ..., N_n : T_n)$ $\hookrightarrow$
      $\top \wedge mgCONF(\overline{RHS}^{\varphi}))$ [$Attr$]
Si le symbole de tête a l'attribut `constructor`.
Où A est un alias pour LHS, $T_1, ..., T_n$ correspondent aux types des variables de LHS, et T_e est le type de LHS.

|| `rule` LHS `=>` RHS `requires` $Cond$ [$Attr$] $||_{\texttt{kore}}$ =
  `axiom` $\overline{Cond}^{\varphi}$ $\wedge$ $Data$ $\rightarrow$
      $(\overline{LHS}^{\varphi} = (\overline{RHS}^{\varphi} \wedge \top))$ [$Attr$]
Si le symbole de tête a l'attribut `function`.
Où $Data$ décrit des associations entre des variables et des patterns.

Stratégie d'évaluation

|| `syntax` A : := B [`strict`, Attr] $||_{\texttt{kore}}$
|| `syntax` A : := B [`seqstrict`, Attr] $||_{\texttt{kore}}$
|| `context` Co $||_{\texttt{kore}}$
|| `context alias` CA $||_{\texttt{kore}}$
Voir section 6.2 pour des exemples de transformation en K computations.

FIGURE 8 – Règles de traduction de 𝕂 vers KORE

terminal, comme dans `syntax AExp ::= Int`, alors il est généré une sorte `B` ainsi qu'un *axiome de sous-typage* ayant l'attribut `subsort`, précisant que `B` est une sous-sorte de `A`, noté $\varphi_{B \subseteq A}$. Mais si `B` a au moins un symbole terminal, comme dans `syntax AExp ::= AExp "-" AExp`, alors est introduit un symbole `B` qui conserve les attributs précisés par l'utilisateur. De plus, pour chaque sorte générée, sont ajoutés un symbole de projection [7], ayant l'attribut `projection`, permettant de projeter une entité de type `SortK` vers un autre sous-type, ainsi qu'un symbole de prédicat, comme `isBool`, ayant l'attribut `predicate`, indiquant qu'une entité est du type associé au symbole de prédicat.

6.2 Traduction de la sémantique dans Kore

Configurations. La configuration initialement déclarée dans 𝕂 est découpée en plusieurs morceaux, un morceau par cellule. Chaque cellule [8] génère une sorte pour typer la cellule, ainsi qu'un symbole avec au moins l'attribut `cell` pour la représenter. De plus, la configuration initiale précise la valeur d'initialisation de chaque cellule, ce qui génère un symbole avec l'attribut `initializer`, correspondant à cette valeur d'initialisation.

Règles de réécriture. La traduction vers Kore d'une règle de réécriture de la forme `rule` LHS `=>` RHS `requires` C_{pre} $[Attr]$ génère un alias (`alias`/`where`) du LHS utilisé dans un axiome commençant par $\hookrightarrow$, si le symbole de tête possède l'attribut `constructor`. Cet axiome encode la règle de réécriture de la manière suivante : $C_{pre} \wedge LHS \hookrightarrow RHS$ [9]. Si la clause `requires` C_{pre} est omise, alors $\overline{C_{pre}}^{\varphi}$ devient $\top$ lors de la traduction. Si le symbole de tête possède l'attribut `function`, l'encodage est similaire mais l'axiome commence par $\rightarrow$. La formalisation de ces traductions sont disponibles à la figure 8, où la fonction $\overline{f}^{\varphi}$ transforme f en un pattern de la Matching Logic, et la fonction $mgCONF$ calcule le reste de la configuration. Si la règle possède l'attribut `anywhere`, la fonction $mgCONF$ est l'identité.

Stratégies d'évaluation. Nous avons vu en section 5.2 trois manières différentes de définir des stratégies d'évaluation dans 𝕂 : les attributs `strict` et `seqstrict`, les contextes et les contextes alias. Lors de la traduction vers Kore, les attributs `strict` et `seqstrict`, mais aussi les contextes et les contextes alias sont traduits en utilisant les K computations et des *freezers*. Intuitivement, un freezer permet de « geler » le reste de la K computation, qui devient non-modifiable, en attendant que la tête de la K computation soit évaluée. Par la suite, nous notons $(\clubsuit^{nb}_{sym}\ \text{arg})$ un freezer où sym est un symbole, nb le numéro de l'argument dont nous attendons la valeur, et arg la liste des autres arguments. Par exemple, pour calculer (4 + 8) - 10, nous construisons la liste $4 + 8 \curvearrowright (\clubsuit^{1}_{\_}\ 10)$ puisque nous souhaitons d'abord évaluer ce qui se trouve entre parenthèses. Nous obtenons ensuite $12 \curvearrowright (\clubsuit^{1}_{\_}\ 10)$ car nous commençons d'abord par calculer le premier élément de la liste, puis nous « dégelons » le calcul pour obtenir 12 - 10, et enfin 2. Cette notion est inspirée des contextes d'évaluation $C[v]$, et des continuations $v \curvearrowright C$ (passer v à la continuation C).

Dans le cas du « et paresseux » du langage IMP, l'attribut `strict(1)` génère la règle de réécriture `rule` E_1 `and` E_2 `=>` $E_1 \curvearrowright (\clubsuit^{1}_{\texttt{and}}\ E_2)$ `requires` $E_1 \notin$ `KResult`, mais aussi la règle symétrique `rule` $E_1 \curvearrowright (\clubsuit^{1}_{\texttt{and}}\ E_2)$ `=>` E_1 `and` E_2 `requires` $E_1 \in$ `KResult`. La sorte `KResult` est une sous-sorte de `K` qui permet de distinguer les valeurs (finales) des expressions. Pour IMP, nous

7. Sauf pour `SortKConfigVar`, la sorte des variables de configuration de 𝕂, comme la variable `$PGM`.

8. Certaines cellules sont ajoutées par rapport à celles précisées par l'utilisateur, comme la cellule `<generatedTop>` qui encapsule toutes les autres cellules. Le processus décrit est le même, outre que l'attribut `cell` devient `cellFragment`, et l'attribut `initializer` devient `cellOptAbsent`, puisque cette cellule n'a pas de valeur initiale.

9. Les propriétés autorisées par le Kprover sont de la forme `rule` LHS `=>` RHS `requires` C_{pre} `ensures` C_{post} et sont encodées ainsi : $C_{pre} \wedge LHS \hookrightarrow C_{post} \wedge RHS$. Dans le cadre de cet article, C_{post} vaut donc $\top$, comme le montre la figure 8.

définissons cette sorte ainsi `syntax KResult ::= Int | Bool`, c'est-à-dire qu'une valeur finale est soit un entier, soit un booléen. Ainsi, la première règle force l'évaluation de E_1 si E_1 n'est pas une valeur, tandis que la deuxième règle simplifie la K computation puisque son élément de tête est une valeur. La condition $E_1 \in$ `KResult` devient `isKResult` E_1 dans le format Kore. D'autres exemples de traduction des attributs définissant une stratégie d'évaluation utilisés à la figure 7 sont disponibles à la figure 9.

Règles générées par l'attribut `strict`	Règles générées par l'attribut `seqstrict`
`rule` E_1 + E_2 `=>` $E_1 \curvearrowright (\circledast^1_+\ E_2)$ `requires` $E_1 \notin$ `KResult`	`rule` $E_1 < E_2$ `=>` $E_1 \curvearrowright (\circledast^1_<\ E_2)$ `requires` $E_1 \notin$ `KResult`
`rule` $E_1 \curvearrowright (\circledast^1_+\ E_2)$ `=>` E_1 + E_2 `requires` $E_1 \in$ `KResult`	`rule` $E_1 \curvearrowright (\circledast^1_<\ E_2)$ `=>` $E_1 < E_2$ `requires` $E_1 \in$ `KResult`
`rule` E_1 + E_2 `=>` $E_2 \curvearrowright (\circledast^2_+\ E_1)$ `requires` $E_2 \notin$ `KResult`	`rule` $E_1 < E_2$ `=>` $E_2 \curvearrowright (\circledast^2_<\ E_1)$ `requires` $E_2 \notin$ `KResult` $\wedge$ $E_1 \in$ `KResult`
`rule` $E_2 \curvearrowright (\circledast^2_+\ E_1)$ `=>` E_1 + E_2 `requires` $E_2 \in$ `KResult`	`rule` $E_2 \curvearrowright (\circledast^2_<\ E_1)$ `=>` $E_1 < E_2$ `requires` $E_2 \in$ `KResult`

Figure 9 – Traduction des attributs `strict` et `seqstrict` dans Kore

Attributs. Les attributs initialement présents dans le fichier 𝕂 sont *a priori* recopiés dans le fichier Kore. Certains attributs pouvant être associés à un symbole génèrent également un axiome, comme le montre la figure 8. Deux cas particuliers existent. Pour l'attribut `predicate`, deux axiomes sont générés au lieu d'un seul, l'un précisant la sémantique quand le prédicat se réduit à faux, l'autre dans le cas vrai. Pour l'attribut `constructor`, un *axiome d'injectivité* est généré pour chaque symbole ayant cet attribut, ainsi que deux axiomes pour chaque ensemble de constructeurs construisant une valeur d'un même type : un *axiome de non-recouvrement*, précisant que deux valeurs d'un certain type commençant par des constructeurs différents sont différentes ; et un *axiome d'exhaustivité*, précisant que nous ne pouvons construire des valeurs d'un certain type qu'avec les constructeurs associés à ce type.

Tout axiome généré est la formalisation de la sémantique que nous avons décrite précédemment de manière informelle pour chaque attribut associé à un symbole. Nous verrons dans la partie suivante, comment tous ces axiomes sont traduits dans Dedukti.

Enfin, de très nombreux symboles rattachés à la bibliothèque standard de 𝕂 (`hooked-symbol`) sont également ajoutés dans le fichier Kore. Leur règle de traduction est similaire au cas `symbol` : en fonction des attributs qu'ils possèdent, certains axiomes seront générés. Cependant, certains de ces symboles semblent uniquement utiles à des backends particuliers de 𝕂.

7 Traduire Kore dans Dedukti

Cette section s'intéresse à l'outil KaMeLo [4] en cours de développement. Nous présentons une formalisation de la traduction effectuée par cet outil, c'est-à-dire les encodages superficiels réalisés, et formalisés à la figure 10. Une version en 𝕂, en Kore et en Dedukti de IMP, est également disponible avec le code source de KaMeLo.

7.1 Traduction élémentaire

Une très grande majorité des encodages sont assez directs, notamment car 𝕂 et Dedukti partagent un très grand nombre de fonctionnalités, comme les règles de réécriture.

Module et import. Tous les modules sont fusionnés dans un unique fichier, en ajoutant le prélude de Dedukti après la traduction du module `BASIC-K`, puisque les déclarations disponibles dans notre prélude dépendent de la sorte `SortK`.

Sortes et symboles. La très grande majorité des sortes et des symboles traduits dans le fichier Kore découlent des déclarations `syntax` et `configuration` présentes dans 𝕂. Nous constatons que la traduction est très simple : toutes les sortes deviennent des symboles de type `SortK`, sauf la sorte `SortK` elle-même, qui a le type `TYPE`. Pour les symboles, la seule différence est que la signature est currifiée. Les sortes et symboles rattachés à la bibliothèque standard de 𝕂 doivent également être définis dans le prélude de Dedukti : les traductions sont similaires à celles présentées à la figure 10.

Traduire les axiomes. Nous notons X-axiome, un axiome de la Matching Logic commençant par l'opérateur X. Un $\exists$-axiome possède soit l'attribut `subsort` (*axiome de sous-typage*), soit l'attribut `functional` (*axiome de fonctionnalité*). Un $=$-axiome est un axiome équationnel ayant l'attribut `assoc`, `comm`, `unit` ou `idem`. Les $\vee$-axiome et $\bot$-axiome possèdent l'attribut `constructor` et sont des axiomes d'exhaustivité des constructeurs. Un $\neg$-axiome possède l'attribut `constructor` et est un axiome de non-recouvrement des constructeurs, tandis qu'un $\rightarrow$-axiome possédant l'attribut `constructor` est un axiome d'injectivité des constructeurs. Les autres $\rightarrow$-axiomes peuvent avoir l'attribut `initializer`, `projection`, `predicate` ou aucun attribut. Un $\hookrightarrow$-axiome correspond à une règle de réécriture.
Seuls les $\hookrightarrow$-axiome et $\rightarrow$-axiome, n'ayant pas l'attribut `constructor`, sont traduits. Ils ont été générés par une règle de réécriture écrite dans 𝕂, et seront donc traduits par une ou des règles de réécriture en Dedukti. La traduction $||\ .\ ||_{\texttt{CTRS}}$ est expliquée aux sections 7.2 et 7.3.

Importation

$||$ `import` I [ ] $||_{\texttt{dk}} = nothing$

Sorte

$||$ `sort` `SortK`{} [ ] $||_{\texttt{dk}} =$ `constant symbol` $\widehat{\texttt{SortK}} : \texttt{TYPE}$

$||$ `sort` S{} [ ] $||_{\texttt{dk}} =$ `constant symbol` $\widehat{S} : \widehat{\texttt{SortK}}$ — Si $S \neq$ `SortK`.

$||$ `hooked-sort` S{} [ ] $||_{\texttt{dk}} = nothing$ — Si S est définie dans le prélude de Dedukti.

Symbole

$||$ `symbol` $C\{\}(T_1, ..., T_n) : T_e$ [$Attr$] $||_{\texttt{dk}} =$
`symbol` $\widehat{C} : \widehat{T_1} \rightarrow ... \rightarrow \widehat{T_n} \rightarrow \widehat{T_e}$

$||$ `hooked-symbol` $C\{\}$ [ ] $||_{\texttt{dk}} = nothing$ — Si C est définie dans le prélude de Dedukti.

Axiome

$||$ `alias` $A\{\}(T_1, ..., T_n) : T_e$ `where` $A\{\}(N_1 : T_1, ..., N_n : T_n) := Cond \wedge LHS$ [ ]
`axiom` $A\{\}(N_1 : T_1, ..., N_n : T_n) \hookrightarrow \top \wedge RHS$ [$Attr$] $||_{\texttt{dk}} =$

$$\begin{cases} \texttt{rule } \widehat{LHS}^r \hookrightarrow \widehat{RHS}^r & \text{si } Cond = \top. \\ ||\ (\widehat{LHS}^r,\ \widehat{RHS}^r,\ \widehat{Cond}^r)\ ||_{\texttt{CTRS}} & \text{sinon.} \end{cases}$$

$||$ `axiom` Ax [x] $||_{\texttt{dk}} = nothing$ — Si $x \in$ {`subsort`, `functional`, `constructor`, `assoc`, `comm`, `unit`, `idem`}.

$||$ `axiom` $Cond \wedge Data \rightarrow (LHS = (RHS \wedge \top))$ [$Attr$] $||_{\texttt{dk}} =$ — Où $Attr$ peut être vide ou contenir les attributs `projection`, `predicate` ou `initializer`.

$$\begin{cases} \texttt{rule } \widehat{LHS}^{Data} \hookrightarrow \widehat{RHS}^{Data} & \text{si } Cond = \top. \\ ||\ (\widehat{LHS}^{Data},\ \widehat{RHS}^{Data},\ \widehat{Cond}^r)\ ||_{\texttt{CTRS}} & \text{sinon.} \end{cases}$$

où $\widehat{S}$ transforme l'identificateur S de Kore, en un identificateur de Dedukti.
$\widehat{S}^r$ transforme le pattern S en un terme de Dedukti.
$\widehat{S}^{Data}$ transforme le pattern S en un terme de Dedukti, en utilisant les associations entre variables et patterns décrites par $Data$.

Figure 10 – Traduction de Kore vers Dedukti via KaMeLo

7.2 Traduction des règles de réécriture conditionnelles

Dans cette section, nous nous intéressons à la traduction des règles de réécriture conditionnelles. Comme Dedukti ne propose pas la possibilité de définir des règles de réécriture conditionnelles, il est nécessaire de trouver un encodage d'un système de réécriture conditionnelle ($CTRS$) vers un système de réécriture non conditionnelle (TRS).

7.2.1 Encodage sur un exemple

Considérons le système suivant :
(1) `rule` $max\ X\ Y$ `=>` Y `requires` X `<Int` Y
(2) `rule` $max\ X\ Y$ `=>` X `requires` X `>=Int` Y.
L'encodage permet d'obtenir le système de réécriture suivant dans Dedukti :
(0) `rule` max `$x` `$y` $\hookrightarrow$ $\flat max$ `$x` `$y` $\flat\ \flat$
(1′) `rule` $\flat max$ `$x` `$y` $\flat$ `$c` $\hookrightarrow$ $\flat max$ `$x` `$y` (`$x` $<$ `$y`) `$c`
(1″) `rule` $\flat max$ `$x` `$y` $true$ `$c` $\hookrightarrow$ `$y`
(2′) `rule` $\flat max$ `$x` `$y` `$c` $\flat$ $\hookrightarrow$ $\flat max$ `$x` `$y` `$c` (`$x` $\geq$ `$y`)
(2″) `rule` $\flat max$ `$x` `$y` `$c` $true$ $\hookrightarrow$ `$x`.
Avec le système obtenu, $max\ 5\ 3$ se calcule ainsi : $max\ 5\ 3 \hookrightarrow_0 \flat max\ 5\ 3\ \flat\ \flat \hookrightarrow_{1'} \flat max\ 5\ 3\ (5 < 3)\ \flat \hookrightarrow^* \flat max\ 5\ 3\ false\ \flat \hookrightarrow_{2'} \flat max\ 5\ 3\ false\ (5 \geq 3) \hookrightarrow^* \flat max\ 5\ 3\ false\ true \hookrightarrow_{2''} 5$.

L'idée générale de l'encodage, proposé dans cette section et initialement proposée par Viry [30], est d'ajouter, pour un symbole défini avec des règles conditionnelles, autant d'arguments qu'il y a de conditions. La règle (0) réécrit un terme dont le symbole de tête est max, par un terme utilisant la version étendue correspondante d'arité 4, $\flat max$, où tous les arguments booléens valent $\flat$, indiquant que les arguments booléens n'ont pas encore été initialisés par une condition. Les règles (1′) et (2′) initialisent les conditions à calculer, tandis que les règles (1″) et (2″) réduisent la taille du terme puisque une des conditions a été évaluée à $true$.

Contrairement à Viry, nous choisissons d'étendre la signature, comme ici avec le symbole $\flat max$, plutôt que de remplacer chaque symbole de la signature par un symbole équivalent mais avec une arité plus grande, permettant de transporter les calculs des conditions. En effet, cela complique le code et oblige à traduire, après coup, les formes normales obtenues.

De plus, cet encodage a l'avantage de ne pas fixer l'ordre d'évaluation des conditions, mais augmente le temps de calcul, cela en doublant le nombre initial de règles.

Enfin, pour générer les règles précédentes, il faut connaître toutes les conditions qui peuvent s'appliquer, pour un symbole de tête donné. Cependant, de nombreuses règles de réécriture écrites dans 𝕂, c'est-à-dire celles ayant un symbole de constructeur comme symbole de tête et n'ayant pas l'attribut `anywhere`, nécessitent d'inférer la configuration. Cela implique que si nous considérons la définition usuelle d'un symbole de tête, ces règles de réécriture auraient le même symbole de tête. Dans le cadre de cet article, nous considérons donc que le symbole de tête d'une règle de réécriture correspond au symbole de tête de la partie du terme gauche de la règle se trouvant dans la cellule `<k>`, sans considérer les symboles `dotk`, `kseq` et `inj`. Si la règle a un symbole de fonction comme symbole de tête ou possède l'attribut `anywhere`, le symbole de tête correspond au symbole de tête au sens usuel.
Nous notons $head_{<k>}$ la fonction qui renvoie le symbole de tête de la cellule `<k>`, pour une règle donnée, sans considérer les symboles `dotk`, `kseq` et `inj`. Nous notons également $\mathcal{C}_\sigma$, l'ensemble des règles qui partagent le même symbole de tête σ, soit $\mathcal{C}_\sigma = \{\ l \overset{c}{\hookrightarrow} r \mid head_{<k>}\ (l \overset{c}{\hookrightarrow} r) = \sigma\ \}$.

7.2.2 Formalisation de l'encodage

Soit $\mathfrak{R}$ un système de réécriture conditionnelle écrit dans 𝕂. Pour éviter des conflits de nommage, nous supposons également que $\flat$ est un nom de symbole non utilisé, et qu'il n'apparaît en tête d'aucun nom de symbole. Nous présentons la traduction notée $||\ .\ ||_{\texttt{CTRS}}$ précédemment. Celle-ci prend en argument un ensemble $\mathcal{E}_{DK}$ de triplets de termes Dedukti de la forme $(LHS,\ RHS,\ c)$, noté ici $LHS \overset{c}{\hookrightarrow} RHS$, obtenu une fois les traductions $||\ .\ ||_{\texttt{kore}}$ et $||\ .\ ||_{\texttt{dk}}$ appliquées sur $\mathfrak{R}$. Après avoir construit les $\mathcal{C}_\sigma$ à partir de $\mathcal{E}_{DK}$, nous déroulons l'algorithme présenté à la figure 11, pour chaque $\mathcal{C}_\sigma$, où X est le nombre de règles conditionnelles dans $\mathcal{C}_\sigma$.

1. Si $X = 0$, $\mathcal{C}_\sigma$ est inchangé et l'algorithme s'arrête. Sinon, initialiser i à 0 et aller en 2.
2. Générer le LHS le plus général pour un symbole σ donné, noté $mglhs_\sigma$.
3. Générer le symbole étendu $\flat\sigma$ de type $T_1 \to ... \to T_{n-1} \to \flat\texttt{Bool} \to ... \to \flat\texttt{Bool} \to T_n$, avec X argument(s) de type $\flat\texttt{Bool}$, où $\flat\texttt{Bool} = \texttt{Bool} \cup \{\flat\}$, et σ de type $T_1 \to ... \to T_n$.
4. Générer la règle de substitution : $mglhs_\sigma \hookrightarrow mglhs_\sigma[\ update_{same}(\sigma, \flat\sigma, \flat)\ ]_\sigma$
5. Pour chaque règle appartenant à $\mathcal{C}_\sigma$:
   (a) Si la règle est de la forme $l \overset{c}{\hookrightarrow} r \in \mathcal{C}_\sigma$ et n'a pas l'attribut `owise` (avec $c \neq \top$) :
   — Incrémenter i de 1
   — Générer une règle d'initialisation :
   $l\ [\ update_{diff}(\sigma, \flat\sigma, \flat, i, \_)\ ]_\sigma \hookrightarrow l\ [\ update_{diff}(\sigma, \flat\sigma, c, i, \_)\ ]_\sigma$
   — Générer une règle de réduction : $l\ [\ update_{diff}(\sigma, \flat\sigma, \texttt{true}, i, \_)\ ]_\sigma \hookrightarrow r$
   (b) Si la règle est de la forme $l \overset{\top}{\hookrightarrow} r \in \mathcal{C}_\sigma$ et n'a pas l'attribut `owise`, générer la règle de réduction : $l\ [\ update_{same}(\sigma, \flat\sigma, \_)\ ]_\sigma \hookrightarrow r$.
   (c) Si la règle $l \overset{\top}{\hookrightarrow} r \in \mathcal{C}_\sigma$ à l'attribut `owise`, générer la règle de réduction : $l\ [\ update_{same}(\sigma, \flat\sigma, \texttt{false})\ ]_\sigma \hookrightarrow r$.

Où :
* $t_1[t_2]_\sigma$ signifie que nous substituons t_2 au sous-terme ayant pour symbole de tête σ dans t_1,
* $arg_i(t)$ correspond au i-ème argument de t et $arity(t)$ au nombre d'arguments de t,
* $mglhs_\sigma$ désigne la configuration initiale où les cellules les plus profondes deviennent, `<c>` y_i si `<c>` $\neq$ `<k>`, `<k>` (`kseq` (`inj` σ z_1 ... $z_{arity(\sigma)}$) L) sinon, avec y_i et z_i des variables fraîches et, L une K computation.
* $update_{diff}(\sigma, \flat\sigma, s_1, i, s_2) = \flat\sigma\ x_1\ ...\ x_{arity(\sigma)+X}$ $avec$ $x_j = \begin{cases} arg_j(\sigma) & \text{si } 1 \leq j \leq arity(\sigma) \\ s_1 & \text{si } j = arity(\sigma) + i \\ s_2 & \text{sinon} \end{cases}$
* $update_{same}(\sigma, \flat\sigma, s) = \flat\sigma\ x_1\ ...\ x_{arity(\sigma)+X}$ $avec$ $x_j = \begin{cases} arg_j(\sigma) & \text{si } 1 \leq j \leq arity(\sigma) \\ s & \text{sinon} \end{cases}$

Figure 11 – Variante de l'encodage de Viry

7.2.3 Extension de l'encodage à l'attribut `owise`

Une manière plus succincte d'écrire l'exemple précédent est d'utiliser l'attribut `owise` :
`rule` *max X Y* `=>` *Y* `requires` *X* `<Int` *Y*
`rule` *max X Y* `=>` *X* [`owise`]
Malheureusement, 𝕂 ne génère pas la condition complémentaire dans le fichier Kore. Pour encoder cet attribut, deux possibilités s'offrent à nous : implémenter un algorithme qui détermine la condition complémentaire ou considérer que toutes les conditions se réduisent nécessairement soit à `true`, soit à `false`. Comme nous ne connaissons pas exactement l'expressivité des conditions pouvant être écrites dans 𝕂, outre qu'elles sont de type `Bool`, nous préférons ajouter l'hypothèse suivante : toute fonction construisant un booléen est une fonction totale.
Sous cette hypothèse, nous pouvons générer la règle présentée en 5.(c) (Figure 11).

De plus, 𝕂 accepte un ensemble de règles non conditionnelles avec au moins une règle ayant l'attribut `owise`. Nous excluons ce cas car nous ne pouvons pas modéliser « Si aucune autre règle ne s'applique », avec une condition booléenne. Cela est équivalent à l'utilisation de l'attribut `priority(nb)`, que nous ne traitons pas encore actuellement.

7.3 Traduction des stratégies d'évaluation

Comme nous l'avons vu à la section 6.2, il est possible que lors de la traduction de 𝕂 vers Kore, des règles de réécriture conditionnelles soient générées, comme c'est le cas pour les stratégies d'évaluation définies par les attributs `strict` et `seqstrict`. Les règles de réécriture générées par ces attributs nécessitent de traduire les K computations, c'est-à-dire les symboles `.` et $\curvearrowright$, les freezers mais aussi le prédicat `isKResult`. L'encodage de ces différents symboles ne pose donc aucune difficulté puisque les règles présentées à la figure 10 pour les symboles s'appliquent aisément. Cependant, ces règles de réécriture conditionnelles s'inscrivent dans un cas de figure connu où l'encodage de Viry n'est pas confluent, notamment car l'ordre d'application de certaines règles de réécriture modifie le résultat de la condition, ce qui peut bloquer le calcul. En effet, la figure 12 montre, à droite, la traduction dans Dedukti des règles `rule` E_1 `and` E_2 `=>` $E_1 \curvearrowright (\clubsuit^1_{\texttt{and}}\ E_2)$ `requires` $E_1 \notin$ `KResult` (règle C) et `rule` $E_1 \curvearrowright (\clubsuit^1_{\texttt{and}}\ E_2)$ `=>` E_1 `and` E_2 `requires` $E_1 \in$ `KResult` (règle H), générées par l'attribut `strict(1)`, et, à gauche, un exemple d'exécution valide mais bloquée [10].

1	`rule` $E1 *and* $E2 $\curvearrowright$ $c $\hookrightarrow$ ♭*and* $E1 $E2 ♭ $\curvearrowright$ $c	(*true and true*) *and false* $\curvearrowright$.
2	`rule` ♭*and* $E1 $E2 ♭ $\curvearrowright$ $c $\hookrightarrow$ ♭*and* $E1 $E2 (*not*(*isKResult* $E1)) $\curvearrowright$ $c	$\hookrightarrow_1$ ♭*and* (*true and true*) *false* ♭ $\curvearrowright$.
3	`rule` ♭*and* $E1 $E2 *true* $\curvearrowright$ $c $\hookrightarrow$ $E1 $\curvearrowright$ ($\clubsuit^1_{\texttt{and}}$ $E2) $\curvearrowright$ $c	$\hookrightarrow_2$ ♭*and* (*true and true*) *false* (*not*(*isKResult* (*true and true*))) $\curvearrowright$.
4	`rule` $E1 $\curvearrowright$ ($\clubsuit^1_{\texttt{and}}$ $E2) $\curvearrowright$ $c $\hookrightarrow$ (♭$\clubsuit^1_{\texttt{and}}$ $E1 $E2 ♭) $\curvearrowright$ $c	$\hookrightarrow^*$ ♭*and* (*true and true*) *false true* $\curvearrowright$.
5	`rule` (♭$\clubsuit^1_{\texttt{and}}$ $E1 $E2 ♭) $\curvearrowright$ $c $\hookrightarrow$ (♭$\clubsuit^1_{\texttt{and}}$ $E1 $E2 (*isKResult* $E1)) $\curvearrowright$ $c	$\hookrightarrow_3$ (*true and true*) $\curvearrowright$ ($\clubsuit^1_{\texttt{and}}$ *false*) $\curvearrowright$.
6	`rule` (♭$\clubsuit^1_{\texttt{and}}$ $E1 $E2 *true*) $\curvearrowright$ $c $\hookrightarrow$ $E1 *and* $E2 $\curvearrowright$ $c	$\hookrightarrow_4$ (♭$\clubsuit^1_{\texttt{and}}$ (*true and true*) *false* ♭) $\curvearrowright$.
7	`rule` *true and* $b $\curvearrowright$ $c $\hookrightarrow$ $b $\curvearrowright$ $c	$\hookrightarrow_5$ (♭$\clubsuit^1_{\texttt{and}}$ (*true and true*) *false* (*isKResult* (*true and true*))) $\curvearrowright$.
8	`rule` *false and* _ $\curvearrowright$ $c $\hookrightarrow$ *false* $\curvearrowright$ $c	$\hookrightarrow^*$ (♭$\clubsuit^1_{\texttt{and}}$ (*true and true*) *false false*) $\curvearrowright$.

Figure 12 – Règles générées avec l'encodage précédent et une exécution bloquée

Intuitivement, ces règles servent à s'assurer que E_1 est bien d'un certain type, afin d'autoriser ou non son évaluation. L'idée de notre nouvel encodage est de spécialiser les termes de gauche des règles, c'est-à-dire d'affiner le pattern-matching afin d'assurer le type souhaité pour E_1. Les axiomes de sous-typage générés lors de la traduction de 𝕂 vers Kore nous permettent de connaitre la valeur booléenne de `isKResult` E_1 afin d'affiner correctement le pattern-matching. Nous n'appliquons donc plus notre encodage basé sur Viry dans ce cas de figure, puisque les conditions initiales deviennent inutiles.

Nous illustrons ce nouvel encodage sur l'exemple précédent.

La règle H est compilée en une règle de réécriture non conditionnelle en spécialisant le terme gauche en un terme correspondant à un booléen, soit `rule` ($inj\{Bool\}\{KItem\}$ $E1) $\curvearrowright$ ($\clubsuit^1_{\texttt{and}}$ $E2) $\curvearrowright$ $c $\hookrightarrow$ ($inj\{Bool\}\{BExp\}$ $E1) *and* $E2 $\curvearrowright$ $c. Ainsi la règle H ne pourra être utilisée que si le terme est bien une expression booléenne complètement calculée. Pour la règle qui demande l'évaluation de la première expression, la règle C, nous appliquons le même principe et compilons cette règle non conditionnelle en autant de règles conditionnelles qu'il y a de cas possibles dans la sorte **BExp**, le cas de la constante excepté, soit :

1. `rule` ($X1 *and* $X2) *and* $E2 $\curvearrowright$ $c $\hookrightarrow$ ($X1 *and* $X2) $\curvearrowright$ ($\clubsuit^1_{\texttt{and}}$ $E2) $\curvearrowright$ $c
2. `rule` ($X1 < $X2) *and* $E2 $\curvearrowright$ $c $\hookrightarrow$ ($X1 < $X2) $\curvearrowright$ ($\clubsuit^1_{\texttt{and}}$ $E2) $\curvearrowright$ $c
3. `rule` (*not* $X1) *and* $E2 $\curvearrowright$ $c $\hookrightarrow$ (*not* $X1) $\curvearrowright$ ($\clubsuit^1_{\texttt{and}}$ $E2) $\curvearrowright$ $c

Nous ne donnons pas de formalisation de cette traduction, faute de place, mais l'exemple détaillé ci-dessus s'adapte aux autres cas présentés à la figure 9, et la généralisation est aisée.

10. Il est également possible d'obtenir *false*.

8 Conclusion

Dans cet article, nous nous sommes principalement intéressés à la traduction dans Dedukti de sémantiques écrites en 𝕂, afin de pouvoir exécuter dans Dedukti des programmes écrits dans le langage décrit par la sémantique. Nous nous sommes appuyés sur le format Kore, sans chercher à le certifier. Nous supposons donc que le fichier Kore produit depuis une sémantique 𝕂 est correcte. Ce travail a nécessité la compréhension de la traduction d'une sémantique 𝕂 dans une théorie de la Matching Logic, nommée Kore, mais également de pouvoir traduire des règles conditionnelles en règles non conditionnelles. Dans l'état actuel, le traducteur KaMeLo traduit la syntaxe du langage, les configurations et les symboles, certains attributs et les règles de réécriture.

Cependant, 𝕂 autorise une forme restreinte de réécriture modulo ACUI, à l'aide des attributs `assoc`, `comm`, `unit` et `idem`. Dans la bibliothèque standard de 𝕂, de très rares symboles ont l'un de ces attributs, comme les dictionnaires (`Map`) et les ensembles (`Set`). A l'heure actuelle, afin de pouvoir exécuter un programme écrit dans le langage de IMP, nous avons utilisé une implantation plus classique des dictionnaires qui n'utilise pas cette fonctionnalité, et ainsi modifié très légèrement la sémantique initiale [11], comme le montre la figure 13.

Implémentation des dictionnaires	Modification de la sémantique	n°
`syntax` **Dico** ::= "`.Dico`" | "(" `KItem` "`|-->`" `KItem` ")" "," **Dico**		
`syntax` **KItem** ::= "`lookup`" `KItem` **Dico** `rule lookup` x ((y `|-->` $\_$) , d) `=> lookup` x d `requires` $x \neq y$ `rule lookup` x ((y `|-->` i) , $\_$) `=>` i `requires` $x = y$	`rule` $\langle\langle\ x \curvearrowright s\ \rangle_k\ \langle\ d\ \rangle_{env}\rangle$ `=>` $\langle\langle$ `lookup` x $d \curvearrowright s\ \rangle_k\ \langle\ d\ \rangle_{env}\rangle$	1'
`syntax` **Dico** ::= "`update`" `KItem KItem` **Dico** `rule update` x v ((y `|-->` w) , d) `=>` ((y `|-->` w) , (`update` x v d)) `requires` $x \neq y$ `rule update` x v ((y `|-->` $\_$) , d) `=>` ((y `|-->` v) , d) `requires` $x = y$	`rule` $\langle\langle\ x = i; \curvearrowright s\ \rangle_k\ \langle\ d\ \rangle_{env}\rangle$ `=>` $\langle\langle\ s\ \rangle_k\ \langle$ `update` x i $d\ \rangle_{env}\rangle$	10'
`syntax` **Bool** ::= `KItem` "`in_dico`" "(" **Dico**")" `rule` x `in_dico` ((y `|-->` $\_$) , d) `=>` x `in_dico`(d) `requires` $x \neq y$ `rule` x `in_dico` ((y `|-->` $\_$) , $\_$) `=>` $true$ `requires` $x = y$ `rule` $\_$ `in_dico` (`.Dico`) `=>` $false$	`rule` $\langle\langle$ `var` x, xl`;` $s\ \rangle_k\ \langle\ d\ \rangle_{env}\rangle$ `=>` $\langle\langle$ `var` xl`;` $s\ \rangle_k\ \langle$ ((x `|-->` 0) , d) $\rangle_{env}\rangle$ `requires` $\neg_{Bool}$ (x `in_dico`(d))	16'

Figure 13 – Nouvelle sémantique de notre langage IMP

Notre outil ne peut donc pas traiter actuellement les sémantiques faisant usage de la réécriture modulo ACUI. Nous avons également supposé que les variables du membre droit d'une règle de réécriture 𝕂 apparaissent nécessairement dans le membre gauche, ce qui est une restriction très commune en théorie de la réécriture. Cependant cette hypothèse est invalidée par la sémantique 𝕂 du langage Michelson [8]. La perspective la plus immédiate de ce premier travail consiste à étendre l'outil KaMeLo de manière à prendre en considération les attributs tels que `priority(nb)` et à mener une expérimentation sur une collection de sémantiques écrites avec 𝕂. Nous comptons également étendre notre transformation d'un CTRS vers un TRS afin de pouvoir garantir la correction et la complétude de celle-ci, en plus de la préservation de la terminaison et de la confluence. Un autre travail futur concerne la réécriture modulo ACUI que nous contournons actuellement. Une piste ici consiste à mettre en place un encodage *ad hoc* de manière à pouvoir utiliser les dictionnaires et les ensembles tels qu'ils sont implantés dans la bibliothèque de 𝕂.

Remerciements : Nous remercions vivement l'équipe 𝕂, notamment Xiaohong Chen, Everett Hildenbrandt, Zhengyao Lin et Dorel Lucanu, pour les réponses rapides à nos multiples questions.

11. Comme les chaînes de caractères ne sont pas natives dans Dedukti, nous considérons qu'une variable est de la forme `var(nb)`. L'égalité sur les identifiants des variables revient donc à une égalité sur les entiers.

Bibliographie

[1] GitHub avec la formalisation de 𝕂 en 𝕂. https://github.com/kframework/k-in-k.

[2] GitHub de Dedukti. https://github.com/Deducteam/Dedukti.

[3] GitHub de Dedukti v3. https://github.com/Deducteam/lambdapi.

[4] GitLab de KaMeLo. https://gitlab.com/semantiko/kamelo.

[5] Site web de l'école d'été ISR 2021. https://dalila.sip.ucm.es/isr2021/.

[6] Site web de 𝕂. https://kframework.org/.

[7] Site web de Sail. https://www.cl.cam.ac.uk/~pes20/sail/.

[8] Sémantique 𝕂 de Michelson. https://github.com/runtimeverification/michelson-semantics.

[9] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system. In *TYPES : Types for Proofs and Programs*, Novi SAd, Serbia, May 2016.

[10] F. Blanqui, G. Dowek, E. Grienenberger, G. Hondet, and F. Thiré. Some Axioms for Mathematics. In N. Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021)*, volume 195 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20 :1–20 :19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[11] D. Bogdănaş and G. Roşu. K-Java : A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.

[12] P. Borras, D. Clement, T. Despeyroux, J. Bertot, G. Kahn, B. Lang, and V. Pascual. Centaur : The system. 24 :14–24, 03 1989.

[13] X. Chen, Z. Lin, M.-T. Trinh, and G. 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.

[14] X. Chen, D. Lucanu, and G. Roşu. Matching Logic Explained. Technical Report http ://hdl.handle.net/2142/107794, University of Illinois at Urbana-Champaign and Alexandru Ioan Cuza University, July 2020.

[15] X. Chen and G. Roşu. Applicative Matching Logic : Semantics of K. Technical Report http ://hdl.handle.net/2142/104616, University of Illinois at Urbana-Champaign, July 2019.

[16] X. Chen and G. Rosu. Matching mu-Logic : Foundation of K Framework (Invited Paper). page 4 pages, 2019. Artwork Size : 4 pages Medium : application/pdf Publisher : Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany Version Number : 1.0.

[17] D. Cousineau and G. Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *TLCA*, 2007.

[18] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics*, pages 243–320, 1990.

[19] G. Dowek. Interacting Safely with an Unsafe Environment. *CoRR*, abs/2107.07662, 2021.

[20] C. Hathhorn, C. Ellison, and G. Roşu. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345. ACM, June 2015.

[21] L. Li and E. L. Gunter. IsaK-static : A complete static semantics of K. In *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Proceedings*, pages 196–215. Springer-Verlag Berlin Heidelberg, 2018.

[22] L. Li and E. L. Gunter. A Complete Semantics of 𝕂 and Its Translation to Isabelle. In A. Cerone and P. C. Ölveczky, editors, *Theoretical Aspects of Computing – ICTAC 2021*, pages 152–171, Cham, 2021. Springer International Publishing.

[23] D. Mulligan, S. Owens, K. Gray, T. Ridge, and P. Sewell. Lem : Reusable Engineering of Real-world Semantics. *ACM SIGPLAN Notices*, 49, 08 2014.

[24] D. Park, A. Ştefănescu, and G. Roşu. KJS : A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015.

[25] G. Rosu. K - A Semantic Framework for Programming Languages and Formal Analysis Tools. In D. Peled and A. Pretschner, editors, *Dependable Software Systems Engineering*, NATO Science for Peace and Security. IOS Press, 2017. From the lecture notes presented at Marktoberdorf'16.

[26] G. Roşu and T. F. Şerbănuţă. An overview of the K semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6) :397–434, Aug. 2010.

[27] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott : Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1) :71–122, 2010.

[28] A. Stefănescu, D. Park, S. Yuwen, Y. Li, and G. Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 74–91, Amsterdam Netherlands, Oct. 2016. ACM.

[29] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-environment : A Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction*, pages 365–370, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[30] P. Viry. Elimination of Conditions. *Journal of Symbolic Computation*, 28(3) :381–401, 1999.

Un Coq apprend à un bébé Colibri à flotter

Arthur Correnson[1] and François Bobot[2]

[1] ENS de Rennes, Rennes, Bretagne, France
arthur.correnson@ens-rennes.fr
[2] CEA, Gif-sur-Yvette, France
francois.bobot@cea.fr

Résumé

L'arithmétique flottante est connue pour être un sujet difficile. Ses propriétés contre-intuitives rendent l'écriture d'algorithmes manipulant les nombres flottants propice à de nombreuses erreurs. Des outils automatiques pour la vérification de programmes flottants existent mais ces outils faisant eux-mêmes usage de calculs en arithmétique flottante, on peut se poser la question de leur propre fiabilité. Dans cet article, nous proposons de vérifier formellement l'implémentation de raisonnements sur les nombres flottants dans le solveur de contraintes Colibri2. En particulier, nous présentons une méthodologie pour mener la preuve de correction de propagateurs de contraintes en utilisant l'assistant de preuve Coq. Nous discutons également de l'intégration des raisonnements prouvés à un développement logiciel complet en OCaml.

1 Introduction

1.1 Solveurs et résolution de contraintes

Les outils de raisonnement automatique sont de plus en plus utilisés dans le contexte de la vérification formelle des logiciels. Parmi ces outils, les solveurs dits Satisfiabilité Modulo Théorie (SMT) sont certainement les plus utilisés car ils permettent d'effectuer de manière automatisée des preuves mobilisant plusieurs théories typiques des problèmes de vérification (théorie des tableaux, théorie des flottants, ...). Le standard SMT-LIB [4] propose un language commun pour interroger les solveur SMT ainsi qu'un ensemble de théories usuelles dans lesquelles les solveurs SMT peuvent raisonner. Si le standard SMT-LIB définit un langage commun pour les solveurs, il laisse en revanche complètement libre le choix des méthodes de résolution. Le solveur Colibri2, nouveau successeur de COLIBRI, propose d'utiliser des méthodes de programmation par contrainte (CP) pour la résolution de problème SMT. Un tel solveur interprète les problèmes SMT comme des ensembles de contraintes reliant plusieurs variables. On cherche ensuite une affectation de chaque variable satisfaisant l'ensemble des contraintes. Les algorithmes utilisés pour la résolution reposent sur l'idée de tenir compte de chacune des contraintes du problème pour restreindre progressivement l'espace de recherche associé à chaque variable. On parle alors de propagation de contraintes. Informellement, un propagateur de contrainte est une fonction qui pour toute contrainte répond à la question suivante : « si la contrainte est satisfaite, que puis-je en déduire sur la valeur de ses variables ». La correction d'un solveur CP est donc directement conditionnée par la correction des propagateurs de contraintes.

Après une présentation rapide de quelques difficultés notables liées à l'écriture correcte de propagateurs flottants, nous proposons une méthodologie pour la preuve de tels propagateurs à l'aide de l'assistant de preuve Coq. Nous détaillerons également comment extraire le code Coq vers du code OCaml fiable et prêt à être intégré aux sources de Colibri2.

1.2 L'arithmétique flottante et ses défauts

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 approximation finie des nombres réels permettant d'effectuer des calculs efficaces. De manière simplifiée, un nombre flottant peut-être vu comme un réel que l'on peut écrire sous la forme $m2^e$ avec $(m, e) \in \mathbb{Z}$. On contraint ensuite la taille de m et e pour assurer que tous les flottants ont une représentation finie. En imposant une précision (p) et des bornes sur l'exposant ($e_{\min}$ et $e_{\max}$), on peut alors décrire un ensemble des nombres flottants comme

$$\mathbb{F}_{p,e_{\min},e_{\max}} = \{m2^e \mid |m| < 2^p \wedge e_{\min} \leq e \leq e_{\max}\}$$

Notons qu'en fonction du choix des constantes p, $e_{\min}$ et $e_{\max}$ il existe plusieurs ensembles de flottants. D'une manière générale, une configuration $(p, e_{\min}, e_{\max})$ est appelée un format flottant. Cette représentation est une vue très mathématique et ne convient pas à l'implémentation de calculs effectifs sur les nombres flottants. Le standard IEEE-754 définit une représentation binaire des nombres flottants sur 16, 32 et 64 bits (cela revient à fixer un format) et formalise le comportement attendu des nombres flottants en machine. En particulier, ce standard définit la notion d'opérateur d'arrondi : des opérateurs permettant le passage d'un réel arbitraire au flottant le plus proche dans un format donné. Plusieurs opérateurs d'arrondi sont définis par le standard, le plus commun étant l'arrondi "au plus proche" aussi appelé RNE (*round to nearest, ties to even*). Pour simplifier la présentation dans la suite, on notera toujours $o : \mathbb{R} \to \mathbb{F}$ l'opérateur d'arrondi au plus proche.

Le standard IEEE-754 définit précisément les différentes opérations arithmétiques sur les flottants ainsi que leur spécification. Dans la suite on notera $\oplus_f^m$ pour l"addition flottante dans un mode d'arrondi donné (ou simplement $\oplus_f$ si le mode d'arrondi n'a pas d'importance) et $\leq_f$, $<_f$, $=_f$ pour les relations binaires. Nous noterons les opérations réelles en suivant les notations conventionnelles des mathématiques. Une propriété fondamentale du standard IEEE-754 est qu'il impose que les opérations arithmétiques correspondent exactement à l'arrondi de leur pendant réel. Par exemple pour deux flottants x et y on a $x \oplus_f y = o(x + y)$.

L'introduction de la notion d'arrondi entraîne des comportements tout à fait singuliers de l'arithmétique flottante qui la rendent fondamentalement plus difficile que l'arithmétique réelle. En particulier, selon l'ordre dans lequel les opérations binaires sont faites, on accumule les erreurs d'arrondis et on perd de ce fait l'associativité de l'addition et de la multiplication par exemple.

En plus des propriétés déjà soulignées, l'arithmétique des nombres flottants présente une autre difficulté de taille : la présence des valeurs spéciales et des zéros signés. Dans les nombres flottants, on s'autorise à manipuler les infinis comme des valeurs usuelles ce qui permet en particulier de donner un sens à la division par zéro. On admettra donc que $\pm 1/0 = \pm\infty$ dans les nombres flottants. L'introduction des infinis comme valeurs oblige à étendre l'arithmétique usuelle et à donner un sens à des expressions telles que $+\infty \oplus_f -\infty$. On introduit donc une valeur spéciale notée `NaN` pour désigner le résultat de toute opération résultant en une valeur non déterminée. Enfin, la présence d'un bit de signe dans la représentation mémoire des nombres flottants tels que décrits dans le standard IEEE-754 différencie un zéro positif noté 0^+ et un zéro négatif noté 0^-. L'égalité sur les flottants confond les deux zéros, mais $1/0^+ =_f +\infty$ alors que $1/0^- =_f -\infty$. Ces singularités engendrées par l'introduction des valeurs spéciales doivent être l'objet d'une attention toute particulière lorsque l'on écrit des algorithmes sur les nombres flottants.

1.3 Raisonnements corrects sur les programmes flottants

Nous venons d'énumérer quelques-unes des nombreuses difficultés de l'arithmétique flottante. Ces dernières rendent l'écriture d'algorithmes flottants particulièrement propices à des erreurs de programmation pouvant avoir des conséquences désastreuses (l'accumulation d'erreurs d'arrondis dans un calcul numérique de trajectoire de missile par exemple). Des outils de vérification formelle existent et permettent de valider la correction de programmes manipulant les nombres flottants [6, 10, 7]. Cependant, ces outils faisant eux mêmes l'usage de l'arithmétique flottante, on peut alors se demander ce qu'il en est de leur propre correction. Si ces derniers produisent des résultats incorrects, la sûreté des programmes vérifiés par leur intermédiaire peut-être compromise. L'effet est d'autant plus pervers que la confiance que nous donnons aux outils de vérification est grande.

Des initiatives pour prouver la correction des outils de vérification eux-mêmes ont déjà été initiées [11, 12]. Suivant cette idée, le solveur Colibri2 (développé en langage OCaml [2]) propose d'intégrer au sein même de son processus de développement la preuve formelle des composants les plus sensibles. Nous proposons dans cet article d'ajouter au solveur Colibri2 un support pour la théorie des nombres flottants. Cette extension consiste essentiellement en l'ajout de nouveaux propagateurs de contraintes spécifiques aux nombres flottants. Nous utiliserons l'assistant de preuve Coq [1] pour programmer et prouver ces propagateurs avant de les extraire pour les intégrer au solveur.

2 Résolution de contraintes, domaines et propagateurs

L'objectif final des travaux présentés dans cet article est l'intégration de raisonnements sur les flottants prouvés corrects dans un solveur CP. Si on se concentre uniquement sur des problèmes simples de contraintes numériques (arithmétique entière, réelle ou flottante par exemple), les solveurs CP raisonnent sur un ensemble de contraintes liant n variables $x_1, \ldots, x_n$ au moyens des relations binaires et opérateurs arithmétiques usuels ($=, \leq, <, +, -$). En fonction de l'arithmétique considérée, les variables prennent leurs valeurs dans un ensemble $E \in \{\mathbb{Z}, \mathbb{R}, \mathbb{F}_{32}, \ldots\}$ et les contraintes peuvent être vues comme des fonctions $c : E^n \to \texttt{bool}$. Dans ce contexte, décider de la satisfiabilité d'une formule φ revient à chercher une affectation $\lambda \in E^n$ des variables telle que pour toute contrainte c dans φ, $c(\lambda) = \texttt{true}$. L'ensemble de toutes les affectations possibles E^n est appelé l'espace de recherche et son sous-ensemble $S_\varphi = \{\lambda \in E^n \mid \forall c \in \varphi, c(\lambda) = \texttt{true}\}$ est appelé espace des solutions. Si $S_\varphi = \emptyset$, il n'existe aucune solution et on dit que φ est insatisfiable, dans le cas inverse φ est satisfiable et toute affectation de S_φ est appelée modèle de φ.

Pour résoudre un ensemble de contraintes numériques φ, les solveurs analysent et combinent les contraintes pour collecter des informations sur les valeurs que peuvent prendre chaque variable et ainsi calculer une sur-approximation $\bar{S}_\varphi \supseteq S_\varphi$ de l'espace des solutions. Pour chaque contrainte c, on élimine dans $\bar{S}$ les affectations qui invalide trivialement la contrainte c. On dit que l'on propage la contrainte c. Initialement, on prend $\bar{S} = E^n$, puis on propage toutes les contraintes dans φ jusqu'à stabilisation de S_φ. Si au cours du processus, S_φ devient $\emptyset$ alors φ est insatisfiable. Si le processus converge vers une approximation $\bar{S}_\varphi \neq \emptyset$, on essaye d'extraire de $\bar{S}$ un modèle en choisissant une valeur pour chaque variable dans l'ordre (d'abord pour x_1 puis pour x_2, etc.). À mesure que des choix de la forme $x_i = v$ sont faits, on peut propager ces décisions comme des contraintes. Si une décision provoque une contradiction, on l'annule ainsi que toutes les propagations qu'elle a engendré pour en faire une nouvelle : on parle de *backtracking*. On alterne les phases de décision, propagation, *backtracking* jusqu'à avoir affecté

une valeur à toutes les variables ou bien avoir testé toutes les décisions possibles sans succès.

En pratique on représente $\bar{S}$ comme une table qui associe à chaque expression e apparaissant dans φ des informations caractérisant une sur-approximation de son domaine d'appartenance. Ces informations peuvent être très simples (par exemple un intervalle d'appartenance) ou plus abstraites (par exemple une information de signe, un majorant, un bit connu, etc.) Intuitivement, toute donnée décrivant un ensemble de valeurs peut être utilisée. Toutefois, on souhaite *a minima* pouvoir propager ces informations à l'aide de propagateurs de contraintes.

Les travaux de Marie Pelleau et ses collaborateurs [13] ont déjà mis en avant un lien fort entre cette notion de domaine dans le contexte des solveurs CP et la notion de domaine abstrait issue de la discipline de l'interprétation abstraite [9]. Ce rapprochement donne un cadre théorique utile pour exprimer le comportement attendu des propagateurs de contraintes et ainsi élaborer leur preuve de correction. Nous rappelons à présent quelques définitions inspirées par ces travaux et qui nous serviront par la suite de support pour la preuve de propagateurs flottants.

Définition 2.1 (Domaine abstrait sur un ensemble E)**.** On appelle domaine abstrait sur l'ensemble E la donnée d'un quadruplet $(D, \top, \bot, \gamma)$ où :

- D est un ensemble et $\gamma : D \to \mathcal{P}(E)$ est une interprétation des éléments de D comme des parties de E.
- $\bot \in D$ représente l'ensemble vide ($\gamma(\bot) = \emptyset$) et permet de caractériser les contradictions.
- $\top \in D$ représente E ($\gamma(\top) = E$) et permet d'initialiser l'espace de recherche.

Dans le contexte de la résolution de contraintes, on équipe également les domaines abstraits avec les opérateurs suivants :

- Un opérateur d'intersection $\sqcap : D \times D \to D$ qui permet de combiner deux informations connues sur une expression.
- Pour chaque opérateur arithmétique $\circ : E \times E \to E$, un opérateur abstrait $\circ^\sharp : D \times D \to D$ qui permet d'assigner un domaine aux expressions arithmétiques.
- Pour chaque relation binaire $\sim \subseteq E \times E$, un propagateur de contrainte $\rho^\sim : D \times D \to D \times D$. Il permet de raffiner les informations connues sur deux expressions e_1 et e_2 sous l'hypothèse que $e_1 \sim e_2$.

Pour garantir la correction des solveurs, on impose que ces nouveaux opérateurs aient certaines propriétés.

Hypothèse 2.1 (Correction de $\sqcap$)**.** Soient d_1 et d_2 deux éléments de D et x un élément de E, on impose que $\gamma(d_1) \cap \gamma(d_2) \subseteq \gamma(d_1 \sqcap d_2)$.

Hypothèse 2.2 (Correction des opérateurs)**.** Soit $\circ : E \times E \to E$ et soient d_1 et d_2 deux éléments de D, on impose que $\{x \circ y \mid x \in \gamma(d_1) \wedge y \in \gamma(d_2)\} \subseteq \gamma(d_1 \circ^\sharp d_2)$.

Hypothèse 2.3 (Correction des propagateurs)**.** Soit $\sim \subseteq E \times E$ une relation binaire, soient d_1 et d_2 deux éléments de D, soit $(d_1', d_2') = \rho^\sim(d_1, d_2)$ et soient x et y deux éléments de E, on impose deux propriétés :

- $\rho^\sim$ améliore d_1 et d_2 : $\gamma(d_1') \subseteq \gamma(d_1) \wedge \gamma(d_2') \subseteq \gamma(d_2)$
- $\rho^\sim$ respecte $\sim$: Si $x \sim y$ alors $x \in \gamma(d_1') \wedge y \in \gamma(d_2')$

Les propriétés de correction des propagateurs et des opérateurs abstraits assurent que leur utilisation permet toujours de calculer une sur-approximation de l'espace de recherche. Le fait que les propagateurs améliorent les domaines permet de garantir la convergence des solveurs : on ne fait jamais grossir l'espace de recherche en propageant une contrainte.

2.1 Domaine d'intervalles

Pour illustrer la démarche adoptée pour le développement et la preuve d'un domaine abstrait, nous prenons l'exemple des intervalles flottants. En présence de contraintes d'inégalités, ce domaine abstrait s'avère très utile car il permet d'identifier rapidement des bornes inférieures et supérieures pour chaque variable du problème à résoudre. Par exemple pour le problème $\varphi := \{x \leq 3 \wedge x \geq 4\}$, l'utilisation d'un domaine d'intervalles révèle immédiatement que φ est insatisfiable après deux propagations :

1. On note dx le domaine de x et on initialise $dx := \top = \{-\infty, +\infty\}$
2. On propage la contrainte $x \leq 3$: dx devient $\texttt{fst}(\rho^{\leq}(dx, \{3, 3\})) = \{-\infty, 3\}$
3. On propage la contrainte $x \geq 4$: dx devient $\texttt{fst}(\rho^{\geq}(dx, \{4, 4\})) = \bot$
4. $dx = \bot$, sans possibilité de *backtracking* la résolution prend fin et φ est insatisfiable.

Dans les réels, il est d'usage de définir les intervalles comme des pairs $(a, b) \in \mathbb{R}^2$. L'ensemble dénoté par l'intervalle (a, b) est ensuite défini comme $\{x \in \mathbb{R} \mid a \leq x \leq b\}$. Pour permettre un meilleur découpage de $\mathbb{R}$ en intervalles, on prolonge généralement cette définition en autorisant des bornes dans $\bar{\mathbb{R}} := \mathbb{R} \cup \{-\infty, +\infty\}$ et on distingue les bornes d'intervalles strictes et non strictes. Dans les flottants, les infinis sont des valeurs usuelles et peuvent donc être traitées indistinctement des autres. De plus, la finitude des nombres flottants permet de définir les opérateurs unaire *pred* et *succ* qui à tout flottant associe respectivement son prédécesseur et son successeur autorisant ainsi à ramener les tests d'inégalités strictes à des tests d'inégalités larges. Les notions de bornes strictes et non strictes peuvent donc être ignorées dans le cas flottant.

La présence du `NaN` perturbe la définition des intervalles flottants. Rappelons que la valeur `NaN` n'est pas comparable, c'est à dire que $\forall x \in \mathbb{F}, \neg(x \sim \texttt{NaN} \vee \texttt{NaN} \sim x)$ (avec $\sim \in \{\leq_f, \geq_f, <_f, >_f, =_f\}$). Tout intervalle flottant ayant une borne `NaN` représente donc un intervalle vide et par conséquent il n'existe pas d'intervalle contenant `NaN`. Néanmoins, dans le contexte d'un solveur, modéliser la possibilité qu'une expression puisse prendre la valeur `NaN` est utile. En effet prendre en compte la valeur `NaN` peut permettre de détecter des contradictions supplémentaires et donc d'accroître les capacités de résolution des solveurs. Par exemple, toute contrainte de la forme $x \sim \texttt{NaN}$ est insatisfiable et il est souhaitable qu'un domaine d'intervalle soit en mesure de capturer cette propriété. Pour prendre en compte `NaN`, nous étendons la définition des intervalles flottants comme $\mathbb{I}(\mathbb{F}) = \mathbb{F}^2 \times \texttt{bool}$. Un triplet $(a, b, \text{nan}) \in \mathbb{F}^2 \times \texttt{bool}$ dénote l'ensemble $\{x \in \mathbb{F} \mid a \leq_f x \leq_f b \vee (x = \texttt{NaN} \wedge \text{nan})\}$.

Nous définissons à présent les opérateurs $\sqcap$, $+^\sharp$ et $\rho^{\leq}$ associés au domaine abstrait des intervalles et tels que spécifiés précédemment.

Définition 2.2 (Concrétisation)**.** Les éléments de $\mathbb{I}(\mathbb{F})$ peuvent être interprétés comme des parties de $\mathbb{F}$ par la fonction de concrétisation γ définie comme :

$$\gamma((a, b, \text{nan})) := \{x \in \mathbb{F} \mid a \leq x \leq b \vee (x = \texttt{NaN} \wedge \text{nan})\}$$

Définition 2.3 (Intersection)**.** L'intersection peut se définir de la façon suivante :

$$(a, b, \text{nan}) \sqcap (a', b', \text{nan'}) := (\max(a, a'), \min(b, b'), \text{nan} \wedge \text{nan'})$$

Théorème 2.1 (Correction et précision de l'intersection)**.** Soient d_1, d_2 deux intervalles de $\mathbb{I}(\mathbb{F})$, alors :

— correction : $\forall x \in \mathbb{F}, x \in \gamma(d_1) \wedge x \in \gamma(d_2) \Rightarrow x \in \gamma(d_1 \sqcap d_2)$

— précision : $\forall x \in \gamma(d_1 \sqcap d_2), x \in \gamma(d_1) \wedge x \in \gamma(d_2)$

Démonstration. Par analyse de cas sur les valeurs nan et nan', et à l'aide des propriétés du max et du min on obtient la précision et la correction de $\sqcap$. □

Définition 2.4 (Opérateurs abstraits)**.** Les opérateurs abstraits $\circ^\sharp$ s'obtiennent à partir des opérateurs binaires $\circ$. Nous prenons l'exemple de l'addition :

$$(a,b,n) +^\sharp (a',b',n') := (a \oplus_f a', b \oplus_f b', n \vee n' \vee \bigvee_{\substack{x\in\{a,a'\}\\ y\in\{b,b'\}}} \text{sum-to-nan}(x,y))$$

avec sum-to-nan(x,y) si et seulement si $x \oplus_f y =$ `NaN` c'est à dire si x et y sont deux infinis opposés ou si l'une des deux valeurs est `NaN`.

Théorème 2.2 (Correction de l'addition abstraite)**.** Soient d_1, d_2 deux intervalles de $\mathbb{I}(\mathbb{F})$, alors $\forall (x,y) \in \mathbb{F}^2, x \in \gamma(d_1) \wedge y \in \gamma(d_2) \implies x \oplus_f y \in \gamma(d_1 +^\sharp d_2)$

Démonstration. Soient $d_1 = (a, b, \text{nan})$ et $d_2 = (a', b', \text{nan'})$, on raisonne par analyse de cas selon que les bornes a, a', b, b' et la somme $x \oplus_f y$ soient `NaN`, $\pm\infty$, ou des valeurs finies. Dans les cas où les trois sommes $a \oplus_f a'$, $x \oplus_f y$ et $b \oplus_f b'$ sont différentes de `NaN`, la monotonie de l'addition flottante s'applique (voir 3.2) et on peut déduire des hypothèses que $a \oplus_f a' \leq_f x \oplus_f y \leq_f b \oplus_f b'$ et donc que $x \oplus_f y \in \gamma(d_1 +^\sharp d_2)$. Dans les autres cas, les règles de l'arithmétique flottante pour les valeurs spéciales s'appliquent et la preuve s'achève par simple calcul. □

Les propagateurs de contraintes $\rho^\sim$ peuvent être définis en deux temps. Dans un premier temps nous définissons un opérateur $\kappa^\sim : D \to D$ qui calcule le domaine des candidats à la satisfaction de la relation $\sim$ par rapport à un domaine donné. Formellement on souhaite que $\kappa^\sim(d) \supseteq \{y \in \mathbb{F} \mid \exists x \in \gamma(d), y \sim x\}$. Par exemple pour les comparaisons flottantes $\leq_f$ et $\geq_f$ on a :

$$\kappa^\leq(a,b,\text{nan}) := \texttt{if } b <_f a \texttt{ then } (a,b,\texttt{false}) \texttt{ else } (-\infty,b,\texttt{false})$$

$$\kappa^\geq(a,b,\text{nan}) := \texttt{if } b <_f a \texttt{ then } (a,b,\texttt{false}) \texttt{ else } (a,+\infty,\texttt{false})$$

Définition 2.5 (Propagateurs)**.** Le propagateur $\rho^\sim$ se construit ensuite à l'aide de l'intersection.

$$\rho^\leq(d_1,d_2) := (\kappa^\leq(d_2) \sqcap d_1, \kappa^\geq(d_1) \sqcap d_2)$$

Théorème 2.3 (Correction du propagateur $\rho^\leq$)**.** Soient d_1 et d_2 deux intervalles flottants et soit $(d_1', d_2') = \rho^\leq(d_1,d_2)$, on a $\forall x \in \gamma(d_1), \forall y \in \gamma(d_2), x \leq_f y \implies x \in \gamma(d_1') \wedge y \in \gamma(d_2')$.

Démonstration. Par minimalité de $-\infty$ et maximalité de $+\infty$ pour les flottants et par correction de $\sqcap$. □

Nous montrons également que $\rho^\leq(d_1,d_2)$ ne dégrade pas la précision de d_1 et d_2.

Théorème 2.4 ($\rho^\leq$ améliore les domaines ou stagne)**.** Soient d_1 et d_2 deux intervalles flottants et $(d_1', d_2') = \rho^\leq(d_1,d_2)$, on a $\gamma(d_1') \subseteq \gamma(d_1) \wedge \gamma(d_2') \subseteq \gamma(d_2)$.

Démonstration. Ce résultat est un corollaire de la précision de $\sqcap$. □

Une implémentation Coq de ce domaine d'intervalles accompagnée des preuves de correction est disponible en ligne à l'adresse https://colibri.frama-c.com/html/doc/farith/doc/F.All.html. La méthodologie employée pour les preuves est présentée en détail dans la partie 3. D'autres développements proposent également de prouver formellement des domaines flottants en utilisant l'assistant de preuve Coq. C'est le cas notamment du projet Verasco [11] qui propose une implémentation en Coq de plusieurs domaines abstraits dont des domaines flottants. Les domaines flottants de Verasco sont développés dans le contexte de la vérification par interprétation abstraite et ne traitent que des flottants de format 32 ou 64 bits. L'implémentation que nous proposons est générique pour tout format flottant.

2.2 Correction complète d'un solveur CP

La correction des propagateurs ne suffit pas à assurer la correction complète d'un solveur CP. En effet, d'autres paramètres sont à considérer dont notamment la terminaison. De même, il faut s'assurer que l'algorithme qui orchestre la résolution fait une utilisation raisonnable des propagateurs. La démarche de prouver formellement l'algorithme de résolution de Colibri2 a déjà été amorcée mais nous ne détaillerons pas ici ces aspects. Notons toutefois que la preuve de correction et de terminaison d'un algorithme de résolution de contraintes requiert que les propagateurs utilisés soient corrects et, à ce titre, notre démarche constitue déjà une étape importante dans l'élaboration d'une telle preuve.

Prouver formellement un algorithme de résolution de contraintes dans un assistant à la démonstration ou tout autre outil de vérification formelle (comme Why3 par exemple [5]) représente un coût de développement conséquent. De ce fait, les fragments entièrement prouvés de Colibri2 sont modestes et fournissent un moteur de résolution correct mais peu puissant. En prouvant les propagateurs séparément, les mêmes propagateurs prouvés peuvent-être utilisés aussi bien dans un algorithme de résolution entièrement formalisé (auquel cas, les preuves de correction des propagateurs jouent un rôle dans la preuve de correction du solveur entier), que dans un algorithme de résolution plus avancé pour lequel on tolère l'absence d'une preuve de correction complète (dans ce cas, la simple correction des propagateurs donne déjà des garanties fortes).

3 Formalisation Coq des nombres flottants

3.1 La bibliothèque Flocq

Nous souhaitons intégrer au solveur Colibri2 un support minimal pour la théorie des nombres flottants. Nous proposons à cet effet une implémentation Coq du domaine d'intervalles flottants présenté dans la partie 2.1. Cette implémentation est ensuite traduite en code exécutable afin d'être intégrée dans les sources du solveur. Pour atteindre ces objectifs, nous avons besoin d'une modélisation des nombres flottants au sein de l'assistant de preuve Coq. Cette même modélisation sera utilisée dans les preuves mais également comme implémentation de référence de l'arithmétique flottante. Utiliser la même modélisation des flottants pour la preuve et pour le calcul permet d'assurer que les propriétés établies en Coq sont préservées après extraction (sous l'hypothèse que le mécanisme d'extraction vers OCaml de Coq est correcte).

Une modélisation des flottants. Le standard IEEE-754 donne une définition très complète des nombres flottants qui convient parfaitement comme guide pour l'implémentation d'une arithmétique flottante en machine. Néanmoins une représentation binaire est assez peu adaptée

pour formaliser la notion de nombre flottant dans un assistant à la démonstration. Pour permettre à la fois des raisonnements formels plus simples sur les nombres flottants et des calculs effectifs au sein d'un assistant à la démonstration, la bibliothèque Coq Flocq [8] propose un compromis et fournit une implémentation des flottants comme des nombres de la forme $m2^e$. Cette modélisation des nombres flottants est prouvée correcte vis à vis du standard IEEE-754 et peut être extraite vers un module OCaml fiable pour le calcul flottant. Ces propriétés font donc de Flocq un outil de choix pour la suite des travaux présentés dans cet article.

Opérateurs flottants. Flocq fournit un type flottant `binary_float` paramétré par une précision et un exposant maximal (l'exposant minimal est déduit à partir de l'exposant maximal et de la précision tel que décrit dans le standard IEEE-754). Le type `binary_float` est défini comme un type algébrique avec un constructeur pour chacune des valeurs spéciales (`NaN`, $0^{\pm}$, $\pm\infty$) et un constructeur pour tous les flottants finis de la forme $m2^e$. Cette modélisation permet de réduire les démonstrations en arithmétique flottante à une analyse de cas sur la forme des flottants considérés.

Les opérateurs usuels paramétrés par un mode d'arrondi sont également fournis. Par soucis de simplicité, nous omettrons les paramètres correspondant au format dans les échantillons de code Coq et nous noterons simplement `float` pour désigner le type des flottants dans un format fixé. En suivant cette convention, on considère les opérateurs suivants :

relations binaires	`Beqb, Bleb, Bltb : float -> float -> bool`
addition flottante	`Bplus : mode -> float -> float -> float`
constante `NaN`	`NaN : float`
constantes infinies	`B754_infinity : bool -> float`
projection dans les réels	`B2R : float -> R`
opérateur d'arrondi générique [1]	`round : mode -> R -> R`

3.2 Monotonie : un réel problème de sémantique

Établir la preuve formelle d'un théorème en Coq est souvent une tâche longue et difficile, en particulier lorsque les preuves impliquent de mobiliser des résultats d'arithmétique et *a fortiori* d'arithmétique flottante. La preuve de correction d'opérateurs sur les domaine abstraits est très consommatrice de ce type de résultats et en particulier de théorèmes de monotonie qui assurent la préservation de la relation `Bleb` à travers les opérations arithmétiques. Ces théorèmes sont triviaux dans $\mathbb{R}$ mais plus difficiles à formuler voire même faux dans les flottants. La monotonie de l'addition réelle par exemple assure que $\forall (x, y, z) \in \mathbb{R}^3, x \leq y \rightarrow x + z \leq y + z$, mais la valeur `NaN` ne pouvant être comparée, il s'ensuit que ce résultat est faux dans les flottants. En effet, pour $x = -\infty$, $y = +\infty$, $z = +\infty$, on a bien $x \leq_f y$ mais $x \oplus_f z =$ `NaN` et donc $\neg(x \oplus_f z \leq_f y \oplus_f z)$. Pour des raisons similaires, la plupart des résultats d'arithmétique flottante sont conditionnés par des hypothèses spécifiques sur les opérandes. Ainsi la monotonie de l'addition dans $\mathbb{F}$ se ré-exprime de la façon suivante :

Théorème 3.1 (Monotonie de l'addition flottante)**.** Soit m un mode d'arrondi et x, y, x', y' quatre flottants. Si $x \oplus_f^m x' \neq$ `NaN`, $y \oplus_f^m y' \neq$ `NaN`, $x \leq_f y$ et $x' \leq_f y'$ alors $x \oplus_f^m x' \leq_f y \oplus_f^m y'$.

1. On pourrait s'attendre à ce qu'un opérateur d'arrondi soit de signature $\mathbb{R} \rightarrow \mathbb{F}$, mais une telle définition ne serait pas calculable car les réels ne sont pas représentables en machine. En revanche, une axiomatisation des réels suffit pour l'écriture de spécifications.

Pour établir la preuve de ce théorème en Coq on introduit deux résultats intermédiaires plus simples `Bplus_le_compat_l` et `Bplus_le_compat_r`. La monotonie de l'addition découle directement de ces deux résultats.

```
Theorem Bplus_le_compat_l :
  forall (m : mode) (x y z : float),
    is_nan (Bplus m z x) = false ->
    is_nan (Bplus m z y) = false ->
    x <= y ->
    Bplus m z x <= Bplus m z y
```

```
Theorem Bplus_le_compat_r :
  forall (m : mode) (x y z : float),
    is_nan (Bplus m x z) = false ->
    is_nan (Bplus m y z) = false ->
    x <= y ->
    Bplus m x z <= Bplus m y z
```

FIGURE 1 – Monotonie de l'addition flottante

Pour établir les preuves de ce type de résultats en arithmétique flottante, une première approche pourrait être de dérouler les définitions des opérations flottantes impliquées et de raisonner directement au niveau de la représentation des flottants. Raisonner à si bas niveau est généralement assez difficile. Une autre approche consiste à raisonner à un niveau d'abstraction plus élevé en plongeant les flottants dans les réels. Cette méthode présente de nombreux avantages. D'une part nous avons l'habitude de manipuler l'arithmétique réelle et les raisonnements en sont d'autant plus faciles à mener. D'autre part, Coq propose des outils pour automatiser les preuves en arithmétique réelle linéaire et non-linéaire. La bibliothèque Flocq fourni une fonction `B2R : float -> R` qui donne l'interprétation réelle des flottants ainsi que de multiples théorèmes définissant la sémantique des opérateurs flottants dans $\mathbb{R}$. Une fois que les propriétés que nous souhaitons démontrer sont établies dans $\mathbb{R}$, il faut pouvoir garantir que les résultats sont transposables dans les flottants. Pour se faire, on introduit des lemmes de préservation garantissant que certaines propriétés réelles sont toujours vraies dans les flottants.

Un premier résultat qui permet le passage de flottant à réel est la correction de l'addition flottante. Ce théorème donne la sémantique de l'addition flottante dans $\mathbb{R}$ et énonce que la somme de deux *petits* flottants finis est égale à l'arrondi de leur somme réelle. Dans le cas où la somme réelle provoque un débordement de capacité même une fois arrondi en un flottant de précision arbitraire, l'addition flottante renvoie une valeur d'*overflow* qui peut varier selon le mode d'arrondi et le signe des opérandes. Les dépassements de capacité peuvent produire la valeur $\pm\infty$ et donc n'ont pas de représentation réelle ; ce qui oblige à formuler le théorème de correction en prenant en compte deux cas selon que la somme déborde ou non. En supposant donnée une fonction `overflow : R -> bool` qui teste les dépassements de capacité pour un format flottant particulier et une fonction `binary_overflow : mode -> bool -> float` produisant la valeur de débordement en fonction d'un signe et d'un mode d'arrondi, le théorème de correction de l'addition flottante peut s'exprimer de la façon suivante :

```
Theorem Bplus_correct :
  forall (m : mode) (x y : float), is_finite x -> is_finite y ->
    if overflow (round m (B2R x + B2R y)) then
      Bplus m x y = binary_overflow m (Bsign x)
    else B2R (Bplus m x y) = round m (B2R x + B2R y)
```

FIGURE 2 – Sémantique réelle de l'addition flottante dans un mode d'arrondi donné

À chaque fois que nous souhaitons utiliser le théorème de correction de l'addition sans

hypothèses sur les opérandes flottants, nous sommes contraints de raisonner par analyse de cas et d'envisager la possibilité que les opérandes soient infinies ou que la somme flottante déborde. Cela rend les preuves longues et fastidieuses. Pour éviter d'avoir à traiter séparément tous ces cas, nous proposons de donner une nouvelle interprétation des flottants dans $\bar{\mathbb{R}} = \mathbb{R} \cup \{+\infty, -\infty\}$.

```
Inductive Rx : Type := Inf : bool -> Rx | Real : R -> Rx.

Definition B2Rx (f : float) : Rx :=
  match x with
  | B754_infinity b => Inf b
  | _ => Real (B2R f)
  end.
```

FIGURE 3 – Plongement des flottants dans $\bar{\mathbb{R}}$

Nous étendons le type `R` de Coq par un type `Rx` qui comprend les constantes $\pm\infty$. La fonction `B2Rx : float -> Rx` donne la sémantique des flottants dans $\bar{\mathbb{R}}$. Pour spécifier les opérateurs flottants dans $\bar{\mathbb{R}}$, nous redéfinissons également un arrondi `roundx` dans $\bar{\mathbb{R}}$. Contrairement à l'arrondi générique de Flocq, `roundx` donne soit une constante infinie (si le réel arrondi est trop grand) soit un réel de la forme $m2^e$ respectant strictement le format flottant imposé ($m < 2^{prec}$ et $e_{\min} \leq e \leq e_{\max}$).

Nous dotons également `Rx` des opérations usuelles $+, -, *, \ldots$ et des relations $\leq, \geq, <, >$ définies comme des extensions naturelles de leurs équivalents réels. Notons qu'il reste des zones d'ombres sur la valeur à attribuer à des expressions telles que `(B2Rx NaN)` ou `(Inf true + Inf false)`. Nous choisissons d'attribuer des valeurs arbitraires et nous montrons formellement qu'en dépit de ces choix, la nouvelle sémantique des flottants dans `Rx` se comporte comme la sémantique habituelle dans `R`. En particulier, nous montrons la monotonie de l'arrondi qui est une propriété fondamentale des nombres flottants.

```
Lemma roundx_mono :
  forall (m : mode) (r1 r2 : Rx), r1 <= r2 -> roundx m r1 <= roundx m r2.
```

FIGURE 4 – Monotonie de l'arrondi dans $\bar{\mathbb{R}}$

Ce passage de `R` à `Rx` nous permet de donner une expression plus générale du théorème de correction de l'addition (5).

```
Theorem Bplus_correct' :
  forall (m : mode) (x y : float),
    is_nan (Bplus m x y) = false ->
    B2Rx (Bplus m x y) = roundx m (B2Rx x + B2Rx y).
```

FIGURE 5 – Sémantique de l'addition flottante dans $\bar{\mathbb{R}}$

L'énoncé du théorème de correction de l'addition flottante dans $\bar{\mathbb{R}}$ est plus générique et l'hypothèse `is_nan (Bplus m x y) = false` est strictement plus faible que de supposer la finitude des opérandes. Dans sa version initiale, si l'on souhaite utiliser le théorème de correction

```
Lemma le_B2Rx :
  forall (x y : float), x <= y -> B2Rx x <= B2Rx y.

Lemma B2Rx_le :
  forall (x y : float), is_nan x = false -> is_nan y = false ->
  B2Rx x <= B2Rx y -> x <= y.
```

FIGURE 6 – Lemmes de préservation de l'ordre par passage de $\bar{\mathbb{R}}$ à $\mathbb{F}$

de l'addition sans avoir aucune hypothèse sur x et y, cela oblige à considérer 8 cas selon que x et y soient finis ou non et selon que leur somme déborde ou non. Le nouveau théorème que nous proposons limite cette analyse à deux cas seulement : la somme flottante vaut `NaN` ou non. Les preuves sont de ce fait plus courtes, plus simples à comprendre et plus faciles à développer. Pour illustrer cela, prenons l'exemple de la preuve du théorème `Bplus_le_compat_r`. Comme décrit précédemment, on commence par introduire des lemmes de préservation (voir figure 6) puis nous établissons la preuve du théorème `Bplus_le_compat_r`.

Démonstration. Soient x, y, z trois flottants et m un mode d'arrondi quelconque. On suppose que $x \oplus_f^m z$ et $y \oplus_f^m z$ sont différents de `NaN` et que $x \leq_f y$. Par le théorème `le_B2Rx`, comme $x \leq_f y$ on a également $\texttt{B2Rx}(x) \leq_f \texttt{B2Rx}(y)$. Montrons à présent que $x \oplus_f^m z \leq_f y \oplus_f^m z$. Par `B2Rx_le` cela équivaut à montrer que $\texttt{B2Rx}(x \oplus_f^m z)$ est plus petit que $\texttt{B2Rx}(y \oplus_f^m z)$.
En application du théorème de correction de l'addition flottante `Bplus_correct'`, cela revient à montrer $\texttt{roundx}_m(\texttt{B2Rx}(x) + \texttt{B2Rx}(z)) \leq_f \texttt{roundx}_m(\texttt{B2Rx}(y) + \texttt{B2Rx}(z))$. Par monotonie de l'arrondi (lemme `roundx_mono`), et monotonie de l'addition réelle cette propriété s'établit trivialement sous l'hypothèse que $\texttt{B2Rx}(x) \leq_f \texttt{B2Rx}(y)$. □

En pratique, cette stratégie de preuve s'est généralisée à tous les résultats d'arithmétique flottante dont nous avons eu besoin d'établir la preuve. Rappelons que l'objectif final de ces travaux est de montrer la correction de propagateurs flottants. Par essence, les propagateurs ont pour but, étant donnée une hypothèse P faite sur deux nombres flottants, d'en déduire une conséquence Q plus précise. Leur preuve de correction se réduit donc à prouver des théorèmes de la forme $P \implies Q$ avec P, Q deux assertions sur les flottants. La méthode de preuve suivante s'applique alors de manière assez systématique.

1. On souhaite prouver des énoncés de la forme $P \implies Q$ où P et Q sont des assertions sur les nombres flottants.
2. On suppose P et on exploite la sémantique réelle des flottants pour en déduire P', une réécriture de P dans $\bar{\mathbb{R}}$ obtenue en remplaçant tous les opérateurs et relations flottants par leurs équivalents dans $\bar{\mathbb{R}}$.
3. En utilisant des résultats simples d'arithmétique réelle on prouve $P' \implies Q'$ avec Q' une réécriture dans $\bar{\mathbb{R}}$ de Q.
4. On développe la preuve du lemme de préservation $Q' \implies Q$ qui assure que la propriété réelle Q' est préservée dans les flottants.
5. On a P, $P \implies P'$, $P' \implies Q'$ et $Q' \implies Q$ donc par transitivité $P \implies Q$ ce qui achève la preuve.

Cette méthode nous a permis de formuler un nombre important de résultats utiles sur les nombres flottants dont notamment des résultats de monotonie rendant possible la preuve

formelle de correction des propagateurs pour le domaine abstrait des intervalles. L'opérateur abstrait $+^\sharp$, l'intersection $\sqcap$ ainsi que les propagateurs pour les contraintes $\rho^=$, $\rho^\leq$, $\rho^\geq$, $\rho^<$ et $\rho^>$ ont été prouvés grâce à cette méthode. Ces propagateurs couvrent exactement le fragment linéaire de l'arithmétique flottante ce qui permet déjà de résoudre les problèmes SMT les plus simples.

4 Extraction vers OCaml : les flottants découvrent le monde réel

4.1 Colibri2

Colibri2 est un solveur de contraintes écrit en OCaml. En son cœur il propose un graphe d'égalité avec domaines. Une telle structure permet de garder à jour des informations sur les valeurs et les domaines associés à chaque sous-terme d'un problème tout en tenant compte des égalités potentielles entre termes. Un nœud du graphe (Node.t) représente une classe d'équivalence entre des termes liés par une contrainte d'égalité. Les domaines permettent d'attacher une information à toute une classe d'équivalence. Les valeurs (Value.t) permettent de représenter les constantes. Pour obtenir un modèle, le solveur cherche à associer à chaque nœud une valeur avec la règle qu'une unique valeur peut être associée à une classe d'équivalence.

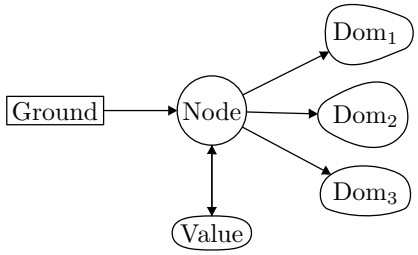


FIGURE 7 – Vision simplifiée des différents objets composant le Egraph dans Colibri2

L'architecture de Colibri2 est modulaire et permet d'enregistrer des nouveaux domaines abstraits. Tout module présentant au moins un type `t` et une fonction d'intersection peut être intégré comme un nouveau module de domaines. Les modules de domaines doivent également exposer une fonction `is_singleton` qui permet de détecter lorsqu'un domaine ne contient qu'une unique valeur et retourne cette valeur le cas échéant. Additionnellement, les modules de domaines peuvent fournir des propagateurs de contraintes.

Cette architecture permet d'implémenter les domaines abstraits séparément du cœur du solveur. En particulier, nous pouvons implémenter un domaine abstrait en Coq, l'extraire en OCaml et l'intégrer directement aux sources de Colibri2. Le domaine des intervalles flottants présenté dans cet article ainsi qu'une extraction vers OCaml de la bibliothèque Flocq sont ainsi intégrés à Colibri2 sous la forme d'un module nommé Farith2.

4.2 Le(s) problème(s) de l'extraction

Pour intégrer la théorie des nombres flottants au solveur Colibri2, deux éléments principaux sont requis. D'une part un module de valeurs qui fournit un type pour les nombres flottants ainsi que ses opérations arithmétiques élémentaires. Celui-ci permettra au Egraph d'associer des

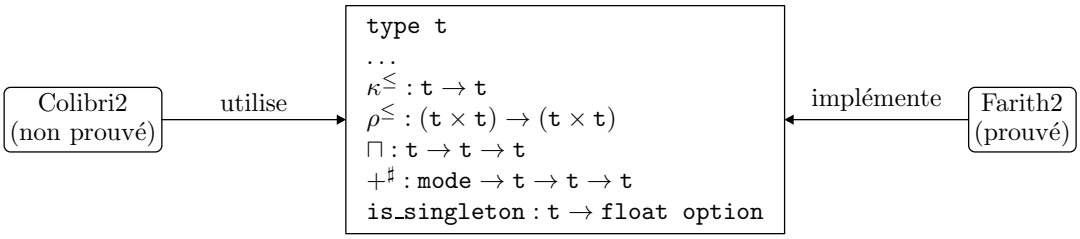


FIGURE 8 – Schéma de collaboration entre Colibri2 et Farith2

valeurs (Value.t) représentant un nombre flottant à un nœud. D'autre part, nous souhaitons enregistrer des nouveaux domaines flottants et leurs propagateurs associés.

Comme décrit plus tôt (voir section 3), la modélisation Coq que nous donnons des nombres flottants expose principalement un type `binary_float` paramétré par deux entiers relatifs (dénotant une précision et un exposant maximale). De plus, les opérateurs sur le type `binary_float` imposent comme préconditions que la précision soit strictement positive et que l'exposant maximal soit supérieure à la précision.

L'extraction naïve du type `binary_float` vers OCaml donne un type simple et incorrect : la partie dépendante du type est effacée et les flottants des différents formats sont confondus. De même, l'extraction d'un opérateur tel que `Bplus` ignore les préconditions. Ces conditions sont nécessaires pour établir la correction des opérateurs et le code extrait n'est donc pas nécessairement correct puisqu'il ne garantit pas leur satisfaction.

4.3 Effacement des types dépendants par instanciation de modules

Une solution partielle à ce problème est de proposer une ou plusieurs instanciations du type `binary_float` et des opérateurs associés. Ces instanciations sont réalisées en Coq de manière à garantir leur utilisation correcte. En particulier on pourra prouver pour chaque opérateur que les paramètres de précision et d'exposant choisis pour l'instanciation vérifient les pré-conditions `0 < prec` et `prec < emax`. En suivant cette démarche, nous pouvons par exemple exposer un module spécifique pour l'arithmétique flottante IEEE-754 sur 32 bits (le type `float32` des flottants 32 bits se réécrit `binary_float 24 128` dans le formalisme de la bibliothèque Flocq).

Un tel module ne présente plus que des types simples directement traduisible en OCaml sans perte d'information. Notons toutefois que les valeurs de type `binary_float prec emax` sont susceptibles de contenir une preuve d'appartenance au bon format flottant. Ces informations sont également effacées par l'extraction. Pour éviter de produire des valeurs flottantes incorrectes, on rend donc privé le type `t`. Les fonctions produisant des valeurs de type `t` ayant été écrites en Coq, on a l'assurance que toutes les valeurs produites par leur intermédiaire sont bien formées. Des constructeurs intelligents de la forme `A -> t` peuvent également être ajoutés afin de permettre la création de nouvelles valeurs de type `t`. La même démarche s'applique aux modules de domaines flottants.

4.4 Un Type flottant pour les gouverner tous

L'effacement des types dépendants par instanciation systématique n'est qu'une solution partielle au problème de l'extraction de code correcte. En effet, l'utilisation de cette méthode force à passer de familles potentiellement infinies de types (`binary_float` par exemple) à un ensemble restreint de types simples (typiquement `float32` et `float64`). De plus, pour chaque

```
Record float : Type := {
  prec := Z;
  emax := Z;
  Hprec := (0 < prec)%Z;
  Hemax := (prec < emax)%Z;
  value := binary_float prec emax;
}
```

FIGURE 9 – Type des flottants de format arbitraire

instance souhaitée, du code Coq doit être écrit rendant difficile l'ajout de nouvelles instances *a posteriori*. Dans le cadre du développement d'un solveur cela est d'autant plus handicapant que la théorie des flottants telle que décrite dans le standard SMT-LIB [3] manipule des flottants de format arbitraire. Pour respecter au mieux le standard et proposer un solveur le plus générique possible, il est donc souhaitable de pouvoir calculer sur des flottants de n'importe format. Nous proposons pour cela une approche expérimentale en Coq permettant de modéliser tous les formats flottants comme un unique type simple.

On commence par définir un type `float` des flottants de format arbitraire. Ce type est un enregistrement à trois champs : une précision `prec`, un exposant `emax` et un nombre flottant de type `binary_float prec emax`. On demande également que les invariants `0 < f.prec` et `f.prec < f.emax` soient vérifiés pour tout enregistrement `f : float`.

Pour définir les opérations binaires sur ce nouveau type flottant, il est nécessaire de vérifier à priori que les deux opérandes flottants partagent le même format. Cette vérification peut-être faite par le calcul. Bien sûr, si les formats flottants des deux opérandes ne coïncident pas, l'opération ne peut aboutir. Dans un langage de programmation comme OCaml, on souhaiterai utiliser une fonction telle que `assert` pour assurer que les deux formats correspondent et lever une exception sinon. À défaut de pouvoir lever une exception en Coq, nous proposons d'introduire une fonction `Assert` définie comme suit :

```
Program Definition Assert {t : Type} (condition : bool)
  (f : condition = true -> t) (default : t) : t :=
  match condition with
  | true => f _
  | false => default
  end.
```

Pour un booléen `b`, une valeur par défaut `d` et une valeur de retour `r`, le programme `assert b x (fun _ => r) d` renvoit `r` si la condition est vraie ou `d` sinon. Notons que le troisième paramètre de la fonction `assert` est une fonction qui prend en paramètre une preuve que la condition est vraie. Cela permet de se servir de cet hypothèse dans la définition de la valeur `r`.

Cette fonction peut-être extraite vers du code OCaml efficace qui effectue une vérification dynamique de la condition et lève une exception si celle-ci n'est pas satisfaite. On ajoute pour cela une directive d'extraction.

```
Extract Inlined Constant assert => "(fun x f -> assert x; f ())".
```

Cette directive d'extraction ne compromet pas la correction dans le cas où l'assertion est satisfaite. En effet dans ce cas la fonction OCaml a la même sémantique que la définition Coq.

Dans le cas contraire, le mécanisme d'exception d'OCaml interrompt l'exécution du programme. Cet argument de correction suppose que le code extrait ne contient pas de `try_with` permettant de rattraper ces exceptions et que les fonctions Coq extraites ne prennent pas en paramètre des fonctions d'ordre supérieur en argument (celles-ci pourraient éventuellement rattraper une exception provoquée par un `assert false`).

L'usage de la fonction `Assert` rend possible la définition des opérations sur le type `float`. Pour deux arguments flottants donnés, on vérifie la compatibilité de leurs formats avec un simple test d'égalité. Cette condition est vérifiée par un `Assert` et dans le cas où la condition est satisfaite, puisque les types des deux flottants sont égaux, on peut utiliser l'addition flottante. Cette démarche nécessite l'introduction d'opérateurs de conversion de type qui servent simplement à guider le vérificateur de type de Coq. En particulier, étant donnée une preuve que deux formats flottants (p, e) et (p', e') sont égaux et un flottant f de format (p', e'), la fonction `same_format_cast` retourne ce même flottant mais explicitement typé comme étant de format (p, e). Pour définir cette fonction on utilise l'outil `Program` de Coq qui permet de simplifier les éventuelles preuves requises pour établir le bon typage d'un terme.

```
Definition same_format (x y : float) : bool :=
  (prec x =? prec y)%Z && (emax x =? emax y)%Z.

Program Definition same_format_cast {p e p' e' : Z}
  (H : ((p =? p') && (e =? e') = true)%Z)
  (f : binary_float p' e') : binary_float p e := f.
```

FIGURE 10 – Opérateurs de conversion de type

```
Definition add (m : mode) (x y : float) : float :=
    Assert (same_format x y) (fun H => {|
      prec := prec x;
      emax := emax x;
      Hprec := Hprec x;
      Hemax := Hemax x;
      value := @Bplus _ _ (Hprec x) (Hemax x) m (value x)
        (same_format_cast H (value y));
    |}) NaN.
```

FIGURE 11 – Addition sur les flottants de format arbtitraire

5 Conclusion

Grâce à l'utilisation de l'assistant de preuve Coq, nous proposons une implémentation formellement prouvée d'une bibliothèque de manipulation d'intervalles flottants accompagnés de propagateurs de contraintes simples. Cette bibliothèque a été conçue dès son développement pour pouvoir être extraite en OCaml. A l'aide de cette bibliothèque, nous proposons une extension au solveur Colibri2 permettant un support minimal et prouvé correct pour la théorie des nombres flottants telle que définie dans le standard SMT-LIB. Par exemple, le problème SMT suivant est maintenant résolu par Colibri2 (qui déclare le problème insatisfiable) :

```
(set-logic ALL)
(declare-const x Float32)
(assert (fp.leq x ((_ to_fp 8 24) RNE 1.0)))
(assert (fp.geq x ((_ to_fp 8 24) RNE 3.0)))
(check-sat)
```

Il reste des propagateurs à implémenter et à prouver. Par exemple nous pourrions ajouter des opérateurs arithmétiques inverses qui, étant donné le domaine d'appartenance d'une expression arithmétique, permettent d'inférer les domaines d'appartenance des variables apparaissant dans cette expression. La preuve de tels propageteurs sera simplifiée par le choix des définitions et des stratégies de preuves que nous avons présentées.

D'autres domaines comme par exemple des domaines de congruences ou des domaines de bits connus ont prouvé leur efficacité pour la résolution de problèmes de contraintes sur les nombres flottants. Un support pour de nouveaux domaines pourrait également être ajoutés dans la continuation de ces travaux.

Références

[1] The coq proof assistant. https://coq.inria.fr/.

[2] The ocaml programming language. https://ocaml.org/.

[3] The smt-lib floatingpoint theory. http://smtlib.cs.uiowa.edu/theories-FloatingPoint.shtml.

[4] The smt-lib standard. http://smtlib.cs.uiowa.edu/.

[5] Why3. http://why3.lri.fr/.

[6] Ali Ayad and Claude Marché. Multi-prover verification of floating-point programs. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning*, pages 127–141, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[7] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining coq and gappa for certifying floating-point programs. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *Intelligent Computer Mathematics*, pages 59–74, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[8] 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.

[9] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[10] Alwyn E. Goodloe, César Muñoz, Florent Kirchner, and Loïc Correnson. Verification of numerical programs : From real numbers to floating point numbers. In Guillaume Brat, Neha Rungta, and Arnaud Venet, editors, *NASA Formal Methods*, pages 441–446, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[11] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified c static analyzer. *SIGPLAN Not.*, 50(1) :247–259, January 2015.

[12] Stéphane Lescuyer and Sylvain Conchon. A Reflexive Formalization of a SAT Solver in Coq. In *TPHOLs 2008 : In Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[13] Marie Pelleau, Antoine Miné, Charlotte Truchet, and Frédéric Benhamou. A Constraint Solver based on Abstract Domains. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, edi-

tors, *VMCAI 2013 - 14th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 7737 of *Lecture Notes in Computer Science*, pages 434–454, Rome, Italy, January 2013. Springer-Verlag.

Hydras & Co.: Formalized mathematics in Coq for inspiration and entertainment

Pierre Castéran[1], Jérémy Damour[2], Karl Palmskog[3], Clément Pit-Claudel[4], and Théo Zimmermann[5]

[1] Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, F-33400 Talence, France
[2] Univ. de Paris, F-75013 Paris, France
[3] KTH Royal Institute of Technology, Stockholm, Sweden
[4] MIT CSAIL, Cambridge, Massachusetts, USA
[5] Inria, Univ. de Paris, CNRS, IRIF, UMR 8243, F-75013 Paris, France

Abstract

Hydras & Co. is a collaborative library of discrete mathematics for the Coq proof assistant, developed as part of the Coq-community organization on GitHub. The Coq code is accompanied by an electronic book, generated with the help of the Alectryon literate proving tool. We present the evolution of the mathematical contents of the library since former presentations at JFLA meetings. Then, we describe how the structure of the project is determined by two requirements which must be continuously satisfied. First, the Coq code needs to be compatible with its ever-evolving dependencies (the Coq proof assistant and several Coq packages both from inside and outside Coq-community) and reverse dependencies (Coq-community projects that depend on it). Second, the book needs to be consistent with the Coq code, which undergoes frequent changes to improve structure and include new material. We believe Hydras & Co. demonstrates that books on formalized mathematics are not limited to providing exposition of theories and reasoning techniques—they can also provide inspiration and entertainment that transcend educational goals.

1 Introduction

1.1 Background

Libraries of formalized mathematics based on proof assistants, such as Coq, Lean, and Isabelle/HOL, are continually growing in size and scope. For example, Mathlib for Lean and the core set of projects in the Mathematical Components family for Coq both amount to hundreds of thousands of lines of code (LOC) and tens of thousands of definitions and theorems [36, 35]. However, after a key mathematical definition or result is added to a library, it must be *documented and maintained* [51, 41].

Maintenance includes not only adaptation to changes in new proof assistant versions, but also reorganization to accommodate new contributions. Documentation is usually two-fold: *source code comments* that describe specific definitions and results in-line and *books* that carefully introduce the library and its idioms [32, 2]. The former tends to be terse and dense and serves experienced users, while the latter is more long-winded and exhaustive and serves beginners. As a library changes and expands, all its documentation needs to be made consistent and complete. Authors use many techniques, including literate programming [30] and custom tools, to pretty-print and check source code snippets and generate proof assistant output.

While books that document proof assistant libraries are valuable to beginners, the purpose of recently published books is mainly instrumental, i.e., to teach a certain topic or technique. We believe that books and libraries can instead become *ends in themselves*—not just sources of exposition and learning, but also sources of inspiration and entertainment.

1.2 Vision

The Hydras & Co. project, part of Coq-community on GitHub [14], aims to be an experimental platform for the collaborative development of documented libraries of formal proofs. Coq-community is a community organization that the first and last authors founded in 2018 with two goals in mind: providing a solution for the long-term maintenance of interesting open source Coq packages, and working collaboratively on documentation projects. The Hydras & Co. project demonstrates that these two goals are not independent: interesting Coq packages can become the basis for new documentation. This umbrella project now includes evolved versions of the former Cantor and Additions libraries [12, 9] (under the new names of Hydra-battles and Addition-chains), the Ackermann sub-library extracted from Russel O'Connor's Goedel library [38, 37], and a preliminary bridge to the Gaia library by José Grimm [25, 22]. By following this approach of commenting interesting Coq packages, we provide new documentation content that contributes to the diversity of the thriving Coq ecosystem.

We call on the Coq users in the JFLA community and beyond to come and join us in this effort, by bringing new interesting projects which are worth presenting to Coq learners, *a.k.a.* Coq users, and guiding them in their exploration. We also always have project ideas to extend further our explorations and anyone is welcome to join the team by sending small or larger contributions through pull requests. The current state of the project is already the result of such evolutions after several of us contributed project solutions and new proposals to the initial version of the first author.

Futhermore, contrary to traditionally published books, the "book" that forms part of this project is intended to be forever evolving. As new Coq formalization patterns and proof techniques appear, the book can be adapted to demonstrate their use (in case they fit well with our applications). By using modern maintenance techniques such as continuous integration and deployment, we can ensure that this documentation stays up to date with the latest Coq releases. With Alectryon [41, 40], we ensure that code and documentation are always in sync.

1.3 Hydra games

We chose to build our library and book on two simple themes which allow many variations: computing powers in a monoid, and Kirby and Paris' hydra battles. In the interest of space, we will only present the second theme in this paper.

Hydra games (also known as *Hydra battles*) appear in an article published in 1982 by two mathematicians, Laurie Kirby and Jeff Paris: *Accessible Independence Results for Peano Arithmetic* [29]. This article describes a game between two players: Hercules and a hydra. A short description of the game can be found in [4, 29, 11]. One can also play with Andrej Bauer's simulator [3]. In a few words:

- A hydra is a finite tree, traditionally presented with the root at the bottom, the leaves of which are called *heads* (Figure 1).
- At every round, Hercules chops off one head of the hydra. If the head is at a distance greater than 1 from the root, then some sub-tree h of the hydra is copied a certain amount n of times. The number n of copies and the sub-tree h may depend on the considered variant of the game and the time elapsed since the beginning of the fight. Figure 1 shows an example with $n = 2$.

Kirby and Paris proved the following theorems, applying combinatorial results about ordinal numbers by Jussi Ketonen and Robert Solovay [28].

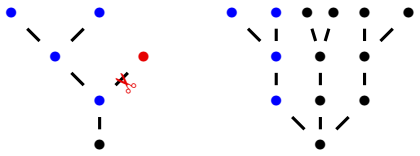


Figure 1: Two successive states of a hydra in a battle. Hercules chopped off the rightmost head of the hydra (red), and the whole left tree except the root node (blue) was copied twice.

Theorem 1. *In the Hydra game, Hercules eventually wins, whichever the strategy of both players: choice of a head to chop off, choice of the number of copies.*

Theorem 2. *Theorem 1 cannot be proved in Peano Arithmetic.*

The contrast between the simplicity of the statements above and the complexity of their proofs convinced us that it is a good theme for a commented library [26] of formal proofs written for the Coq proof assistant [19].

Complex formalizations and proofs are explained in the book. Whenever various reasonable choices exist, we try to present and compare the alternatives. For instance, Figures 7 and 8 show two radically different proofs of the equality $\omega + 42 + \omega^2 = \omega^2$. The first one is a simple proof by computation, the second one shows how this equality is a consequence of the axioms of the set-theoretic model by Kurt Schütte [43].

This work is also an opportunity to provide concrete examples of formalization and proof techniques: operational type classes, functions defined by equations, dependently typed functions, etc. It may be also used as a library on ordinal numbers, for instance for proving termination properties. Prior stages of this project have already been presented at JFLA [10, 11]. In this paper, we present recent evolutions of the library: new results, interaction with the Coq-community project [14], and documentation generated with Alectryon [41, 40].

2 Recent developments

The 2018 article [11] presented a formal proof of the following theorem:

Theorem 3. *Let μ be any ordinal strictly less than ϵ_0. There is no function mapping hydras to the segment $[0, \mu)$ that could be used as a measure in proving termination for* any *hydra battle.*

Our proof was based on the construction of a battle between Hercules and the hydra where the number of copies at each round was given by the elimination of an existential quantifier. So, it was mandatory to consider the class of *all* hydra battles, or, equivalently, the class of battles where the hydra choses arbitrarily its number n of copies at every round of the battle.

Unfortunately, the examples most commonly shown in the literature (see for instance [29, 4, 3]) assume that the hydra grows k copies at step k of the game, which is incompatible with our proof. We prove now that Theorem 3 still holds with these typical battles. Since we are looking for a minoration of the length of such battles, we can work with the following hypotheses and invariants, without any loss of generality.

- The game starts at an initial step $k = i$, where i is any natural number.
- Hercules always chops off the rightmost head of the hydra.

- The hydra is always the tree representation of some ordinal strictly below ϵ_0 in Cantor normal form (thus, the rightmost branch is also one of the shortest). For instance, Figure 1 shows the hydras respectively associated with ω^{ω^2+1} and $\omega^{\omega^2} \times 3$.

In mathematical terms, if at step $k \geq i$, the hydra is associated with the ordinal α, at step $k+1$ it is associated with $\{\alpha\}(k+1)$, *i.e.* the $(k+1)$th element of the canonical sequence of α [28]. In the following, the expression "the hydra α" is an abbreviation of "the hydra associated with the ordinal α".

Our new proof of Theorem 3 is based on a systematic study of strictly decreasing sequences of ordinals below ϵ_0, borrowed from Ketonen and Solovay [28], and the formalization of which is described in detail in chapters 5 and 6 of [13].

Besides Theorem 3, we study also the number of steps of a battle: Let $\alpha < \epsilon_0$ be an ordinal. We prove that the number of steps of the battle starting with α at step i is greater or equal than $H'_\alpha(i) - i$, where H' is a slight variant of the Hardy hierarchy of rapidly growing functions [53, 28, 42, 52]. The function H'_α is defined by transfinite recursion over α on Figures 2 and 3.

$$H'_0(i) = i \tag{1}$$
$$H'_\alpha(i) = H'_{(\{\alpha\}(i+1))}(i) \quad \textit{if } \alpha \textit{ is a limit ordinal} \tag{2}$$
$$H'_\alpha(i) = H'_\beta(i+1) \quad \textit{if } \alpha = \beta + 1 \tag{3}$$

Figure 2: The H' rapidly growing hierarchy of arithmetical functions

```
Equations H'_ (alpha: E0) (i:nat) :  nat  by wf  alpha Lt :=
  H'_ alpha  i with E0_eq_dec alpha Zero :=
    { | left _zero =>  i ;
      | right _nonzero
          with Utils.dec (Limitb alpha) :=
          { | left _limit =>  H'_ (Canon alpha (S i))  i ;
            | right _successor =>  H'_ (Pred alpha) (S i)}}.
```

Figure 3: H' definition using the Equations plug-in.

Using H's equations as rewrite rules, we can study a concrete example. We take the hydra of figure 4 and $i = 0$ as initial configuration. By a sequence of rewritings and inductions, we prove that the number of steps of the considered battle is greater or equal than 2^{2^N}, where $N = 2^{70} - 1$. By comparison with the diagonalized Ackermann function $\lambda i.\, A(i,i)$, we prove also that, for $\alpha \geq \omega^\omega$, the function computing the length of the battle starting with the hydra α and the initial step i is not primitive recursive.

Proof engineering improvements. The original release of the Cantor library preceded support for type classes in Coq [48]. Changes over time before the move of Hydras & Co. to Coq-community have since then introduced type classes for some concepts in the Hydra-battles library related to ordinals, e.g., to provide a uniform way to compare ordinals within different representation systems. Most type classes initially used a *semi bundled* approach [35], but we

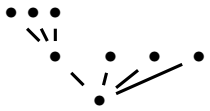


Figure 4: The hydra associated with the ordinal $\omega^3 + 3$

have recently *unbundled* classes to enable more sharing during resolution. For instance, we use the (single-field) `Decision` type class [49], that describes deciding a proposition, in parameters to other classes rather than in their fields. Due to the modest depth of our class hierarchy, we believe unbundling will not lead to scalability issues. Orthogonally to type classes, we introduced automation for definitions using the Paramcoq [27] and Equations [47] plug-ins, as illustrated in Figure 3.

3 Integration with Coq-community

3.1 Background

Coq-community is an informal organization run by volunteer Coq users on GitHub that aims to maintain interesting open source Coq projects for the long term and facilitate collaboration among Coq users on documentation, tooling, etc. Coq-community was created in 2018, inspired by the Elm Community organization [54]. Such "Community Package Maintenance Organizations" exist in many software ecosystems, as they avoid the common problem of an important package becoming unmaintained after its author has moved on to other projects or has disappeared [55].

In the case of Coq, this problem is likely even more prevalent than in other ecosystems, as many packages are created by graduate students or researchers for a specific paper and not planned to be maintained for the long term by authors. However, authors are generally open to having someone else who expresses interest in their work take over package maintenance.

Coq-community makes it easy to change maintainers by defining a process for transferring or forking an unmaintained package, tooling for setting up good maintenance practices (such as continuous integration), and by making it possible for someone to take over a package without making a long-term commitment (as maintainers who drop out can easily be replaced by some other volunteers).

As of 2021, Coq-community hosts over 50 projects maintained by over 30 volunteers. The hosted projects come from a variety of origins. Some had been maintained in the past by the Coq development team on behalf of the authors, but this meant that only minimal changes required to make the project build with new Coq versions were performed. Some were still maintained by their original authors, but were transferred to enable other users to help out with maintenance and facilitate adoption of best practices on, e.g., continuous integration. Others were simply unmaintained and were revived after their transfer to Coq-community.

Given the objectives of Coq-community, we believe it makes sense to propose a transfer anytime we encounter an interesting Coq project that is insufficiently maintained. After a transfer, the Coq-community maintainers are explicitly allowed to perform large changes and refactorings. This means, for instance, that we can consolidate packages by merging them, or split up a single package into several packages.

3.2 Integration of primitive recursive functions

In order to prove formally that the length of the kind of hydra battles we consider is not given by any primitive recursive function, we chose to use a formalization of primitive recursive functions that was originally part of Russell O'Connor's formal proof of Gödel's first incompleteness theorem [38, 37]. For this purpose, and above all in consideration of the scientific interest of this contribution, we decided to host and maintain O'Connor's work in Coq-community.

Since computability is a key topic in computer science teaching and O'Connor's library is a nice illustration of dependently typed programming, we decided to devote a full chapter (chapter 9 of [13]) to the formalization of primitive recursive functions, with comments on the definitions and proofs and (counter-)examples and exercises. As part of the writing process, we made the formalization into a new sub-library of the Hydra-battles library, dubbed Ackermann.

3.3 Towards a bridge to Gaia

The Gaia project by José Grimm aimed to formalize mathematics in Coq in the style of Nicolas Bourbaki. The formalization of the first book in the Elements of Mathematics series by Bourbaki, on the theory of sets, was initially described in a technical report in July 2009 [20]. The set-theoretic axioms and basic definitions in Gaia were derived from an earlier development by Carlos Simpson [46, 45]. Grimm then wrote (and continually updated) technical reports describing the formalization of Bourbaki's two subsequent books [21, 24] and additional topics in number theory [22, 23], before he passed away in 2019.

In 2020, members of Coq-community transferred the Gaia source code to GitHub and adapted it for recent releases of the Mathematical Components library, which Gaia heavily relies on. Anonymous volunteers ("collaborators of Nicolas Bourbaki") then finished the only in-progress proof left by Grimm. At around 155,000 LOC, Gaia is currently one of the largest maintained open source Coq projects [25].

Gaia contains definitions of ordinals in Cantor and Veblen normal form [22], adapted from the historical Cantor contribution [12]. The data types for ordinals are essentially defined the same way as in Hydras & Co., but they are not identical inside Coq, e.g., due to residing in different modules. There are also minor differences in how ordinal arithmetic is implemented, due to the different evolutionary paths taken since divergence from the ancestor library.

As an initial step in bridging Gaia and Hydras & Co., we were able to establish a correspondence between both implementations of ordinals through rewriting lemmas. For instance, we proved that multiplication of ordinals in Cantor normal form in Hydras & Co. refine Gaia's corresponding multiplication operation, and then imported Gaia's proof of associativity of the multiplication almost for free.

The initial draft bridge code mostly uses the SSReflect proof language and idioms from the Mathematical Components library. We made this design decision since we believe it is less challenging to reason about Hydra-battles code using SSReflect and MathComp than to reason about Gaia without SSReflect.

3.4 Package Genealogy, Dependencies, and Organization

Both due to its prior stages [10, 11] and the recent integrations described just above, the current Hydras & Co. Coq code has a complex inheritance. Figure 5 illustrates the relationships between Hydras & Co. packages and show their ancestry from historical Coq contributions. To indicate the scope and relative sizes of packages, Table 1 breaks down lines of code for each package in

the Hydras & Co. galaxy, as reported by the `coqwc` tool, and lists their repository names in Coq-community and identifiers in the Coq opam package index [18].

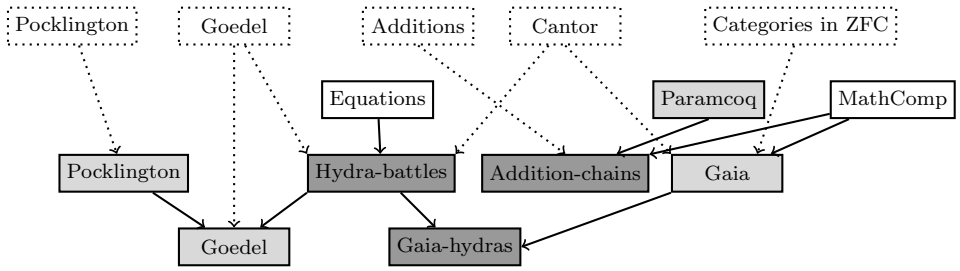


Figure 5: Genealogy and dependencies for Hydras & Co. packages. Dotted boxes represent historical Coq contributions, while regular boxes represent maintained Coq packages. Dark gray packages are maintained in the Hydras & Co. GitHub repository, while light gray packages are maintained in other Coq-community repositories. Dotted lines represent Coq code ancestry, while regular lines represent direct code dependencies.

Before the Ackermann sub-library was moved from the Goedel package repository to the Hydras & Co. repository, the Goedel package had around 7,000 specification LOC and 38,000 proof LOC. Figure 5 and Table 1 thus reveal that the seemingly large Coq formalization of Gödel's incompleteness theorem can be viewed as a modestly-sized proof on top of general libraries for, on the one hand, first-order logic, primitive recursive functions, and Peano Arithmetic (Ackermann), and on the other hand, prime numbers and their properties (Pocklington).

Package name	Repository	Index identifier	Version	Spec LOC	Proof LOC
Pocklington	`pocklington`	`coq-pocklington`	`8.12.0`	825	3,798
Goedel	`goedel`	`coq-goedel`	`8.13.0`	2,799	10,762
Gaia	`gaia`	`coq-gaia`	`1.12`	28,850	124,839
Hydra-battles	`hydra-battles`	`coq-hydra-battles`	`0.5`	14,368	48,385
Addition-chains	`hydra-battles`	`coq-addition-chains`	`0.5`	1,961	2,210
Gaia-hydras	`hydra-battles`	`coq-gaia-hydras`	`0.5`	81	177

Table 1: Current numbers of lines of code for packages in the Hydras & Co. galaxy.

In traditional software development, it is common for packages to continually depend on a large number of packages. In contrast, projects in the Coq ecosystem are often subject to "dependency aversion", where maintainers eschew depending on useful packages because it may lead to work in adapting to upstream changes. With Hydras & Co., we hope to demonstrate that projects with complex dependencies are feasible to manage using recent advances in build management and infrastructure.

4 Modernizing the build infrastructure

4.1 Documentation with Alectryon

The Hydras & Co. book is written in LaTeX, but it makes very frequent references (about once per page, 274 snippets over 281 pages) to parts of the Coq development, showing definitions (Fig. 3), computation results (Fig. 6), lemmas (Fig. 7) and parts of proof scripts (Fig. 8). The order in which these references appear in the book is independent of the structure of our libraries,

so we chose to maintain the book as a standalone document, separate from the Coq source code. This contrasts with the Coqdoc approach, where explanatory prose is embedded within Coq source files (a detailed discussion of different approaches to documenting Coq developments can be found in [41]).

Originally, we copied snippets from Coq sources into LaTeX manually, and recorded and inserted the corresponding outputs manually as well. This approach is common, but brittle: changes to Coq definitions or lemmas had to be reflected in the book's sources, and we found multiple instances where the book and the Coq development had diverged.

We solved this maintenance issue by moving to Alectryon, a tool that automatically records Coq proofs [40]. Instead of copy-pasting fragments into LaTeX, we now embed small LaTeX files automatically generated by Alectryon from our Coq development. Our build system guarantees that these LaTeX snippets are always up-to-date and consistent with the code (and, by comparing these snippets across releases of Coq or versions of Hydras & Co., we can easily spot unexpected changes).

Importing snippets into a LaTeX document was not one of the original use cases of Alectryon. Firstly, the original Alectryon did not have support for exporting to LaTeX. Secondly, it was geared towards documenting individual source files, where code and prose are interleaved within the same file. In this style, the code is documented in the same order as it is compiled. We extended Alectryon to support our needs by implementing a LaTeX backend, and by programming it to generate individual snippet files, one per `snippet` comment block. The later part was straightforward: all it took was to build a custom Alectryon *driver*, a small (about 100 lines) Python program that leverages most of the Alectryon toolchain but defines a custom frontend that understands our snippet annotations (and otherwise exposes the exact same command line as Alectryon).

Once the infrastructure was in place, the transition happened gradually, over a few weeks: for each snippet of Coq code that was in the book, we had to take the following steps:

1. Mark the snippet in the Coq sources (we use special comments `(* begin snippet` *name* `*) ... (* end snippet` *name* `*)`)
2. Configure output display, using special Alectryon annotations to configure what should be shown (only the inputs, inputs and outputs, some steps of the proof but not all, some proof states at key moments in a proof, etc.)
3. Replace the copy-pasted inputs and output in LaTeX with an `input` command.

```
Eval cbv  zeta beta delta [pow_17]  in  pow_17.
= fun x : A =>
  x * x * (x * x) * (x * x * (x * x)) *
  (x * x * (x * x) * (x * x * (x * x))) * x
: A -> A
```

Figure 6: Automatically capturing the output of computations

Applications in other proof assistants. The Alectryon toolchain is intended to eventually support the languages of many proof assistants, and already has preliminary support for Lean 3. Our Alectryon extensions for snippet generation are in principle applicable to any language

```
Example Ex42: omega + 42 + omega^2 = omega^2.
Proof.
  rewrite <-  Comparable.compare_eq_iff.
  ──────────────────────────────────────
  compare (omega + 42 + phi0 2) (phi0 2) = Eq
  reflexivity.
Qed.
```

Figure 7: A simple proof by computation

supported by Alectryon. However, snippet delimiters are language specific, and our tooling currently assumes Coq-compatible delimiters; we expect to lift this restriction in the near future. We view Hydras & Co. as an incubator for experimental Alectryon features that, when deemed stable and useful, can be disseminated to a wider audience including users of other proof assistants.

4.2 Technologies supporting the monorepo structure

A monorepo (shorthand for monolithic repository) is a version control repository containing multiple independent or related packages. Monorepos have gained increasing popularity following experiences in large companies such as Google, but are also used at a smaller scale for managing open source projects. They are known to simplify the management of dependencies, making cross-packages changes, and refactorings [8].

One of the few early uses of monorepos in the Coq ecosystem was for the Mathematical Components library [32], whose maintainers developed a custom build infrastructure to check multiple packages on each commit. In light of that recent tooling improvements have made monorepo use easier, we chose to use an explicit monorepo structure for Hydras & Co. in Coq-community. Other Coq-community projects have since adopted a similar structure.

In the context of Hydras & Co., we rely on the following tools.

The Dune build system [50]. Dune was originally designed to build OCaml projects, but was recently extended to support Coq. Dune allows building packages contained in a single source tree separately, which is essential to be able to publish the different libraries contained in a single repository as separate (opam) packages to the Coq package index.

However, Dune is currently inconvenient for local IDE-based Coq code development and does not yet support building Coqdoc files. Therefore, we still support building the whole project using `make` via the `coq_makefile` tool bundled with Coq, and we rely on this build system in our documentation generation pipeline. We expect to migrate fully to Dune as soon as these limitations are lifted.

Docker-Coq-Action [34]. This GitHub Action provides a very simple way of setting up Continuous Integration (CI) for a Coq project with an opam file. At the current time, we rely on it, together with the mathcomp Docker images [33], to test our two main libraries, Hydra-battles and Addition-chains, with multiple versions of Coq.

The Coq Nix Toolbox [17]. This toolbox, based on the Nix package manager, provides an alternative way of setting up CI for a Coq project (also relying on GitHub Actions). We use

Let us prove again the equality $\omega+42+\omega^2=\omega^2$. We recall that ω^2 is an abbreviation of $\phi_0(2)$, *i.e* the third additive principal ordinal, and that `F` is a notation for the coercion which maps natural numbers to ordinals.

```
Example Ex42: omega + 42 + omega^2 = omega^2.
```

Our proof is very different from the computational proof of Figure 7. By definition of additive principal ordinals, it suffices to prove the inequality $\omega + 42 < \phi_0(2)$.

```
  assert (HAP:= AP_phi0 2).
  elim  HAP; intros _ H0; apply H0; clear H0.
```

```
HAP: In AP (phi0 (F 2))
──────────────────────────
omega + F 42 < phi0 (F 2)
```

Since the set AP of additive principals is closed under addition (by Lemma *AP_plus_closed*), it suffices to prove the inequalities $\omega < \omega^2$ and $42 < \omega^2$.

```
  Check AP_plus_closed.
```

```
AP_plus_closed
     : forall alpha beta gamma : Ord,
       In AP alpha ->
       beta < alpha ->
       gamma < alpha -> beta + gamma < alpha
```

```
  assert (Hlt: omega < omega^2) by
      (rewrite omega_eqn; apply phi0_mono, finite_mono;
        auto with arith).
```

```
HAP: In AP (phi0 (F 2))
Hlt: omega < phi0 (F 2)
──────────────────────────
omega + F 42 < phi0 (F 2)
```

```
  apply AP_plus_closed; trivial.
```

```
──────────────────────────
F 42 < phi0 (F 2)
```

```
  (* ... *)
```

Figure 8: A proof interleaved with text (from the book)

this Nix-based CI for several things.

1. To test the build of the project (as a single unit) with `coq_makefile` (whereas the Docker-based CI relies on the opam packages, which use Dune to build the libraries contained in the project).

2. To generate the documentation of the project (book in PDF format and Coqdoc HTML pages) by relying on the output of the `coq_makefile` build.

Title	Year	Category
Coq'Art [6]	2004	Traditional
Software Foundations [39]	2007	Executable
Certified Programming with Dependent Types [15]	2011	Executable
Program Logics for Certified Compilers [1]	2014	Traditional
Programs and Proofs [44]	2014	Executable
Formal Reasoning About Programs [16]	2017	Traditional
Computer Arithmetic and Formal Proofs [7]	2017	Traditional
Mathematical Components [32]	2018	Traditional

Table 2: Coq book categorization.

3. To test the bridge to the Gaia library, because the Nix-based CI supports out-of-the-box caching of build dependencies. Given that Gaia takes more than 5 minutes to build, this build is only done once, then reused at each new run of the CI.

4. To test compatibility with the Goedel library, which was made to depend on the Hydra-battles library since the Ackermann sub-library was moved from the former to the latter. For this, we rely on the fact that the Coq Nix Toolbox has native support for generating a CI configuration that includes reverse dependency compatibility testing.

The artifacts of the Nix builds are stored in the Coq-community binary cache on Cachix [31]. This means that a given version never has to be built twice and Continuous Integration can be almost instantaneous when no change has been made (e.g., after merging a pull request).

By relying on two different technologies for CI, we are able to fit a larger range of use cases, while also providing more assurance that the project does build correctly in a variety of configurations. Given that the two technologies are well maintained (within Coq-community), relying on both does not incur a significant cost, compared to the benefits they provide.

5 Comparison of Hydras & Co. with other Coq books

We believe that Coq books can broadly be divided into two categories:

- *Executable textbooks* that are in effect large, well-commented Coq programs from which other representations of the material (HTML, PDF) are derived using tools.

- *Traditional textbooks* generated from documentation languages such as LaTeX that are accompanied by standalone Coq formalizations and/or code snippets.

Table 2 shows a category breakdown for what we believe are the most prominent Coq books in English. Note that several traditional books, such as Coq'Art, used custom tools during their writing process to keep code snippets synchronized with Coq. However, once a book is published, the accompanying code generally takes on a life of its own [5] and may come to substantially diverge from the book.

We believe the Hydras & Co. electronic book has found an attractive set of tradeoffs between the properties of executable books and traditional books. Specifically, in an executable book, the proof assistant language places limits on the structure. For example, Coq may not permit repeating a definition from another file verbatim, since the definition uses variables that are not present in the current context. Thanks to our Alectryon based toolchain, the Hydras & Co. book can include "live" code fragments at any point in the text. And in contrast to traditional books, we validate the consistency of code fragments during every build of the electronic book.

On the one hand, we follow executable books in providing continuous delivery of new revisions at a high pace. But on the other hand, we also aim to make regular timestamped versions compatible with specific versions of Coq, akin to new editions of traditional books.[1]

The flip side of our approach is that we separate the Coq sources from the LaTeX sources that describe the code, and our use of snippet delimiters in the Coq source files may lower source readability. For books and libraries with small scope and specialized audiences, distributing knowledge representations between sources and documentation formats the way we do in Hydras & Co. may introduce overhead without providing complementary benefits. More specifically, we currently distribute the book as a PDF generated by LaTeX that links to corresponding HTML documentation of the Coq sources generated by Coqdoc. In contrast, executable books such as Software Foundations can provide a single uniform HTML representation of both book contents and code. We hope that future toolchain improvements can address all these issues.

6 Conclusion and perspectives

Hydras & Co. wants to provide a connection between scientific literature (e.g., [29, 28, 43]) and proof assistant technology. For the mathematician, it can give a concrete view of the mathematical content, not only through full proofs, but also through illuminating computations: examples, functions associated with constructive proofs, etc. For the Coq learner, it provides a consistent set of examples, allowing to present and compare various formalization and proving techniques. It is also a medium sized library (more than 50,000 LOC), dependent on various tools and libraries of the Coq ecosystem, which may be also used to experiment with new maintenance techniques of the code and its documentation. To facilitate easy dissemination and reuse, all code and documentation is available under the permissive MIT license.

We plan to extend Hydras & Co. in two main directions. Firstly, we plan to bridge the combinatorial results of Hydras & Co. and the set-theoretic content of Gaia, enabling the transfer of many interesting theorems between the two packages. Secondly, we aim to write a formal proof in Coq of the original statement of Theorem 2, using O'Connor's formalization of Peano Arithmetic [37]. Moreover, Hydras & Co. is not limited to the study of ordinal numbers and their applications. We are also developing a package about efficient exponentiation algorithms, and aim to eventually include new topics. We invite new collaborators to join us in our efforts.

Acknowledgments

We thank the anonymous reviewers for their comments, and we are grateful to the original authors and current maintainers of the Coq packages we use and depend on: José Grimm (Gaia), Russell O'Connor (Goedel), Matthieu Sozeau (Equations), the Mathematical Components team, Marc Lasson and Chantal Keller (Paramcoq), and the authors and maintainers of Coq and its associated tools.

References

[1] Andrew W. Appel. *Program Logics for Certified Compilers*. Cambridge University Press, Cambridge, United Kingdom, 2014. https://www.cs.princeton.edu/~appel/papers/plcc.pdf.

[1] The latest Hydras & Co. release is 0.5, available at https://github.com/coq-community/hydra-battles/releases/tag/v0.5.

[2] Jeremy Avigad, Leonardo de Moura, Soonho Kong, and Sebastian Ullrich. Theorem proving in Lean 4, 2021. https://leanprover.github.io/theorem_proving_in_lean4/.

[3] Andrej Bauer. The hydra game. http://math.andrej.com/2008/02/02/the-hydra-game.

[4] Andrej Bauer. The hydra game source code. https://github.com/andrejbauer/hydra, 2008.

[5] Yves Bertot and Pierre Castéran. Coq'Art examples and exercises. https://github.com/coq-community/coq-art.

[6] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, Heidelberg, 2004. https://www.labri.fr/perso/casteran/CoqArt/.

[7] Sylvie Boldo and Guillaume Melquiond. *Computer Arithmetic and Formal Proofs*. ISTE Press - Elsevier, 2017.

[8] Gleison Brito, Ricardo Terra, and Marco Tulio Valente. Monorepos: a multivocal literature review. *CoRR*, 2018. https://arxiv.org/abs/1810.09477.

[9] Pierre Castéran. Additions. User Contributions to the Coq Proof Assistant, 2004. https://github.com/coq-contribs/additions.

[10] Pierre Castéran. Utilisation en Coq de l'opérateur de description. In *Actes des Journées Francophones des Langages Applicatifs*, pages 30–44, 2007. http://jfla.inria.fr/2007/actes/PDF/03_casteran.pdf.

[11] Pierre Castéran. Hydra ludica, une preuve d'impossibilité de prouver simplement. In Sylvie Boldo and Nicolas Magaud, editors, *Journées Francophones des Langages Applicatifs*, pages 91–104, 2018. https://hal.inria.fr/hal-01707376/document.

[12] Pierre Castéran and Évelyne Contejean. On ordinal notations. User Contributions to the Coq Proof Assistant, 2006. https://github.com/coq-contribs/cantor.

[13] Pierre Castéran. Hydras & Co. https://github.com/coq-community/hydra-battles/releases/tag/v0.5, 2021.

[14] The Coq community project. https://github.com/coq-community/.

[15] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. http://adam.chlipala.net/cpdt/.

[16] Adam Chlipala. Formal reasoning about programs, 2017. http://adam.chlipala.net/frap/.

[17] Cyril Cohen and Théo Zimmermann. A Nix toolbox for reproducible Coq environments, Continuous Integration and artifact reuse. The Coq Workshop, July 2021.

[18] Coq Development Team. Coq opam package index. https://coq.inria.fr/opam/www/.

[19] Coq Development Team. The Coq Proof Assistant. https://coq.inria.fr.

[20] José Grimm. Implementation of Bourbaki's Elements of Mathematics in Coq: Part one, theory of sets. Research Report RR-6999, INRIA, 2009. https://hal.inria.fr/inria-00408143.

[21] José Grimm. Implementation of Bourbaki's Elements of Mathematics in Coq: Part two; ordered sets, cardinals, integers. Research Report RR-7150, INRIA, 2009. https://hal.inria.fr/inria-00440786.

[22] José Grimm. Implementation of three types of ordinals in Coq. Research Report RR-8407, INRIA, 2013. https://hal.inria.fr/hal-00911710.

[23] José Grimm. Fibonacci numbers and the Stern-Brocot tree in Coq. Research Report RR-8654, INRIA, 2014. https://hal.inria.fr/hal-01093589.

[24] José Grimm. Implementation of Bourbaki's Elements of Mathematics in Coq: Part three structures. Research Report RR-8997, INRIA, 2016. https://hal.inria.fr/hal-01412037.

[25] José Grimm, Alban Quadrat, and Carlos Simpson. Gaia. https://github.com/coq-community/gaia. A Coq-community project.

[26] Hydra battles. https://github.com/coq-community/hydra-battles. A Coq-community project.

[27] Chantal Keller and Marc Lasson. Parametricity in an Impredicative Sort. In *Computer Science*

Logic, volume 16, pages 381–395. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2012. http://drops.dagstuhl.de/opus/volltexte/2012/3685.

[28] Jussi Ketonen and Robert Solovay. Rapidly growing Ramsey functions. *Annals of Mathematics*, 113(2):267–314, 1981. http://www.jstor.org/stable/2006985.

[29] Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14(4):285–293, 1982. https://faculty.baruch.cuny.edu/lkirby/accessible_independence_results.pdf.

[30] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 01 1984.

[31] Domen Kožar. Cachix, 2018–2021. https://cachix.org.

[32] Assia Mahboubi and Enrico Tassi. Mathematical Components. https://doi.org/10.5281/zenodo.3999478, 2018. With contributions by Yves Bertot and Georges Gonthier.

[33] Erik Martin-Dorel. Docker images of mathcomp, 2018–2021. https://github.com/math-comp/docker-mathcomp.

[34] Erik Martin-Dorel. A gentle introduction to container-based CI for Coq projects. The Coq Workshop, July 2020.

[35] The mathlib Community. The Lean Mathematical Library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[36] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Deep generation of Coq lemma names using elaborated terms. In *International Joint Conference on Automated Reasoning*, pages 97–118, 7 2020. https://arxiv.org/abs/2004.07761.

[37] Russel O'Connor. Goedel. https://github.com/coq-community/goedel. A Coq-community project.

[38] Russell O'Connor. Essential incompleteness of arithmetic verified by Coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 245–260, Berlin, Heidelberg, 2005. Springer. https://arxiv.org/abs/cs/0505034.

[39] Benjamin Pierce et al. Software Foundations. https://softwarefoundations.cis.upenn.edu.

[40] Clément Pit-Claudel. Alectryon. https://github.com/cpitclaudel/alectryon.

[41] Clément Pit-Claudel. Untangling mechanized proofs. In *International Conference on Software Language Engineering*, pages 155–174, New York, NY, USA, 2020. Association for Computing Machinery. https://dl.acm.org/doi/pdf/10.1145/3426425.3426940.

[42] Hans Jürgen Prömel. Rapidly growing Ramsey functions. In *Ramsey Theory for Discrete Structures*, pages 97–103. Springer, Cham, 2013. https://link.springer.com/chapter/10.1007/978-3-319-01315-2_8.

[43] Kurt Schütte. *Proof Theory*. Springer, 1977. https://link.springer.com/book/10.1007%2F978-3-642-66473-1.

[44] Ilya Sergey. Programs and Proofs: Mechanizing Mathematics with Dependent Types, 2014. https://doi.org/10.5281/zenodo.4996238.

[45] Carlos Simpson. Category theory in ZFC. User Contributions to the Coq Proof Assistant, 2004. https://github.com/coq-contribs/cats-in-zfc.

[46] Carlos Simpson. Set-theoretical mathematics in Coq, 2004. https://arxiv.org/abs/math/0402336.

[47] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP), July 2019. https://hal.inria.fr/hal-01671777.

[48] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293, Berlin, Heidelberg, 2008. Springer. https://sozeau.gitlabpages.inria.fr/www/research/publications/First-Class_Type_Classes.pdf.

[49] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical*

Structures in Computer Science, 21(4):795–825, 2011. https://arxiv.org/abs/1102.1323.

[50] The Dune authors. Dune: A composable build system for OCaml, 2016–2021. https://dune.build.

[51] Floris van Doorn, Gabriel Ebner, and Robert Y. Lewis. Maintaining a library of formal mathematics. In Christoph Benzmüller and Bruce Miller, editors, *Intelligent Computer Mathematics*, pages 251–267, Cham, 2020. Springer International Publishing.

[52] Stan Wainer. A classification of the ordinal recursive functions. *Archiv für mathematische Logik und Grundlagenforschung*, 13(3):136–153, Dec 1970. https://link.springer.com/article/10.1007%2FBF01973619.

[53] Stan Wainer and Wilfried Buchholz. Provably computable functions and the fast growing hierarchy. In Stephen G. Simpson, editor, *Contemporary Mathematics*, volume 65, pages 179–198. American Mathematical Society, Providence, RI, USA, 1987. http://nbn-resolving.de/urn/resolver.pl?urn=nbn:de:bvb:19-epub-3843-7.

[54] Théo Zimmermann. *Challenges in the collaborative evolution of a proof language and its ecosystem*. PhD thesis, Université de Paris, 2019. https://hal.inria.fr/tel-02451322.

[55] Théo Zimmermann and Jean-Rémy Falleri. A grounded theory of community package maintenance organizations-registered report. In *ICSME 2021-37th International Conference on Software Maintenance and Evolution*, 2021.

Macle : un langage dédié à l'accélération de programmes OCaml sur circuits FPGA

Loïc Sylvestre[1], Jocelyn Sérot[2] et Emmanuel Chailloux[1]

[1] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
loic.sylvestre@lip6.fr
emmanuel.chailloux@lip6.fr
[2] Institut Pascal, UMR 6602 UCA/CNRS/SIGMA
jocelyn.serot@uca.fr

Résumé

Les circuits reprogrammables de types FPGA constituent un matériel de choix pour la réalisation d'accélérateurs de calculs. L'implémentation O2B de la machine virtuelle OCaml permet d'appeler des circuits externes réalisés sur FPGA depuis un langage de haut niveau (OCaml) compilé en bytecode. La conception de circuits s'appuie sur des langages de description de matériel (HDL) souvent fort éloignés des langages algorithmiques. C'est pourquoi l'on présente Macle, un langage applicatif dédié à la programmation de calculs séquentiels et parallèles synthétisables en circuits. On décrit la chaîne de compilation de Macle vers un HDL et son intégration à O2B.

1 Introduction

Contexte Les FPGA (*Field-Programmable Gate Array*) sont des circuits composés d'un grand nombre [1] de cellules logiques configurables dont les interconnexions peuvent être elles-mêmes re-programmées. Ce type de matériel fait l'objet d'un intérêt croissant de la part des programmeurs, notamment d'applications embarquées à performance ou sûreté critique, puisqu'il permet la réalisation d'architectures spécifiques modifiables. Mais programmer le matériel est difficile : les détails d'implantation rendent les codes sources complexes et sujets aux erreurs de programmation ou de spécification [2]. La programmation efficace de ce type de circuit passe par l'usage de langages de description de matériel (*Hardware Description Languages*, HDL) comme VHDL ou Verilog [13]. Ces langages décrivent les fonctions logiques à réaliser au niveau dit transfert de registre (*Register Transfer*, RT), c.à.d., essentiellement, des opérations élémentaires sur un ensemble de registres constituant l'état du programme. En somme, la description de circuits efficaces dans un HDL nécessite non seulement des connaissances en électronique numérique mais aussi l'expérience des outils de synthèse [3].

La programmation impérative dans des langages comme C, C++ ou Python, projetés en matériel via des outils de *synthèse haut niveau* (*High Level Synthesis*) comme OpenCL [5] permet de pallier en partie ces difficultés. Mais ces outils laissent peu de contrôle au programmeur quant à l'efficacité et la fiabilité du code ainsi engendré au niveau RT.

Les langages de programmation fonctionnels ont parfois été présentés comme une solution à ces limitations. Outre le niveau d'abstraction offert, la sémantique de ces langages s'accorde en

1. De quelques dizaines à plusieurs centaines de milliers.
2. Un circuit peut être synthétisable sans pour autant implémenter correctement la fonctionnalité souhaitée.
3. Chaque outil de synthèse supporte un sous-ensemble des descriptions exprimables dans un HDL. Maîtriser la syntaxe et la sémantique (complexe) d'un HDL ne suffit pas : il faut comprendre de quelle manière une description est synthétisée.

effet mieux au modèle d'exécution intrinsèquement concurrent du niveau RT [6]. En pratique, plusieurs approches ont été suivies :

1. Embarquer, dans un langage fonctionnel généraliste comme Haskell, un langage dédié (*Domain Specific Language*, DSL) permettant la description de matériel. On peut citer Lava [2], Hawk [9] et Cλash [1] en Haskell, HardCaml en OCaml [7], Kami [3] en Coq. Le programmeur bénéficie alors des traits de haut niveau du langage hôte pour spécifier des circuits.
2. Compiler un langage fonctionnel éventuellement restreint vers des FPGA. C'est l'approche suivie, par exemple, par le langage SAFL [10](*Statically Allocated Parallel Functional language*), le langage Hume [15](*Higher-order Unified Meta-Environment*), le compilateur SHard [12] pour un sous-ensemble de Scheme et le projet FHW [16, 19] pour Haskell.
3. Implémenter une machine virtuelle pour un langage fonctionnel ciblant des FPGA. C'est l'approche suivie par O2B [14] (OCaml *on Board*), une adaptation d'OMicroB [17] sur le processeur *softcore* Nios2 d'Intel (ex-Altera). Pour « tailler sur mesure » le matériel et ainsi programmer des applications hautement optimisées, O2B propose une approche *hybride* via l'ajout de primitives permettant l'appel des fonctions externes directement implémentées au niveau RT, depuis des programmes OCaml.

Cet article présente Macle (*autoMata-based AppliCative LanguagE*), un langage dédié à l'accélération de programmes OCaml sur FPGA. Macle permet de programmer des circuits utilisables comme fonctions externes en OCaml via O2B.

Plan Après une présentation de l'approche « machine virtuelle » suivie par O2B (section 2), cet article définit le langage Macle (section 3), donne une vue d'ensemble de la chaîne de compilation Macle vers un HDL (section 4) puis décrit ses langages intermédiaires (section 5) ainsi que l'interopérabilité entre Macle et OCaml (section 6). Suivent des tests de performance pour valider notre démarche (section 7), une présentation de travaux connexes (section 8) et une conclusion qui suggère des pistes de travaux futurs (section 9).

2 La plateforme O2B

La plateforme expérimentale O2B (OCaml *on Board* [14]) est une adaptation de l'implémentation OMicroB [17] de la machine virtuelle OCaml pour cibler des processeurs *softcore* réalisés sur circuits FPGA. Écrite en C, O2B est étendue de manière à pouvoir exécuter des fonctions externes implantées directement en matériel sous forme de cellules logiques. De telles fonctions – que nous appelons *circuits*[4] – ont alors accès à l'ensemble des dispositifs d'entrée-sortie d'un FPGA[5] et peuvent réaliser des calculs de manière très efficace en tirant parti du parallélisme massif disponible sur celui-ci et du fait que ces calculs peuvent être cadencés directement par l'horloge de base du FPGA.

La communication entre les circuits et le bytecode OCaml interprété par O2B se fait via un ensemble de registres dédiés associés au FPGA et *mappés* dans la mémoire du processeur *softcore*. Il est aussi possible de partager directement de la mémoire entre le processeur et les circuits.

4. Ce sont des *custom components* dans la terminologie des outils de développement Intel.
5. Elles peuvent ainsi lire la valeur d'un capteur ou activer une LED par exemple.

Le flot de compilation permettant d'engendrer la configuration complète du FPGA correspondant à un programme source est décrit sur la figure 1 dans le cas d'un FPGA de la famille Intel [6]. Le bytecode d'un programme OCaml – engendré par le compilateur `ocamlc` – est transformé en tableau C via l'outil `bc2c` de l'environnement OMicroB. Ce tableau est embarqué dans le programme C implémentant l'interpréteur de bytecode et la bibliothèque d'exécution (*runtime*) d'OMicroB. Ce programme est associé aux fonctions du *Board Support Package* donnant accès aux ressources matérielles de la carte cible ; l'ensemble étant compilé en un code exécutable par le processeur *softcore* (Nios2 dans le cas des FPGA Intel).

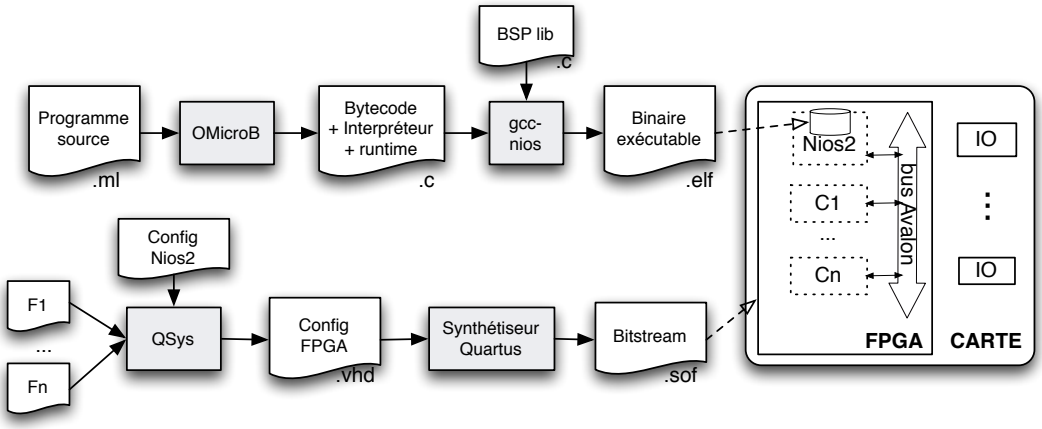


Figure 1 – Flot de compilation O2B vers une cible Intel

La configuration complète du FPGA comprend l'architecture exacte du processeur *softcore* utilisé, la description des fonctions externes F1 ··· Fn devant être implantées sous la forme de circuits C1 ··· Cn ainsi que, le cas échéant, le nombre et la nature des périphériques associés. Techniquement, cette étape de configuration est assurée par l'outil QSys de la chaîne Quartus d'Intel. Elle engendre un ensemble de fichiers VHDL qui constituent la description de la plateforme matérielle. C'est cette description – une fois synthétisée par les outils de la chaîne Quartus pour produire un fichier de configuration (ou *bitstream*) – qui est utilisée pour re-configurer le FPGA.

La version d'O2B présentée dans [14] laissait au programmeur le soin de spécifier lui-même des circuits dans un langage de description de matériel (VHDL). L'étape de configuration sous QSys, mais aussi la réalisation d'une interface de communication entre ces circuits et l'environnement d'exécution OCaml étaient effectuées manuellement. Dans la version courante d'O2B, les descriptions des circuits en VHDL ainsi que l'interface de communication sont engendrées depuis un langage de programmation dédié (Macle). L'étape de configuration est elle aussi automatisée par un ensemble de scripts : la génération du *bitstream* étant devenue entièrement automatique, la programmation FPGA est ainsi mise à la portée des programmeurs sans expérience préalable en description de matériel, en vérification voire en synthèse de circuits.

6. Seule famille actuellement supportée par O2B – mais l'approche suivie permet de cibler facilement d'autre familles de circuits (Xilinx par exemple).

3 Le langage Macle

Macle est un langage dédié à la programmation de circuits FPGA. Il permet d'enrichir la bibliothèque d'exécution d'un langage algorithmique comme OCaml en y ajoutant des fonctions externes automatiquement accélérées en matériel. Ces fonctions, comme le circuit qui calcule le n-ième terme de la suite de Fibonacci ci-dessous, sont définies par le programmeur dans un style applicatif.

```
circuit fibonacci n =
  let rec f a b n =
    if n < 1 then a else f b (a + b) (n - 1)
  in f 0 1 n
```

3.1 Motivation

Les fonctions Macle sont compilées sous forme de circuits spécifiés dans un langage de description de matériel classique [7]. Nous souhaitons que ces circuits soient efficaces et facilement utilisables comme fonctions externes depuis des programmes OCaml exécutés par O2B. Naturellement, cela conduit à des choix dans la conception du langage.

Macle est un langage strict en appel par valeur avec une syntaxe *à la ML*. Par construction, les circuits engendrés à partir de Macle n'allouent pas de mémoire et n'utilisent pas de pile d'appels. Schématiquement, la compilation de Macle vers VHDL expanse (*i.e.* intègre) les applications de fonctions, transforme en automates les définitions de fonctions mutuellement récursives terminales et implémente de façon parallèle les liaisons locales (*let ... and ...*) ainsi que les applications d'opérateurs, sous forme de barrières de synchronisation.

Ce schéma de compilation, visant à produire des circuits efficaces, restreint cependant l'expressivité du langage. D'une part, les récursions non terminales sont rejetées statiquement. D'autre part, les fonctions Macle ne sont pas des valeurs à part entière ; elles peuvent seulement être appliquées ou passées en paramètres à d'autres fonctions. Le système de types de Macle est monomorphe. Il garantit que la compilation d'un circuit bien typé produit une description VHDL synthétisable.

3.2 Syntaxe

La syntaxe du noyau de Macle explicitement typé [8] est définie figure 2. Nous appelons $\mathcal{X}$ un ensemble dénombrable de noms dénotés par les lettres x et f. Un *circuit* est une définition de fonction globale dont le corps est une expression. Une expression est une variable x, une constante c, une application d'opérateur ou de fonction, une conditionnelle ou une définition locale. Chaque valeur est typée avec un type de base τ. Un type σ est soit un type de base, soit un type fonctionnel $\sigma \to \sigma$. Chaque fonction est typée avec un type fonctionnel. Chaque définition de fonction B comporte un nom x, une séquence d'arguments w, un type de retour τ et un corps e. Les fonctions peuvent prendre en paramètres des fonctions. Une analyse statique sur la syntaxe des expressions empêche la définition de fonctions récursives non terminales. La classe des opérateurs binaires (*resp.* unaires) comporte uniquement des fonctions combinatoires.

7. VHDL dans notre cas, mais rien ne s'oppose à la génération de Verilog.
8. Il correspond à l'arbre de syntaxe engendré par inférence des types à partir des programmes sources Macle non typés.

circuit	$\phi ::= \textsf{circuit}\ B$	*variable*	$x, f \in \mathcal{X}$
fonction	$B ::= f\ w : \tau = e$	*opérateur unaire*	$\ominus ::= - \mid \textsf{not}$
arguments	$w ::= (x_1 : \sigma_1)\ \cdots\ (x_n : \sigma_n)$	*opérateur binaire*	$\oplus ::= + \mid\ =\ \mid\ <\ \mid \cdots$
expression	$e ::= x$	*constante*	$c ::= \textsf{true} \mid \textsf{false} \mid n$
	$\mid c$	*type de base*	$\tau ::= \texttt{bool} \mid \texttt{int}$
	$\mid \ominus e$	*type*	$\sigma ::= \tau \mid \sigma \to \sigma$
	$\mid e_1 \oplus e_2$		
	$\mid f\ e_1 \cdots e_n$		
	$\mid \textsf{if}\ e_1\ \textsf{then}\ e_2\ \textsf{else}\ e_3$		
	$\mid \textsf{let}\ x_1 : \tau_1 = e_1\ \textsf{and} \cdots x_n : \tau_n = e_n\ \textsf{in}\ e$		
	$\mid \textsf{let rec}\ B_1\ \textsf{and} \cdots B_n\ \textsf{in}\ e$		
	$\mid \textsf{let}\ B\ \textsf{in}\ e$		

Figure 2 – Syntaxe de Macle explicitement typé

3.3 Typage

Le typage de Macle est défini figures 3 et 4 par un ensemble de règles. L'environnement de typage Γ (*resp.* l'environnement de typage des opérateurs Δ) comporte des liaisons de la forme $\Gamma(x) = \sigma$ (*resp.* $\Delta(op) = \sigma$). Le jugement $\Gamma \vdash e : \tau$ signifie « dans l'environnement de typage Γ, l'expression e est bien typée, de type τ ». Le jugement $\vdash_{\textsf{circuit}} \phi : \tau$ signifie « le circuit ϕ est bien typé, de type τ ». Le jugement $\Gamma \vdash_{\textsf{arg}} e : \sigma$ signifie « dans l'environnement de typage Γ, l'argument e est bien typé, de type σ ».

L'environnement vide est noté $\varnothing$. L'extension $\Gamma[x : \sigma]$ de l'environnement Γ avec le nom x de type σ est définie par $\Gamma[x : \sigma](x) = \sigma$ et $\Gamma[x : \sigma](y) = \Gamma(x)$ si $x \neq y$. L'abréviation $\Gamma[x_1 : \sigma_1,\ \cdots\ x_n : \sigma_n]$ désigne l'environnement $\Gamma[x_1 : \sigma_1] \cdots [x_n : \sigma_n]$. Étant donnée une suite d'arguments $w \triangleq (x_1 : \sigma_1) \cdots (x_n : \sigma_n)$, les abréviations $\Gamma[w]$ et $\texttt{fun-ty}(w, \tau)$ désignent respectivement l'environnement $\Gamma[x_1 : \sigma_1, \cdots x_n : \sigma_n]$ et le type $\sigma_1 \to \cdots \sigma_n \to \tau$.

Let
$$\frac{i \in \{1, \cdots n\} \qquad \Gamma \vdash e_i : \tau_i \qquad \Gamma[x_1 : \tau_1, \cdots x_n : \tau_n] \vdash e : \tau}{\Gamma \vdash \textsf{let}\ x_1 : \tau_1 = e_1\ \textsf{and} \cdots x_n : \tau_n = e_n\ \textsf{in}\ e : \tau}$$

Arg-Val
$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash_{\textsf{arg}} e : \tau}$$

Arg-Fun
$$\frac{\Gamma(f) = \sigma_1 \to \sigma_2}{\Gamma \vdash_{\textsf{arg}} f : \sigma_1 \to \sigma_2}$$

Let-Fun
$$\frac{\Gamma[w] \vdash e : \tau \qquad \Gamma[f : \texttt{fun-ty}(w, \tau)] \vdash e' : \tau'}{\Gamma \vdash \textsf{let}\ f\ w : \tau = e\ \textsf{in}\ e' : \tau'}$$

App
$$\frac{\Gamma(x) = \sigma_1 \to \cdots \sigma_n \to \tau \qquad i \in \{1, \cdots n\} \qquad \Gamma \vdash_{\textsf{arg}} e_i : \sigma_i}{\Gamma \vdash x\ e_1 \cdots e_n : \tau}$$

LetRec
$$\frac{\begin{array}{c}\textsf{first-order}(\texttt{fun-ty}(w_i, \tau_i)) \\ \Gamma' = \Gamma[f_1 : \texttt{fun-ty}(w_1, \tau_1), \cdots f_n : \texttt{fun-ty}(w_n, \tau_n)] \\ i \in \{1, \cdots n\} \qquad \Gamma'[w_i] \vdash e_i : \tau_i \qquad \Gamma' \vdash e : \tau\end{array}}{\Gamma \vdash \textsf{let rec}\ f_1\ w_1 : \tau_1 = e_1\ \textsf{and} \cdots f_n\ w_n : \tau_n = e_n\ \textsf{in}\ e : \tau}$$

Circuit
$$\frac{\textsf{first-order}(\texttt{fun-ty}(w, \tau)) \qquad \varnothing[w] \vdash e : \tau}{\vdash_{\textsf{circuit}}\ \textsf{circuit}\ f\ w : \tau = e : \texttt{fun-ty}(w, \tau)}$$

Figure 3 – Typage de Macle (1)

Les règles Arg-Val, Arg-Fun et App vérifient le bon typage des applications de fonctions ; les arguments étant soit des expressions, soit des noms de fonctions. Un circuit est une fonction

globale typée dans un environnement de typage vide (règle CIRCUIT). La condition de bord $\mathsf{first\text{-}order}(\sigma_1 \to \cdots \sigma_n)$ dans les règles LETREC et CIRCUIT impose que chaque type σ_i soit un type de base. Il en résulte que les définitions bien typées de circuits et de fonctions récursives terminales ne prennent pas de fonctions en paramètres. Les autres règles sont standard.

$$\textsc{True}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathtt{bool}} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathtt{bool}} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash n : \mathtt{int}} \qquad \textsc{Var}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \textsc{Unop}\ \frac{\Delta(\ominus) = \tau_1 \to \tau_2 \qquad \Gamma \vdash e : \tau_1}{\Gamma \vdash \ominus\, e : \tau_2}$$

$$\textsc{If}\ \frac{\Gamma \vdash e : \mathtt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \tau} \qquad \textsc{Binop}\ \frac{\Delta(\oplus) = \tau_1 \to \tau_2 \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \oplus e_2 : \tau}$$

Figure 4 – Typage de MACLE (2)

Grâce à cette discipline de types, et en particulier les restrictions qu'elle impose quant à l'utilisation des fonctions, nous pouvons garantir que les techniques de compilation présentées dans les sections suivantes s'appliquent correctement à tout programme MACLE bien typé.

4 Compilation de MACLE

La chaîne de compilation MACLE est schématisée figure 5. Elle vise à produire, à partir d'une spécification haut niveau, la description d'un circuit en VHDL portable et synthétisable sur FPGA. Les fonctions MACLE sont d'abord typées, puis traduites dans un langage intermédiaire (VSML) dans lequel sont réalisées des optimisations. Suit un second langage intermédiaire (ESML) à partir duquel la description VHDL est engendrée puis synthétisée pour obtenir le fichier de configuration du circuit FPGA. Une interface de communication (FFI O2B) – fondée sur la couche d'interopérabilité existante entre C et OCAML – permet l'appel des circuits MACLE depuis du bytecode OCAML exécutable par O2B sur le même FPGA cible. À des fins de simulation sur PC, MACLE et VSML sont traduits en OCAML tandis qu'ESML est interprété par un évaluateur écrit en OCAML suivant la sémantique décrite en sous-section 5.2.

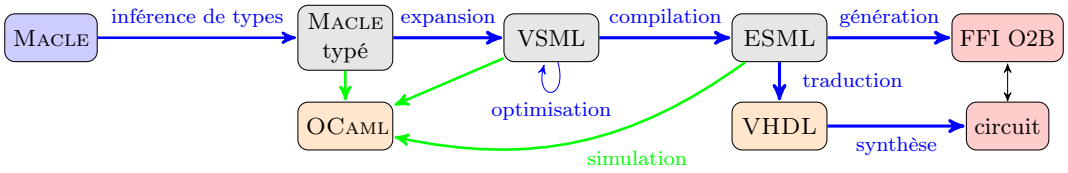


Figure 5 – Compilation de MACLE vers circuit

4.1 Exemple de compilation

La figure 6 illustre les étapes de traduction menant d'une fonction MACLE très simple (somme des carrés de 1 à n) à une description VHDL. Les langages intermédiaires VSML et ESML sont présentés plus en détails à la section 5. L'encodage de MACLE dans VSML transforme la fonction récursive locale S en une construction d'automate à états paramétrés,

MACLE

```
circuit somme_carres n =
  let carre x = x × x in
  let rec S m acc =
    if m < 1 then acc else
    S (m − 1) (acc + square m)
  in S n 0
```

VSML

```
circuit somme_carres (n : int) : int =
  let automaton
      S(m : int, acc : int) →
        if m < 1 then acc
        else S(m − 1, acc + m × m)
      end
  in S(n, 0)
```

ESML

```
circuit somme_carres : sig
    input start : bool,
    input n : int,
    output rdy : bool,
    output result : int,
    local m : int,
    local acc : int
  end =
  let automaton
        Idle →
          if start then
            do rdy ≔ false
             and m ≔ n
             and acc ≔ 0
             then
             continue S
          else
            do rdy ≔ true then
            continue Idle
      | S →
          if m < 1 then
            do result ≔ acc
            then continue Idle
          else
            do m ≔ m − 1
            and acc ≔ m × m
            then continue S
  end
  in continue Idle
```

VHDL

```
library ieee;                                                    1
use ieee.std_logic_1164.all;                                     2
use ieee.numeric_std.all;                                        3
                                                                 4
entity somme_carres is                                           5
port(signal clk : in std_logic;                                  6
     signal reset : in std_logic;                                7
     signal n : in signed(30 downto 0);                          8
     signal start : in std_logic;                                9
     signal rdy : out std_logic;                                 10
     signal result : out signed(30 downto 0));                   11
end entity;                                                      12
                                                                 13
architecture RTL of somme_carres is                              14
  signal m : signed(30 downto 0);                                15
  signal acc : signed(30 downto 0);                              16
  type state_t is (Idle,S);                                      17
  signal STATE : state_t;                                        18
begin                                                            19
  process(reset,clk) begin                                       20
    if reset = '1' then                                          21
      rdy <= true;                                               22
      result <= to_signed(0,31);                                 23
      m <= to_signed(0,31);                                      24
      acc <= to_signed(0,31);                                    25
      STATE <= Idle;                                             26
    elsif rising_edge(clk) then                                  27
      case STATE is                                              28
        when Idle =>                                             29
          if start = '1' then                                    30
            rdy <= false;                                        31
            m <= n;                                              32
            acc <= to_signed(0,31);                              33
            STATE <= S;                                          34
          else                                                   35
            rdy <= true;                                         36
            STATE <= Idle;                                       37
          end if;                                                38
        when S =>                                                39
          if m < to_signed(1,31) then                            40
            result <= acc;                                       41
            STATE <= Idle;                                       42
          else                                                   43
            m <= m - to_signed(1,31);                            44
            acc <= acc + resize(m * m,31);                       45
            STATE <= S;                                          46
          end if;                                                47
      end case;                                                  48
    end if;                                                      49
  end process;                                                   50
end architecture;                                                51
```


Figure 6 – Étapes de compilation d'un circuit MACLE vers VHDL

pouvant potentiellement se terminer et retourner une valeur. La compilation de VSML vers ESML remplace les états paramétrés par des variables « globales », explicite les terminaisons des calculs via un nouvel état « puit » Idle et ajoute des variables de synchronisation permettant l'appel (start) du circuit engendré et l'attente du résultat (rdy, result). ESML est finalement traduit en un sous-ensemble de VHDL synthétisable.

4.2 Codage en VHDL

L'encodage de l'automate en VHDL (dans le cadre droit) se fait, classiquement, sous la forme d'un *process* synchrone contrôlé par un signal d'horloge clk pour cadencer l'exécution du circuit et un signal rst pour initialiser l'automate (de manière asynchrone). Ce *process* manipule des signaux représentant d'une part les entrées, sorties et variables locales de l'automate ESML et d'autre part un signal **STATE** mémorisant l'état courant du système. Dans l'exemple de la figure 6, ce signal peut prendre deux valeurs : Idle et S, correspondant aux deux états de l'automate ESML. Les lignes 22 à 26 sur la figure décrivent ce qui se passe lors d'un *reset* asynchrone : les signaux rdy, result, m et acc sont initialisés et l'état de l'automate est positionné à la valeur Idle. Les lignes 28 à 49 décrivent ce qui se passe lors d'un front montant de l'horloge clk, front montant qui est la réalisation matérielle du « top » cadençant implicitement l'automate ESML. A chaque fois, en fonction de la valeur de l'état courant, un certain nombre de signaux sont modifiés [9], y compris, possiblement, celui codant l'état (*e.g.* **STATE** <= Idle). Un point fondamental est que ces modifications sont effectuées de manière *parallèle synchrone*. Lorsque plusieurs signaux sont modifiés lors d'une même activation du *process* (comme les signaux rdy, m, acc et **STATE** aux lignes 31 à 34, la mise à jour de ces signaux s'effectue non pas de manière séquentielle, mais simultanément pour l'ensemble des signaux concernés et à la fin de l'activation du *process*. Cela signifie en particulier que la valeur d'un signal apparaissant à droite du signe <= est toujours celle qu'a ce signal au début de l'activation, la nouvelle valeur éventuellement affectée à ce signal n'étant visible qu'à la *prochaine* activation du *process*. Dans l'exemple sus-cité, nous pouvons voir :

```
m <= m - to_signed(1,31);
acc <= acc + resize(m * m,31);
STATE <= S;
```

Ici, la valeur de m utilisée dans l'appel à `resize` ne dépend pas de la soustraction de la ligne précédente [10]. Cette sémantique particulière est liée à la manière dont les machines à états sont synthétisées en matériel [11]. C'est elle qui justifiera en particulier la sémantique opérationnelle donnée au langage ESML à la sous-section 5.2, notamment quant aux notions d'*environnement courant* et d'*environnement de sortie*.

5 Langages intermédiaires

La section précédente a donné une vue d'ensemble de la chaîne de compilation de Macle vers VHDL, illustrant le comportement des circuits au niveau RT à partir d'un exemple de fonction Macle qui – par étapes de traduction successives à travers deux langages intermédiaires (VSML et ESML) – est implémentée par un automate synchrone encodé en VHDL. Ces deux langages de

9. Via la syntaxe `<signal> <= <expression>`.
10. Et, de fait, l'ordre dans lequel ces deux lignes sont écrites est parfaitement indifférent.
11. Sous la forme d'un machine dite de Moore dans lequel l'état est codé dans un registre mis à jour à chaque front montant avec une valeur calculée à partir de sa sortie et des entrées.

machines à états sont des abstractions de Macle pour VSML et de VHDL pour ESML. Après une présentation de la syntaxe de VSML on s'intéresse principalement à la définition d'une sémantique d'évaluation d'ESML pour comprendre le modèle d'exécution des circuits définis en Macle. Par contre la traduction de VSML vers ESML est indiquée de manière succinte.

5.1 VSML

VSML (*Valued State Machine Language*) est un sous-ensemble de Macle dans lequel toutes les applications de fonctions sont en position terminale. Les circuits Macle sont mis sous cette forme par expansion complète des appels de fonctions à l'exception des fonctions mutuellement récursives. Les boucles récursives résultantes s'apparentent à des automates hiérarchiques avec états paramétrés *à la* Lucid Synchrone [4].

La syntaxe de VSML est définie figure 7. Un circuit ϕ est une déclaration de fonction globale. Une expression e est soit un atome a, soit une structure de contrôle. La classe des atomes comprend les constantes, variables et applications d'opérateurs. VSML comporte quatre structures de contrôle : *application d'état*, conditionnelle, liaisons locales ou automate à états paramétrés. Les états sont dénotés par la lettre q.

circuit	$\phi ::= \textbf{circuit}\ f\ w : \tau = e$
transition	$t ::= q\ w \to e$
arguments	$w ::= (x_1 : \tau_1,\ \cdots\ x_n : \tau_n)$
expression	$e ::= a$
	$\mid q(a_1, \cdots a_n)$
	$\mid \textbf{if}\ a\ \textbf{then}\ e_1\ \textbf{else}\ e_2$
	$\mid \textbf{let}\ x_1 : \tau_1 = e_1\ \textbf{and}\ \cdots\ x_n : \tau_n = e_n\ \textbf{in}\ e$
	$\mid \textbf{let automaton}\ t_1 \mid \cdots\ t_n\ \textbf{end in}\ e$
atome	$a ::= x \mid c \mid\ \ominus\, a \mid a_1 \oplus a_2$

variable	$x, q, f \in \mathcal{X}$
opérateur unaire	$\ominus ::= - \mid \textbf{not}$
opérateur binaire	$\oplus ::= + \mid\ =\ \mid\ <\ \mid \cdots$
constante	$c ::= \textbf{true} \mid \textbf{false} \mid n$
type de base	$\tau ::= \texttt{bool} \mid \texttt{int}$

Figure 7 – Syntaxe de VSML

En VSML, les constructions de langage sont simples et référentiellement transparentes, facilitant la mise en œuvre d'optimisations de code. Le compilateur Macle implémente d'une part une passe de propagation des atomes et d'autre part une forme de *let-floating* [11] visant à maximiser le degré de parallélisme.

5.2 ESML

ESML (*Extended State Machine Language*) est un langage parallèle synchrone qui intègre un nombre réduit de constructions de VHDL, en particulier la compositions d'automates synchrones et l'affectation.

La syntaxe d'ESML est définie figure 8. Un circuit Φ comprend une signature S et un corps P. La signature explicite les entrées, les sorties et les variables locales du circuit ainsi que leurs types respectifs. Le corps du circuit est une composition parallèle d'automates $A_1 \parallel \cdots A_n$ [8] tous cadencés par une même horloge globale. Chaque automate A est formé d'un ensemble de transitions et d'une instruction d'initialisation. Chaque transition t associe à un état source q une instruction.

L'instruction **continue** q suspend l'exécution de l'automate englobant jusqu'au top d'horloge suivant et fait passer celui-ci dans l'état q. L'instruction **do** $x_1 \coloneqq a_1$ **and** $\cdots\ x_n \coloneqq a_n$ **then** s lie les

circuit	$\Phi ::= \textsf{circuit}\ f : S = P$	*variable*	$x, q, f \in \mathcal{X}$
produit	$P ::= A \parallel P \mid A$	*opérateur unaire*	$\ominus ::= - \mid \textsf{not}$
automate	$A ::= \textsf{let automaton}\ t_1 \mid \cdots\ t_n\ \textsf{end in}\ s$	*opérateur binaire*	$\oplus ::= + \mid = \mid < \mid \cdots$
transition	$t ::= q \to e$	*constante*	$c ::= \textsf{true} \mid \textsf{false} \mid n$
instruction	$s ::= \textsf{continue}\ q \mid \textsf{if}\ a\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \mid \textsf{do}\ x_1 := a_1\ \textsf{and}\ \cdots\ x_n := a_n\ \textsf{then}\ s$	*signature*	$S ::= \textsf{sig}\ d_1, \cdots\ d_n\ \textsf{end}$
		déclaration	$d ::= m\ x : \tau$
atome	$a ::= x \mid c \mid \ominus a \mid a_1 \oplus a_2$	*mode*	$m ::= \textsf{input} \mid \textsf{output} \mid \textsf{local}$
		type de base	$\tau ::= \texttt{bool} \mid \texttt{int}$

Figure 8 – Syntaxe des circuits ESML

valeurs v_i des atomes a_i aux variables x_i dans l'instruction s (à la manière d'un *let ... and ...*), puis écrit $x_i := v_i$ dans l'environnement [12] au top d'horloge suivant. Un exemple de circuit ESML a été donné figure 6.

Les figures 9, 10 et 11 définissent la sémantique opérationnelle à réduction [18] d'ESML. Toute valeur ESML est soit une constante c, soit un glaçon $\#\langle A \rangle$ représentant l'état d'un automate A à la fin d'un instant synchrone. Un environnement d'exécution ρ comporte des liaisons de la forme $\rho(x) = v$. L'extension d'un environnement ρ est notée $\rho[x_1 \mapsto v_1, \cdots x_n \mapsto v_n]$ de même que précédemment. Un contexte d'évaluation E (*resp.* F) est un atome (*resp.* une instruction) avec un unique trou $[\,]$ dans lequel s'insère un atome. Un contexte d'évaluation M est un automate avec un unique trou $\langle\rangle$ dans lequel s'insère une instruction.

$$
\begin{array}{l}
v ::= c \mid \#\langle A \rangle \\
E ::= \ominus\,[\,] \mid [\,] \oplus a \mid v \oplus [\,] \\
F ::= \textsf{if}\ [\,]\ \textsf{then}\ s_1\ \textsf{else}\ s_2 \\
\quad \mid \textsf{do}\ x_1 := [\,]\ \textsf{and}\ x_2 := a_2\ \textsf{and}\ \cdots\ x_n := a_n\ \textsf{then}\ s \\
\quad \mid \cdots \\
\quad \mid \textsf{do}\ x_1 := v_1\ \textsf{and}\ \cdots\ x_{n-1} := v_{n-1}\ \textsf{and}\ x_n := [\,]\ \textsf{then}\ s \\
M ::= \textsf{let automaton}\ ts\ \textsf{end in}\ \langle\rangle
\end{array}
$$

Figure 9 – Valeurs et contextes d'évaluation ESML

La relation $\rho_0 \vdash a \longrightarrow a'$ signifie « dans l'environnement ρ_0, l'atome a se réduit en a' ». Elle est définie par la règle d'inférence Context-Atom (ou réduction sous un contexte d'évaluation E) ainsi que trois règles de réécriture : R-Unop, R-Binop et R-Var. L'application d'un opérateur $(\ominus\ v)$ (*resp.* $v_1 \oplus v_2$) se réduit *toujours* en une valeur, notée $[\![\ominus]\!](v)$ (*resp.* $[\![\oplus]\!](v_1, v_2)$).

La réduction d'une instruction s ou d'un automate A dans un environnement courant ρ_0 met à jour l'environnement ρ de l'instant synchrone suivant (ou *environnement de sortie*). La relation $\rho_0 \vdash s; \rho \longrightarrow s'; \rho'$ signifie « dans l'environnement courant ρ_0 l'instruction s et l'environnement de sortie ρ, se réduisent en s' dans l'environnement de sortie ρ' ». Elle est définie par la règle d'inférence Context-Instruction et trois règles de réécriture : R-If-True, R-If-False et R-Do. Intuitivement, l'instruction $\textsf{do}\ x_1 := v_1\ \textsf{and}\ \cdots\ x_n := v_n\ \textsf{then}\ s$ écrit $x_i := v_i$ dans

12. L'environnement est partagé en lecture. Chaque sortie ou variable locale ne peut être écrite que par un seul et même automate au cours de l'exécution du circuit. Les entrées ne peuvent pas être écrites. Le traduction de VSML en ESML supporte ces contraintes par construction.

l'environnement de sortie, puis calcule l'instruction s en substituant chaque occurrence de x_i par v_i (l'environnement courant étant laissé inchangé).

$$\textsc{Context-Atom}\ \frac{\rho_0 \vdash a \longrightarrow a'}{\rho_0 \vdash E[a] \longrightarrow E[a']} \qquad \textsc{Context-Instruction}\ \frac{\rho_0 \vdash a \longrightarrow a'}{\rho_0 \vdash F[a]; \rho \longrightarrow F[a']; \rho} \qquad \textsc{Context-Automaton}\ \frac{\rho_0 \vdash s/\rho \longrightarrow s'/\rho'}{\rho_0 \vdash M\langle s\rangle;\ \rho \longrightarrow M\langle s'\rangle;\ \rho'}$$

$$\boxed{\rho_0 \vdash a \longrightarrow a'}$$

$$\rho_0 \vdash \ominus\, v \longrightarrow [\![\ominus]\!](v) \quad \text{(R-Unop)}$$
$$\rho_0 \vdash v_1 \oplus v_2 \longrightarrow [\![\oplus]\!](v_1, v_2) \quad \text{(R-Binop)}$$
$$\rho_0 \vdash x \longrightarrow \rho_0(x) \text{ si } x \in \mathsf{dom}(\rho_0) \quad \text{(R-Var)}$$

$$\boxed{\rho_0 \vdash s; \rho \longrightarrow s'; \rho'}$$

$$\rho_0 \vdash \textbf{if true then } s_1 \textbf{ else } s_2; \rho \longrightarrow s_1; \rho \quad \text{(R-If-True)}$$
$$\rho_0 \vdash \textbf{if false then } s_1 \textbf{ else } s_2; \rho \longrightarrow s_2; \rho \quad \text{(R-If-False)}$$
$$\rho_0 \vdash \textbf{do } x_1 \coloneqq v_1 \textbf{ and } \cdots\ x_n \coloneqq v_n \textbf{ then } s; \rho \quad \text{(R-Do)}$$
$$\longrightarrow \{x_1 \mapsto v_1, \cdots x_n \mapsto v_n\}(s); \rho[x_1 \mapsto v_1, \cdots x_n \mapsto v_n]$$

$$\boxed{\rho_0 \vdash A; \rho \longrightarrow A'; \rho'}$$

$$\rho_0 \vdash \textbf{let automaton } ts \textbf{ end in } (\textbf{continue } q); \rho \quad \text{(R-Continue)}$$
$$\longrightarrow \#\langle \textbf{let automaton } ts \textbf{ end in } ts(q)\rangle; \rho$$

Figure 10 – Sémantique des instructions ESML

La relation $\rho_0 \vdash A; \rho \longrightarrow A'; \rho'$ signifie « dans l'environnement courant ρ_0, l'automate A et l'environnement de sortie ρ se réduisent en A' dans l'environnement de sortie ρ' ». Elle est définie par la règle d'inférence Context-Automaton et la règle de réécriture R-Continue réalisant un branchement à l'état suivant de l'automate est l'attente (l'automate est gelé) jusqu'à l'instant synchrone suivant. Dans cette dernière règle, la notation $ts(q_i)$ représente l'instruction s_i telle que $ts \triangleq q_1 \to e_1 \mid \cdots\ q_n \to e_n$ et $i \in \{1, \cdots n\}$.

$$\textsc{Par-left}\ \frac{\rho_0 \vdash A; \rho \longrightarrow A'; \rho'}{A \| P/\rho_0; \rho \longrightarrow A' \| P/\rho_0; \rho'} \qquad \textsc{Par-right}\ \frac{P/\rho_0; \rho \longrightarrow P'/\rho_0; \rho'}{v \| P/\rho_0; \rho \longrightarrow v \| P'/\rho_0; \rho'}$$

$$\boxed{P/\rho_0; \rho \longrightarrow P'/\rho_0'; \rho'}$$

$$\#\langle A_1\rangle \| \cdots\ \#\langle A_n\rangle/\rho_0; \rho' \longrightarrow A_1 \| \cdots\ A_n \| P'/\rho'; \rho' \quad \text{(R-Clock)}$$

$$\boxed{\rho \vdash \Phi \longrightarrow P/\rho; \rho}$$

$$\textbf{circuit } f : \textbf{sig } m_1\ x_1 : \tau_1, \cdots m_n\ x_n : \tau_n \textbf{ end} = P/\rho_0 \longrightarrow P/\rho_0, \varnothing \quad \text{(R-Circuit)}$$
$$\text{si } x_1, \cdots x_n \in \mathsf{dom}(\rho_0)$$

Figure 11 – Sémantique des circuits ESML

La relation $P/\rho_0; \rho \longrightarrow P'/\rho_0'; \rho'$ signifie « le produit P dans l'environnement courant ρ_0 et l'environnement de sortie ρ se réduit en P' dans l'environnement courant ρ_0' et l'environnement de sortie ρ' ». Elle est définie par deux règles d'inférence Par-Left et Par-Right (qui réduisent le produit en laissant intact l'environnement courant) ainsi qu'une règle de réécriture R-Clock (qui réduit un produit totalement évalué $\#\langle A_1\rangle \| \cdots\ \#\langle A_1\rangle/\rho_0; \rho$ en un produit $A_1 \| \cdots\ A_n/\rho; \rho$) caractérisant le passage à l'instant synchrone suivant.

La relation $\rho \vdash \Phi \longrightarrow P/\rho; \rho$ signifie « dans l'environnement initial ρ, le circuit Φ se réduit en P dans l'environnement courant et de sortie ρ ». Elle est définie par la règle de réécriture R-Circuit qui vérifie que la signature S du circuit à réduire est compatible avec l'environnement initial ρ.

La compilation de VSML vers ESML nécessite d'aplatir la hiérarchie d'automates, de globaliser les paramètres des états et de transformer les liaisons locales en composition parallèle d'automates afin d'exploiter le parallélisme implicite des programmes Macle.

6 Interopérabilité entre Macle et OCaml

Pour faciliter le développement d'applications OCaml sur FPGA utilisant O2B, le compilateur Macle engendre non seulement une description de matériel, mais aussi l'ensemble des fichiers nécessaires à la synthèse du circuit et à son utilisation depuis du bytecode OCaml. Le format d'entrée du compilateur Macle supporte ainsi des applications mêlant déclarations Macle et OCaml, tels les deux exemples figure 12. Les déclarations de circuits Macle précèdent le programme OCaml. L'appel *print_int* en OCaml affiche un entier sur la sortie standard en utilisant le lien de communication série (UART) reliant le PC et le FPGA. Le programme OCaml figure 12a compose deux circuits Macle. Le circuit *somme_temps_de_vol4* figure 12b calcule la somme des temps de vol des suites de Syracuse comprises entre deux entiers. Ce calcul s'effectue en parallèle par paquets de 4 éléments à la fois.

```
circuit fact n =
  let rec f acc n =
    if n < 1 then acc else
    f (n × acc) (n − 1)
  in f 1 n

circuit gcd a b =
  let rec g a b =
    if a < b then g a (b − a) else
    if a > b then g (a − b) a
             else a
  in g a b

let n = 10;;

for i = 1 to n do
  print_int (gcd i (fact n))
done;;
```

(a)

```
circuit somme_temps_de_vol4 a b =
  let temps_de_vol n =
    let rec syracuse t n =
      if n ≤ 1 then t else
      if n mod 2 = 0 then syracuse (t + 1) (n/2)
                     else syracuse (t + 1) (3 × n + 1)
    in syracuse 0 n
  in
  let rec somme acc i =
      if i ≥ b then acc else
      if i > b − 4 then somme(acc + temps_de_vol i) (i + 1)
                   else somme (acc + temps_de_vol i
                                   + temps_de_vol (i + 1)
                                   + temps_de_vol (i + 2)
                                   + temps_de_vol (i + 3)) (i + 4)
  in somme 0 a

print_int (somme_temps_de_vol4 1 1024);;
```

(b)

Figure 12 – Programme OCaml étendu avec deux circuits Macle

Les circuits Macle sont paramétrés par des valeurs extérieures (entiers ou booléens) définies au niveau du programme OCaml. Dès lors, il est naturel d'étendre Macle avec de nouvelles constructions syntaxiques pour que ces circuits puissent déréférencer voire modifier physiquement des valeurs allouées dans le tas OCaml.

La figure 13 définit une extension de Macle avec filtrage de listes OCaml, accès et modification de références OCaml. Trois constructeurs de types (`unit`, τ `list` et τ `ref`) sont introduits. Les règles de typage MatchList, Get et Set sont standard.

La compilation de ces nouvelles constructions (listes et références) consiste d'une part à ajouter celles-ci à la syntaxe de VSML et d'autre part à introduire des opérateurs supplémentaires en ESML (sans effet de bord) pour manipuler les mots mémoire OCaml (e.g. consulter la taille d'un bloc). Les accès mémoire sont alors encodés en ESML à l'aide de variables de synchronisation réalisant des requêtes séquentielles sur un bus mémoire.

$$
\begin{array}{l}
e ::= \cdots \\
\quad \mid \textbf{match}\ e\ \textbf{with}\ [] \to e_1 \mid x_1 :: x_2 \to e_2 \\
\quad \mid\ !e \\
\quad \mid\ e := e' \\
\tau ::= \cdots \\
\quad \mid \texttt{unit} \\
\quad \mid \tau\ \texttt{list} \\
\quad \mid \tau\ \texttt{ref}
\end{array}
$$

(a) Syntaxe

$$
\textsc{MatchList}\ \frac{\Gamma \vdash e : \tau\ \texttt{list} \qquad \Gamma \vdash e_1 : \tau' \qquad \Gamma[x_1 : \tau, x_2 : \tau\ \texttt{list}] \vdash e_2 : \tau'}{\Gamma \vdash \textbf{match}\ e\ \textbf{with}\ [] \to e_1 \mid x_1 :: x_2 \to e_2 : \tau'}
$$

$$
\textsc{Get}\ \frac{\Gamma \vdash e : \tau\ \texttt{ref}}{\Gamma \vdash\ !e : \tau} \qquad \textsc{Set}\ \frac{\Gamma \vdash e : \tau\ \texttt{ref} \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e := e' : \texttt{unit}}
$$

(b) Typage

Figure 13 – Extension de Macle avec accès mémoire

La figure 14 donne un exemple de circuit Macle calculant la longueur d'une liste OCaml.

```
circuit longueur ℓ =
  let rec aux ℓ acc =
    match ℓ with
      [] → acc
    | _ :: ℓ' → aux ℓ' acc
  in aux ℓ 0
```

Figure 14 – Calcul de la longueur d'une liste en Macle

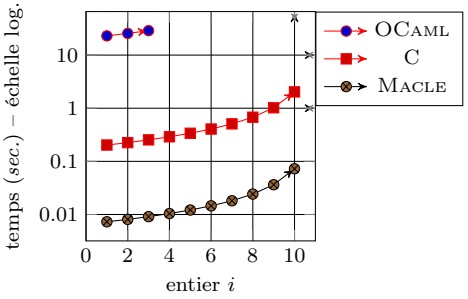
7 Tests de performance

Cette section compare les temps d'exécution de fonctions OCaml, C et Macle incorporées à la bibliothèque d'exécution d'O2B et appelées par des programmes de test écrits en OCaml. Le facteur d'accélération de Macle vis-à-vis d'OCaml et C est mesuré dans un même environnement d'exécution, O2B, compilé par gcc avec optimisation activée (-Os) à travers le backend Nios2. Le code assembleur produit est exécuté par un processeur softcore Nios2 réalisé sur le FPGA Intel Max 10 embarqué sur une carte DE10-Lite de Terasic. Ce modèle comprend 50K cellules logiques et 1638 Kbits de mémoire pour une fréquence d'horloge de 50 MHz. Les programmes OCaml sont compilés par le compilateur ocamlc. Le programme en bytecode engendré est modifié par l'outil ocamlclean. Les circuits Macle sont synthétisés à travers la chaîne de développement Quartus Lite. La taille du tas O2B est de 11756 mots. La taille de la pile O2B est de 512 mots. La taille d'un mot est de 4 octets. Le temps d'exécution d'une fonction externe C (*resp.* OCaml) est mesuré aux extrémités du corps de celle-ci (*resp.* avant et après avoir été appliquée) par la différence de deux appels à la fonction nios_timer_get_us() définie dans O2B. Le temps d'exécution d'un circuit Macle est défini comme le temps d'exécution de la fonction C qui appelle ce circuit.

7.1 Comparaison du temps d'exécution entre OCaml, C et Macle

Nous implémentons en OCaml, C et Macle l'algorithme d'Euclide $\gcd(a, b)$ calculant le plus grand diviseur commun de deux entiers a et b. La fonction OCaml est récursive terminale, la version C est itérative ; la version Macle est donnée figure 12a. Nous comparons le temps

d'exécution de gcd$(i, n!)$ pour i allant de 1 à n avec n fixé à la valeur 10. Sur la figure 15b, Nous observons que le facteur d'accélération de la version MACLE atteint 3000 avec le bytecode OCAML (sous sa forme OMICROB) et 28 avec C, ce qui confirme l'hypothèse d'efficacité des circuits MACLE.



(a)

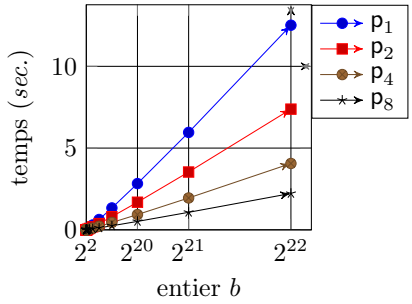
i	ratio C/MACLE	ratio OCAML/MACLE	OCAML (sec.)
1	27.85	3155	23
2	27.81	3164	25.6
3	27.86	3159	28.8
4	27.85		
5	27.84		
6	27,89		
7	28,04		
8	27.96		
9	28.07		
10	27.95		

(b)

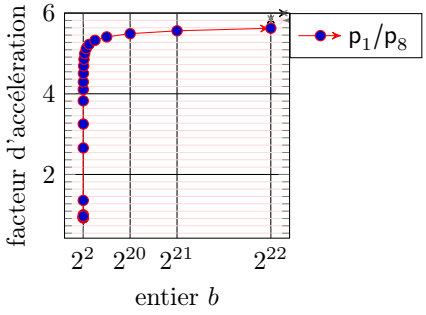
Figure 15 – Temps d'exécution des versions OCAML, C et MACLE du PGCD

7.2 Parallélisme

Pour illustrer le parallélisme des circuits programmés en MACLE, nous considérons le circuit *somme_temps_de_vol4*(a, b) présenté à la figure 12b. Ce circuit comporte une boucle récursive terminale effectuant la somme des expressions (temps_de_vol $(i+k)$) pour k variant de 0 à 3, en traitant les éléments $(i+k)$ en parallèle par paquet de 4. La fonction temps_de_vol est une boucle non bornée qui calcule le *temps de vol* de la suite de Syracuse i. Par exemple, (temps_de_vol 10) vaut 6, (temps_de_vol 11) vaut 14, (temps_de_vol 12) vaut 9. Nous généralisons cette fonction à des paquets de 1, 2, 4 et 8 liaisons locales parallèles. La figure 16a mesure le temps d'exécution de ces quatre versions pour a égal à 1 et b variant de 2^1 à 2^{22}.



(a)



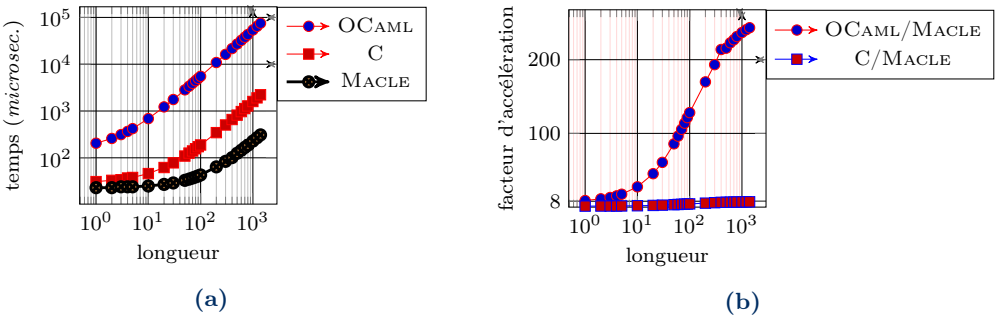
(b)

Figure 16 – Calculs irréguliers exécutés en parallèle par groupes de 1 à 8

La figure 16b met en évidence un facteur d'accélération de 5,6 pour le circuit p_8 vis-à-vis de p_1, soit de 70% par rapport à l'optimum (un facteur d'accélération pour p8 vis-à-vis de p1).

7.3 Calcul de la longueur d'une liste

Nous implémentons une fonction OCaml, une fonction C et un circuit Macle calculant la longueur d'une liste OCaml. La version OCaml est récursive terminale (c'est `List.length`) ; La version C est itérative ; la version Macle est donnée en figure 14. On compare le temps d'exécution de ces deux versions en fonction de la longueur de la liste d'entrée. Nous observons le facteur d'accélération de la version Macle en fonction de la longueur de la liste. La figure 17a donne les valeurs mesurées. La figure 17b donne le facteurs d'accélération de Macle. Celui-ci atteint seulement 7 vis-à-vis de C pour une liste de 1400 éléments, et 270 vis-à-vis d'OCaml. Cela met en évidence un goulet d'étranglement dû au transfert de données. En effet, notre utilisation du bus Avalon ne permet pas de paralléliser les lectures en mémoire.



(a) (b)

Figure 17 – Performance d'un calcul sur des listes en fonction de leurs longueurs

7.4 Synthèse des tests de performance

Ces premiers résultats sont encourageants et confortent l'approche suivie. Sur le calcul du PGCD, les gains, de l'ordre de 30, de Macle vis-à-vis de C semblent optimaux.

Sur le calcul parallèle du temps de vol de la suite de Syracuse, et même si les calculs ne sont pas réguliers, nous obtenons un gain de 70% de la taille des paquets traités en parallèle, soit environ 5,6 pour des paquets de taille 8. Ces deux gains se multiplient si le calcul du PGCD est effectué par paquet en parallèle vis-à-vis du C séquentiel.

Le calcul de la longueur d'une liste du tas OCaml met en évidence un goulet d'étranglement du au transfert de données. Notre utilisation du bus Avalon ne permet pas de paralléliser les lectures en mémoire. Le gain de 7 par rapport à C est moindre qu'escompté, mais non négligeable tout de même. Une meilleure localisation mémoire et un contrôle plus fin de la taille des blocs transférés sur le bus depuis la mémoire devrait permettre d'améliorer les performances.

8 Travaux connexes

Cette section présente les travaux en compilation de langages fonctionnels sur FPGA les plus proches des nôtres.

SAFL (*Statically Allocated Parallel Functional Language*) [10] est un langage strict avec une syntaxe *à la ML*. Il est compilé en Verilog. Les choix de conception du langage suivent deux principes : les sous-expressions indépendantes sont exécutées en parallèle et les circuits

engendrés n'allouent pas de mémoire (il n'y a ni pile, ni tas). Chaque fonction est compilée en un circuit indépendant. Un mécanisme d'exclusion mutuelle (arbitre) est inséré dans le code engendré pour séquentialiser les accès concurrents à une même fonction (e.g. dans $f(1)+f(2)$).

Le compilateur SHARD (*a Scheme to Hardware Compiler*) [12] pour SCHEME cible VHDL. Les sous-expressions indépendantes sont calculées en parallèle. Le langage supporte les entiers, les tableaux globaux, l'allocation de fermetures et la récursion. Il est possible d'encoder des listes avec des fermetures (et ainsi implanter, par exemple, un tri fusion). La gestion mémoire est élémentaire : les fermetures sont désallouées dès qu'elles sont appliquées, ce qui (d'après les auteurs) est source d'erreurs et difficilement utilisable en pratique.

Le langage FLOH (*Functional Language On Hardware*) [16] est une langage non-strict proche du langage *noyau* du compilateur GHC. FLOH est compilé vers SYSTEMVERILOG par une succession de transformations préservant la sémantique. La compilation des appels de fonctions s'inspire du mécanisme de SAFL (arbitres). Des extensions de FLOH supportent la récursion (par introduction de piles d'appels explicites) et le polymorphisme (en monomorphisant) [19].

9 Conclusion et travaux futurs

Cet article a présenté MACLE : langage dédié à la programmation FPGA en style applicatif. Les *circuits* MACLE sont compilés vers des automates synchrones parallèles en VHDL pour reconfigurer un FPGA. L'implémentation O2B de la machine virtuelle OCAML ciblant un processeur *softcore* réalisé sur le même FPGA permet l'appel de ces circuits MACLE (vus comme des fonctions externes C) depuis des programmes OCAML. La reconfiguration du FPGA est entièrement assurée par le compilateur MACLE. Le programmeur a ainsi accès, de façon transparente, à la puissance de calcul du FPGA. MACLE permet par ailleurs le parcours et la modification physique de valeurs allouées dans le tas OCAML, vu comme une mémoire partagée entre les circuits et le *softcore*.

Les premiers résultats obtenus sont encourageants. Nous observons des facteurs d'accélérations importants (de l'ordre de 30) des circuits MACLE vis-à-vis de fonctions C exécutées sur le *softcore*. Par contre, les accès au tas OCAML depuis les circuits MACLE – via des requêtes sur le bus assurant la communication entre le *softcore* et le reste du FPGA – sont moins efficaces (7 fois plus rapide que C). En exploitant le parallélisme présent implicitement dans les circuits (e.g. liaisons locales, applications de fonctions et d'opérateurs), MACLE peut cependant accélérer significativement certains calculs, réguliers ou irréguliers.

MACLE et O2B sont des logiciels libres [13] disponibles aux adresses suivantes :

https://github.com/lsylvestre/macle
https://github.com/jserot/O2B

Travaux futurs L'approche portable d'O2B et de MACLE devrait faciliter leur portage sur d'autres familles de FPGA ainsi que la construction d'un mode simulateur enrichi pour la mise au point des programmes. L'introduction de squelettes de parallélisme sur des structures de données allouées, en particuliers les vecteurs, aidera à l'écriture d'algorithmes parallèles tout en optimisant l'accès mémoire par paquets. L'ajout d'un mécanisme de gestion d'erreurs et d'un allocateur de mémoire devrait aussi permettre d'implémenter en MACLE une machine virtuelle OCAML simplifiée.

13. Les travaux sur MACLE et O2B sont soutenus par l'Initiative de Recherche et Innovation sur le Logiciel Libre (IRILL).

Références

[1] C. Baaij and J. Kuper. Using Rewriting to Synthesize Functional Languages to Digital Circuits. In *International Symposium on Trends in Functional Programming*, pages 17–33. Springer, 2013.

[2] P. Bjesse, K. Claessen, M. Sheeran, and S. Singh. Lava : hardware design in Haskell. *ACM SIGPLAN Notices*, 34(1) :174–184, 1998.

[3] J. Choi, M. Vijayaraghavan, B. Sherman, and A. Chlipala. Kami : A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP) :1–30, 2017.

[4] J.-L. Colaço, G. Hamon, and M. Pouzet. Mixing Signals and Modes in Synchronous Data-flow Systems. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded Software*, pages 73–82, 2006.

[5] T. S. Czajkowski, U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh. From OpenCL to high-performance hardware on FPGAs. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 531–534. IEEE, 2012.

[6] P. Gammie. Synchronous Digital Circuits as Functional Programs. *ACM Computing Surveys (CSUR)*, 46(2) :1–27, 2013.

[7] The HardCaml OCaml Library. https://github.com/janestreet/hardcaml.

[8] F. Maraninchi and Y. Rémond. Mode-automata : About Modes and States for Reactive Systems. In *European Symposium On Programming*, pages 185–199. Springer, 1998.

[9] J. Matthews, B. Cook, and J. Launchbury. Microprocessor specification in hawk. In *Proceedings of the 1998 International Conference on Computer Languages (Cat. No. 98CB36225)*, pages 90–101. IEEE, 1998.

[10] A. Mycroft and R. Sharp. A Statically Allocated Parallel Functional Language. In *International Colloquium on Automata, Languages, and Programming*, pages 37–48. Springer, 2000.

[11] S. Peyton Jones, W. Partain, and A. Santos. Let-floating : moving bindings to give faster programs. In *Proceedings of the first ACM SIGPLAN International Conference on Functional programming*, pages 1–12, 1996.

[12] X. Saint-Mleux, M. Feeley, and J.-P. David. SHard : a Scheme to Hardware Compiler. In *Workshop on Scheme and Functional Programming*, 2006.

[13] J. Sérot. *La programmation des circuits FPGA et le langage VHDL. Une introduction pour les programmeurs et par l'exemple.* Ellipse, 2019.

[14] J. Sérot and E. Chailloux. OCaml sur circuit FPGA. *32 ème Journées Francophones des Langages Applicatifs*, pages 72–74, 2021.

[15] J. Sérot and G. Michaelson. Compiling Hume down to gates. In *Draft Proceedings of 11th International Symposium on Trends in Functional Programming*, pages 191–226, 2011.

[16] R. Townsend, M. A. Kim, and S. A. Edwards. From Functional Programs to Pipelined Dataflow Circuits. In *Proceedings of the 26th International Conference on Compiler Construction*, pages 76–86, 2017.

[17] S. Varoumas, B. Vaugon, and E. Chailloux. A Generic Virtual Machine Approach for Programming Microcontrollers : the OMicroB Project. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, Toulouse, France, Jan. 2018.

[18] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and computation*, 115(1) :38–94, 1994.

[19] K. Zhai, R. Townsend, L. Lairmore, M. A. Kim, and S. A. Edwards. Hardware Synthesis from a Recursive Functional Language. In *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*, pages 83–93. IEEE, 2015.

Déboîter les constructeurs

Nicolas Chataing[1,2], Camille Noûs[3], and Gabriel Scherer[4]

[1] ENS
[2] Lexifi
[3] Laboratoire Cogitamus
[4] INRIA

Résumé

Nous proposons une implémentation d'une nouvelle fonctionnalité pour OCaml, le déboîtement [1] de constructeurs. Elle permet d'éliminer certains constructeurs de la représentation dynamique des valeurs quand cela ne crée pas de confusion entre différentes valeurs au même type. Nous décrivons :

— un cas d'usage sur les grands entiers où la fonctionnalité améliore les performances de code OCaml idiomatique, éliminant le besoin d'écrire du code non-sûr.
— l'analyse statique nécessaire pour accepter ou rejeter le déboîtement d'un constructeur,
— et l'impact sur la compilation du filtrage par motif.

Pour notre analyse statique, nous devons normaliser certaines expressions de type, avec une relation de normalisation qui ne termine pas nécessairement en présence de types mutuellement récursifs ; nous décrivons une analyse dynamique de terminaison qui garantit la normalisation sans rejeter les définitions de types qui nous intéressent.

1 Introduction

OCaml, ainsi que les autres langages de la famille ML, permet de définir des types qui sont des synonymes, ou abréviations, de types existants, et aussi de nouveaux types de données (sommes ou enregistrements) qui sont distincts de tous les types déjà existants.

```
type an_abbrev = int list
type a_datatype = Short of int | Long of int list
```

Dans l'implémentation de référence du langage, la représentation dynamique (à l'exécution) d'un type somme est la suivante :

— Si elle est constituée d'un constructeur *constant* (sans paramètre), comme `[]` ou `None`, c'est une valeur dite *immédiate* représentée par un entier (`0` pour le premier constructeur constant dans la définition de type, `1` pour le deuxième, etc.).
— Si elle est constituée d'un constructeur *non-constant* (avec un ou plusieurs paramètres), comme `Short n`, elle est représentée par (un pointeur vers) un *bloc* mémoire, débutant avec un *en-tête* suivi par la représentation de chaque paramètre du constructeur. L'en-tête contient une étiquette (tag), le numéro du constructeur (dans l'ordre de définition), et une *arité*, le nombre de paramètres du bloc,

Le filtrage par motif permet de faire une distinction de cas sur les constructeurs d'un type somme, comme dans les deux filtrages suivants :

```
| Some v -> ...                    | Short n -> ...
| None -> ...                      | Long li -> ...
```

Le filtrage peut être exécuté efficacement, car un test rapide permet de distinguer des constructeurs différents : deux constructeurs constants sont des entiers différents, deux constructeurs non-constants portent des étiquettes différentes, et OCaml utilise un bit pour distinguer

1. Dans cet article, nous marquons par une couleur particulière les premières apparitions des mots français rigolos exprimant un concept habituellement décrit en anglais, comme le déboîtement (unboxing). Quand le mot n'est pas évident, nous donnons sa traduction d'outre-Manche au premier usage.

les valeurs immédiates (en particulier les constructeurs constants) des blocs (en particulier les constructeurs non-constants).

1.1 Déboîter les types mono-constructeurs

Depuis OCaml 4.06 (juin 2016), OCaml respecte une annotation de déboîtement sur les types sommes contenant un seul cas, dont le constructeur a un seul paramètre (ou sur les enregistrements ayant un seul champ) :

```
type type_name = Type of string [@@unboxed]
type 'a exi = { run: 'b . ('a -> 'b) -> 'b } [@@unboxed]
```

Cette annotation a pour effet que le constructeur (ou l'enregistrement) est effacé de la représentation de la valeur à l'exécution. Par exemple `Type "list"` est représenté exactement comme `"list"`, une valeur de type `string`, au lieu d'être un bloc contenant un paramètre de type `string`. On gagne un pouillème au moment de créer et d'inspecter ces valeurs, et surtout on a une représentation mémoire un peu plus compacte qui peut avoir un impact positif sur les performances sur de gros jeux de données.

En pratique les programmes où cette annotation fait une vraie différence pour les performances sont rares : l'allocation d'un nouveau bloc est très rapide et les programmes bénéficient souvent d'une bonne localité en mémoire, rendant la lecture rapide aussi. L'annotation a surtout l'intérêt de rassurer les programmeurs et programmeuses, en leur donnant la garantie que les nouvelles définitions introduites, pour avoir un type séparé d'un type existant plutôt qu'un synonyme, n'introduisent aucun surcoût.

1.2 Déboîter plus de constructeurs ?

Jeremy Yallop a proposé (Yallop, 2020) d'étendre le déboîtement à certains constructeurs de types sommes ayant plusieurs constructeurs.

```
type a_big_number =
  | Short of int [@unboxed]
  | Long of int list
```

L'annotation `[@@unboxed]` précédente portait sur toute la déclaration, mais ici l'annotation `[@unboxed]` ne porte que sur le constructeur `Short` ; on demande à ce que ce constructeur ne soit pas représenté en mémoire, que `Short n` soit représenté exactement comme l'entier `n`. Cette demande sera acceptée après que le typeur aura vérifié qu'il n'y a pas de confusion possible avec les autres valeurs de ce type, de la forme `Long li`, dont la représentation est toujours distinguable de celle des entiers. Dans le cas général on peut demander à déboîter plusieurs constructeurs du même type, tant que cela n'introduit toujours pas de doublons, deux valeurs différentes ayant la même représentation.

Il faut parfois rejeter cette annotation, car retirer le constructeur introduirait des doublons :

```
type clash_between_int_and_int =
| Int of int [@unboxed]
| Also_int of int [@unboxed]
Error: This declaration is invalid, some [@unboxed] annotations introduce
       overlapping representations.

type t = Constant_constructor_0
type clash_between_constant_constructors =
| T of t [@unboxed]
| Another_constant_constructor_0
Error: This declaration is invalid, some [@unboxed] annotations introduce
```

`overlapping representations.`

Dit autrement : les types *sommes* des langages ML représentent des unions *disjointes*, et le filtrage par motif repose sur le fait de pouvoir distinguer (rapidement) dans quel cas de l'union on se trouve. Une stratégie d'implémentation est d'utiliser des *blocs* et leurs étiquettes pour tous les constructeurs ; cela impose la séparation (disjointness) par construction. Mais dans certains cas, la représentation des valeurs des paramètres porte déjà assez d'information pour les distinguer, et la séparation peut être obtenue sans représenter explicitement les constructeurs.

1.3 Notre approche

— Nous définissons une notion de *tête* d'une valeur qui est une approximation simple de la valeur, efficace à calculer dynamiquement.
— Nous approximons un type par l'ensemble des têtes de leurs valeurs. La représentation des têtes est choisie pour que ces ensembles soient représentables de façon compacte et efficace.
— Nous calculons les ensembles de têtes correspondant à chaque constructeur d'un type de donnée (déboîté ou non), et rejetons la définition si les ensemble des têtes des différents constructeurs ne sont pas disjoints.

Cette approche garantit par construction que les doublons sont rejetés statiquement : si les têtes sont disjointes, les valeurs sont nécessairement disjointes.

Par ailleurs, avec cette approche les têtes suffisent à distinguer les valeurs ; comme elles sont calculables efficacement pendant l'exécution du programme, elles peuvent être utilisées pour exécuter le filtrage par motif.

Dans notre implémentation, la tête d'une valeur est un élément de $\{\texttt{Imm}, \texttt{Block}\} \times \mathbb{Z}_{\texttt{int}}$ (où $\mathbb{Z}_{\texttt{int}}$ est l'ensemble des entiers OCaml de type `int`) défini ainsi :

— La tête d'une valeur immédiate n est la paire $(\texttt{Imm}, n)$
— La tête d'un bloc d'étiquette t est la paire $(\texttt{Block}, t)$.

D'autres choix de têtes sont possibles ; l'approche s'adapte à d'autres langages de programmation ou d'autres implémentations en changeant ce choix. Même au sein d'un langage et d'une représentation dynamique des valeurs, déterminée par une implémentation, on peut choisir une notion de tête plus ou moins fine, qui garde plus ou moins d'information sur la valeur. Une définition moins fine risque de rejeter plus des définitions de types de données, quand les têtes ne sont pas disjointes alors que les valeurs auraient pu l'être. Une définition plus fine permet d'accepter plus de déboîtements de constructeurs, au risque d'être plus coûteuse à calculer pendant le filtrage.

1.3.1 Contribution

Notre contribution, présentée dans cet article, consiste en une implémentation des grandes lignes de la proposition de Jeremy Yallop, dans une version expérimentale du compilateur OCaml [2]. Nous expliquons :

— Une étude de cas où le déboîtement permet un gain de performance, à code égal, ou un gain de propreté et robustesse du code à performances égales.
— Comment décider si une définition doit être rejetée car elle introduit des confusions de représentations.
— En particulier, comment « déplier » des définitions de type pour calculer la représentation des valeurs d'un type donné, en présence de définitions mutuellement récursives qui pour-

2. Nous reprenons les grandes lignes de la spécification de Jeremy Yallop, mais l'avons modifiée pour la rendre plus simple et plus prédictible. Dans cet article nous discutons seulement de notre version, voir Chataing and Scherer (2021) pour une comparaison.

raient nous faire « boucler ».

Les sujets supplémentaires suivants sont traités en annexe :

— Comment adapter le compilateur de filtrage par motif aux de constructeurs déboîtés.
— Des questions sur le support d'autres implémentations utilisant d'autres représentations.
— Un ensemble d'extensions intéressantes qui restent à étudier.

Nous ne traiterons pas dans cet article des questions spécifiques à l'implémentation de ce travail dans le compilateur OCaml.

2 Étude de cas : les grands entiers

Cette nouvelle fonctionnalité est motivée par des considérations de performances. Pour justifier ce travail, voici une étude de cas où elle est utile pour obtenir de bonnes performances.

La bibliothèque Zarith (Miné and Leroy, 2012) fournit une implémentation efficace d'entiers de taille arbitraire, se reposant sur la bibliothèque `Gmp` (Granlund and contributors, 1991) qui fait référence dans ce domaine.

Certains utilisateurs de grands entiers font la majorité de leur calcul sur des entiers plus grands que le type `int` habituel – le type des « petits » entiers. Mais, à l'inverse, une quantité d'utilisateurs font des calculs avec des valeurs que l'on peut presque toujours représenter avec un `int` (sur une architecture avec des mots de 64 bits), mais veulent un résultat précis et mathématiquement exact même dans les cas, rares, de dépassement. Pour cet usage courant, on souhaite minimiser le surcoût par rapport à une implémentation utilisant directement les `int`.

Pour minimiser le surcoût de la précision arbitraire, il est essentiel que le cas courant d'opérations sur des petits entiers ne demande pas d'allocation mémoire, ni d'appel à une fonction C non-triviale (pouvant allouer ou échouer). On veut que la bibliothèque propose un raccourci (fast path) pour les petits entiers.

Zarith implémente des grands entiers avec un type `Zarith.t` qui contient soit des entiers OCaml (comme au type `int`), soit une valeur étrangère (bloc de tag `Custom_tag`). Il n'est pour l'instant pas possible de définir un type ayant cette rerprésentation en OCaml ; l'implémentation de Zarith doit tricher en utilisant les fonctions bas-niveau et non-sûres du module `Obj`, abandonnant la sûreté mémoire ordinairement garantie par typage [3].

```
type t (* int or gmp integer (in a custom block) *)
external is_small_int: t -> bool = "%obj_is_int"
external unsafe_to_int: t -> int = "%identity"
external of_int: int -> t = "%identity"

external c_add: t -> t -> t = "ml_z_add"
let add x y =
  if is_small_int x && is_small_int y then begin
    let z = unsafe_to_int x + unsafe_to_int y in
    (* Overflow check -- Hacker's Delight, section 2.12 *)
    if (z lxor unsafe_to_int x) land (z lxor unsafe_to_int y) >= 0
    then of_int z
    else c_add x y
  end else
    c_add x y
```

3. Une version précédente de Zarith utilisait du code assembleur pour implémenter le raccourci, ce qui permet d'utiliser le drapeau de dépassement présent sur certaines architectures pour détecter efficacement le dépassement. Mais appeler du code assembleur par le mécanisme d'IFE (FFI) OCaml a un coût qui rendait en fait cette version plus lente que la version OCaml actuelle.

Avec notre travail sur les constructeurs déboîtés, il est possible d'écrire le code ainsi :

```
type custom_gmp_t
type t =
  | Short of int [@unboxed]
  | Long of custom_gmp_t
let of_int n = Short n

external c_add: t -> t -> t = "ml_z_add_boxcustom"
let add a b =
  match a, b with
  | Short x, Short y ->
      let z = x + y in
      (* Overflow check -- Hacker's Delight, section 2.12 *)
      if (z lxor x) land (z lxor y) >= 0
      then Short z
      else c_add a b
  | _, _ -> c_add a b
```

Ce code est équivalent à la version précédente quand on prend le raccourci (il génère exactement le même code machine sur cette partie), mais s'écrit en OCaml sûr, sans utiliser `Obj`.

Dans le cas du long chemin (slow path), il reste une différence : cette représentation emboîte les valeurs de type `custom_gmp_t` dans le constructeur `Long`, alors que la version non-sûre les représente directement, sans doublons – ce sont des blocs de tag `Custom_tag`, jamais des entiers.

Notre implémentation des annotations de déboîtement ne permet pas pour l'instant de déboîter le constructeur `Long`, car son argument `custom_gmp_t` est un type abstrait pour le typeur, dont nous supposons qu'il pourrait contenir n'importe quelle valeur. Il est seulement peuplé par le code C, qui y met toujours des valeurs étrangères (tag `Custom`). Nous avons prévu d'étendre à terme notre implémentation pour permettre à l'utilisateur d'annoter ces types abstraits avec des informations (non vérifiables) sur leur structure étrangère, pour permettre de déboîter les deux constructeurs de ce type.

Nous avons fait l'inévitable test de performance sur deux micro-bancs-de-test, d'abord un banc *sans dépassements* qui ne manipule que des petits entiers, et ensuite un banc *avec dépassements* qui construit des entiers `Gmp` pour 16% de ses opérations. (Attention : les deux bancs, avec ou sans dépassements, font des calculs différents, donc la comparaison de leur performance n'a pas de sens.)

version	sans dépassements	16% de dépassements
`int`	6.3s	8.3s (et faux)
`Obj`	1.24x plus lent	2.17x plus lent
`unboxed`	1.24x plus lent	2.38x plus lent
`boxed`	1.56x plus lent	2.41x plus lent

La ligne `int` indique les performances d'une version du code qui n'utilise que des petits entiers, sans aucun test pour détecter les dépassements. Dans le banc de test avec dépassements, elle renvoie un résultat erroné !

La ligne `Obj` indique les performances de la version actuelle de Zarith, écrite avec du code non-sûr. Le surcoût lié aux tests de dépassement est étonnamment faible, 1.24x sans dépassements. Le surcoût de 2.17x dans le banc avec dépassements ne correspond pas aux tests de dépassements, mais au coût des opérations de la bibliothèque `gmp`.

La ligne `boxed` indique les performances d'une version emboitée du code, écrite directement avec un type somme usuel, `type t = Short of int | Long of Gmp.t`.

La ligne `unboxed` indique les performances de notre solution. Sans dépassements, nous sommes dans le régime où notre représentation est identique à celle de Zarith, et les performances sont identiques : l'annotation de déboîtement égale les performances du code non-sûr. En présence de dépassements, notre représentation emboîte les « longs » entiers, et les performances sont proches de la version `boxed`. Une implémentation plus complète que la nôtre, permettant d'annoter les types abstraits avec des informations externes, éliminerait cette dernière différence avec la version `Obj` ; nous pourrions alors recoder Zarith pour utiliser cette fonctionnalité.

Toutes ces mesures ont été effectuées avec le compilateur natif, `ocamlopt`.

Conclusion de l'étude de cas Notre implémentation du déboîtement de constructeurs accélère sensiblement la définition « usuelle » en OCaml, permettant d'égaler les performances du code non-sûr de Zarith pour les programmes manipulant uniquement des petits entiers. Le déboîtement du constructeur de grands entiers est conceptuellement possible, mais n'est pas encore géré par notre prototype.

3 Têtes et formes de tête

Nous devons rejeter les annotations de déboîtement qui introduiraient des doublons, deux valeurs distinctes d'un type se retrouvant avec la même représentation dynamique.

Comme expliqué en introduction (§1.3), nous approximons les valeurs d'un type par leur *tête*, choisie pour être facile à calculer dynamiquement, et pouvoir raisonner statiquement sur les ensembles des têtes correspondant à chaque type.

Dans cette section, nous donnons des têtes une description abstraite, de haut niveau, qui ne dépend pas du langage de programmation – notre traitement s'étend à tous les langages de la famille ML – et des détails d'implémentation.

3.1 Calculer la tête d'un type

Dans cette description de haut niveau, une tête h est soit un constructeur de type somme C of τ, soit un « nom de type primitif » $\widehat{\texttt{t}}$ (par exemple $\widehat{\texttt{int}}$, $\widehat{\texttt{array}}$, $\widehat{\texttt{tuple}}$, $\widehat{\texttt{function}}$), soit $\top$, qui représente n'importe quelle valeur. Une forme de tête (head shape) représente un multi-ensemble de têtes par une liste finie de têtes (notée comme une somme $h_1 + h_2 + \cdots + \emptyset$; on manipule $+$ comme un symbole associatif et on note h pour la somme $h + \emptyset$).

$$h ::= C \text{ of } \tau \mid \widehat{\texttt{t}} \mid \top \qquad\qquad H ::= \emptyset \mid h + H$$

Définissons une petite grammaire d'expressions de types et de définitions de types de données :

$$\begin{array}{lcl} \tau & ::= & \alpha \mid (\tau_i)_i\ t \\ d & ::= & \text{type } (\alpha_i)_i\ t\ =\ (C_j \text{ of } \tau_j)_j\ (C_k^{\text{unboxed}} \text{ of } \tau'_k)_k \end{array}$$

Chaque définition de type somme vient avec une famille (éventuellement vide) de constructeurs emboîtés, et une famille (éventuellement vide) de constructeurs déboîtés.

Notation $((x_i)_{i\in I}, (x_i)_i)$**.** Nous notons $(\tau_i)_{i\in I}$ une famille indicée sur $i \in I$, et souvent $(\tau_i)_i$ quand l'ensemble d'indices n'a pas d'importance.

Nous définissons un jugement $\tau \Rightarrow H$ qui associe une forme de tête H à un type τ. Notre jugement est paramétré par une relation $\widehat{\texttt{T}}$ entre constructeurs de types t et nos têtes primitives $\widehat{\texttt{t}}$. Par exemple on aura typiquement $(\texttt{int}, \widehat{\texttt{int}}) \in \widehat{\texttt{T}}$.

$$\textsc{Var}\ \frac{}{\alpha \Rightarrow \top} \qquad \textsc{Prim}\ \frac{(t, \widehat{\mathtt{t}}) \in \widehat{\mathtt{T}}}{(\tau_i)_i\ t \Rightarrow \widehat{\mathtt{t}}}$$

$$\textsc{Type}\ \frac{\mathsf{type}\ (\alpha_i)_i\ t\ =\ (C_j\ \mathsf{of}\ \_)_j\ (C_k^{\mathsf{unboxed}}\ \mathsf{of}\ \tau'_k)_k \qquad \forall k,\quad \tau'_k[(\alpha_i)_i \leftarrow (\tau_i)_i] \Rightarrow H_k}{(\tau_i)_i\ t \Rightarrow \sum_j C_j + \sum_k H_k}$$

Quand nous calculons la forme de tête d'un type $(\alpha_i)_i\ t$, les variables α_i pourront ensuite être instantiées avec n'importe quel type ; la règle Var leur donne donc la forme $\top$ de toutes les valeurs possibles.

La règle Prim donne leur forme aux types primitifs connus de la relation $\widehat{\mathtt{T}}$.

La règle Type calcule la forme de tête d'un type de donnée, dont les paramètres sont instanciés par des types concrets $(\tau_i)_i$. Elle dépend de la définition de $(\alpha_i)_i\ t$ dans l'environnement de typage, contenant une famille de constructeurs emboîtés C_j et une famille de constructeurs déboîtés C_k^{unboxed}. La forme du type est obtenue en ajoutant aux constructeurs emboîtés C_j l'union des formes des paramètres de type $(\tau'_i)_i$ des constructeurs déboîtés, après avoir remplacés les paramètres α_i par leur instance τ_i.

Considérons par exemple les définitions suivantes :

```
type 'a id = Id of 'a [@unboxed]
type 'a proc =
  | Base of int id [@unboxed]
  | Ask of ('a -> 'a proc) [@unboxed]
  | Tell of 'a * 'a proc
```

Avec une relation $\widehat{\mathtt{T}}$ bien choisie, la forme de tête du type `bool proc` est $\mathtt{Tell} + \widehat{\mathtt{int}} + \widehat{\mathtt{function}}$, comme le justifie la dérivation suivante :

$$\dfrac{\dfrac{\mathsf{type}\ \alpha\ \mathsf{id}\ =\ \mathsf{Id}^{\mathsf{unboxed}}\ \mathsf{of}\ \alpha}{\mathsf{int\ id} \Rightarrow \widehat{\mathtt{int}}} \qquad \dfrac{\dfrac{}{(\mathsf{int}, \widehat{\mathtt{int}}) \in \widehat{\mathtt{T}}}}{\mathsf{int} \Rightarrow \widehat{\mathtt{int}}} \qquad \dfrac{\dfrac{}{((\mathsf{bool} \to \mathsf{bool\ proc}), \widehat{\mathtt{function}}) \in \widehat{\mathtt{T}}}}{(\mathsf{bool} \to \mathsf{bool\ proc}) \Rightarrow \widehat{\mathtt{function}}}}{\mathsf{bool\ proc} \Rightarrow (\mathtt{Tell}\ \mathsf{of}\ \mathsf{bool} * \mathsf{bool\ proc}) + \widehat{\mathtt{int}} + \widehat{\mathtt{function}}}$$

Hypothèse. Notre jugement dépend d'un choix de type primitifs $\widehat{\mathtt{T}}$ et d'un environnement global de déclarations de types de données. Nous faisons de plus l'hypothèse que les types couverts par ces deux mécanismes sont disjoints : si $(\alpha_i)_i\ t$ est déclaré dans l'environnement, alors aucun $(\tau_i)_i\ t$ n'appartient à $\widehat{\mathtt{T}}$, et réciproquement.

Un type qui n'est ni déclaré ni primitif est traité comme inconnu, il n'y a pas de jugement correspondant. Nous pourrions facilement étendre ce modèle minimal, pour gérer par exemple les abbréviations (avec une règle anologue à Type) et les types abstraits, à qui nous donnons la forme de tête $\top$ – nous parlons des types abstraits plus en détail en Annexe C.2.

3.2 Digression formelle : sémantique des têtes

Pour donner une sémantique précise à ces formes de têtes, il faut inévitablement parler un peu plus des représentations de bas niveau des valeurs.

Hypothèse. Nous supposons donnés un ensemble Data de valeurs de bas niveau, et une *fonction de représentation* $\mathsf{repr} : \mathsf{Value} \to \mathsf{Data}$ ayant la propriété suivante : si $C^{\mathsf{unboxed}}(v)$ est la valeur formée par l'application d'un constructeur déboîté à un argument v, alors on doit avoir

$$\mathsf{repr}(C^{\mathsf{unboxed}}(v)) = \mathsf{repr}(v)$$

Les constructeurs déboîtés sont invisibles dans la représentation des valeurs.

Notation ($\mathcal{M}(S)$, $\{\!\{\ldots\}\!\}$, $\lfloor M \rfloor_{\mathsf{set}}$, $M_1 + M_2$)**.** Nous notons $\mathcal{M}(S)$ l'ensemble des multi-ensembles d'éléments de S, $\{\!\{\ldots\}\!\}$ pour décrire un multi-ensemble plutôt qu'un ensemble, et $\lfloor M \rfloor_{\mathsf{set}}$ l'ensemble des éléments d'un multi-ensemble M : si $M \in \mathcal{M}(S)$, alors $\lfloor M \rfloor_{\mathsf{set}} \in \mathcal{P}(S)$. Nous notons $M_1 + M_2$ pour l'union de deux multi-ensembles.

Définition ($v : \tau$)**.** Nous notons $v : \tau$ quand v est une valeur de type τ. Nous ne spécifions pas le typage des valeurs dans ce document, à part les règles suivantes pour les types de données :

$$\frac{\mathsf{type}\ (\alpha_i)_i\ t\ =\ (C_j\ \mathsf{of}\ \tau_j\ )_j\ (C_k^{\mathsf{unboxed}}\ \mathsf{of}\ \tau'_k)_k \qquad v : \tau_j[(\alpha_i \leftarrow \tau''_i)_i]}{C_j\ v : (\tau''_i)_i\ t}$$

$$\frac{\mathsf{type}\ (\alpha_i)_i\ t\ =\ (C_j\ \mathsf{of}\ \tau_j\ )_j\ (C_k^{\mathsf{unboxed}}\ \mathsf{of}\ \tau'_k)_k \qquad v : \tau'_k[(\alpha_i \leftarrow \tau''_i)_i]}{C_k^{\mathsf{unboxed}}\ v : (\tau''_i)_i\ t}$$

Définition ($\mathsf{repr}(\tau)$)**.** Nous construisons à partir de notre fonction $\mathsf{repr} : \mathsf{Value} \to \mathsf{Data}$ une fonction $\mathsf{repr} : \mathsf{Type} \to \mathcal{M}(\mathsf{Data})$, qui associe à un type le multi-ensemble des représentations de ses valeurs :

$$\mathsf{repr}(\tau) := \{\!\{\mathsf{repr}(v) \mid v : \tau\}\!\}$$

Hypothèse ($\mathsf{repr}(\widehat{\mathsf{t}})$)**.** On suppose une famille de représentations $\mathsf{repr}(\widehat{\mathsf{t}}) \in \mathcal{P}(\mathsf{Data})$, telle que

$$\forall (v : \tau), \qquad (\tau, \widehat{\mathsf{t}}) \in \widehat{\mathsf{T}} \qquad \Longrightarrow \qquad \mathsf{repr}(v) \in \mathsf{repr}(\widehat{\mathsf{t}})$$

Définition ($\mathsf{repr}(h)$, $\mathsf{repr}(H)$)**.** En plus des représentations primitives $\mathsf{repr}(\widehat{\mathsf{t}})$, nous définissons des représentations $\mathsf{repr}(h), \mathsf{repr}(H) \in \mathcal{M}(\mathsf{Data})$ des têtes et des formes de tête :

$$\begin{array}{lcllcl} \mathsf{repr}(C\ \mathsf{of}\ \tau) & := & \{\!\{\mathsf{repr}(C(v)) \mid v : \tau\}\!\} & \mathsf{repr}(\emptyset) & := & \emptyset \\ \mathsf{repr}(\top) & := & \mathsf{Data} & \mathsf{repr}(h + H) & := & \mathsf{repr}(h) + \mathsf{repr}(H) \end{array}$$

Définition (Sans doublons)**.** On dit d'un type τ (ou d'une forme de tête H) qu'il (elle) est *sans doublons* si le multi-ensemble $\mathsf{repr}(\tau)$ (ou $\mathsf{repr}(H)$) n'a aucun élément répété plusieurs fois.

Deux types ou formes de têtes sont *disjointes* si l'intersection de leur représentation est vide, autrement dit si leur union est sans doublons.

Lemme 1. Si le type τ a la forme de tête H, la représentation de H sur-approxime celle de τ :

$$\tau \Rightarrow H \qquad \Longrightarrow \qquad \mathsf{repr}(\tau) \subseteq \mathsf{repr}(H)$$

En particulier, si H est sans doublons, alors τ est aussi sans doublons.

3.3 Accepter ou rejeter une définition de type

Pour décider si la définition d'un type de donnée $(\alpha_i)_i\ t$ peut introduire des doublons, on calcule la forme de tête de $(\alpha_i)_i\ t$:

$$\textsc{Typedecl}\ \frac{(\alpha_i)_i\ t \Rightarrow R}{(\mathsf{type}\ (\alpha_i)_i\ t\ =\ \ldots) \Rightarrow R}$$

et on regarde si cette forme contient des conflits : on rejette la déclaration si la forme de tête H contient des têtes dont les représentations de bas niveau ne sont pas disjointes, c'est-à-dire si $\mathsf{repr}(H)$ est un multi-ensemble contenant des doublons.

Remarquons que, dans ce test, l'expression de type $(\alpha_i)_i\ t$ contient des variables libres α_i. Si ces variables interviennent pendant le calcul de la forme de tête, on obtient $\top$ dans la forme de tête calculée. Ce n'était pas le cas dans notre exemple bool proc, qui ne contient pas de constructeur $C^{\mathsf{unboxed}}\ \mathsf{of}\ \alpha_i$.

À l'inverse, quand calcule la forme de tête d'un type, on rencontre des expressions de type comme int id dans notre exemple où les paramètres ont une instance concrète. On calcule donc une forme de tête pour ce type ($\widehat{\texttt{int}}$) qui est plus fine que la forme de tête $\top$ du cas général $\alpha\,\mathsf{id}$. Prendre ainsi en compte les paramètres de type permet d'accepter plus de définitions : si nous approximions la forme de tête de int id par celle de α id, c'est-à-dire $\top$, la définition de α proc ci-dessus serait rejetée par notre critère, puisque sa forme de tête $(\texttt{Tell of} \ldots) + \top + \widehat{\texttt{function}}$ contient des doublons : l'ensemble de valeurs correspondant à $\top$ a une intersection non vide avec celui de `Tell of` ... et celui de $\widehat{\texttt{function}}$.

Cette approche n'est pas modulaire, elle repose sur la disponibilité et le dépliage systématique des définitions de type. Cela ne pose pas de problème de performances car le nombre de définitions de types reste petit en pratique ; nous discutons modularité en Annexe C.3.

4 Contrôle dynamique de la terminaison

4.1 Intuition

Le jugement $\tau \Rightarrow H$ donne une spécification du calcul de la forme de tête, mais pas un algorithme effectif pour la calculer, à cause des problèmes de terminaison en présence de types de données mutuellement récursifs. Considérons par exemple la définition

```
type 'a t = Foo | Loop of 'a t [@unboxed]
```

Calculer naïvement la forme de tête de `int t` (par exemple) conduirait à une boucle infinie, puisque la règle Type suggère de calculer la forme de tête du constructeur déboîté `Loop`, à savoir `int t` de nouveau.

Dit autrement, notre algorithme revient à calculer la forme normale d'expression de types pour une relation d'expansion / de récriture dépliant certaines définitions de type. Il n'existe pas toujours de forme normale, ce processus peut ne pas terminer dans le cas de définitions de types mutuellement récursives.

Interdire statiquement les cycles, comme pour les abréviations ? Dans le cas des définitions d'abréviations/synonymes (les définitions de la forme $\mathsf{type}\ (\alpha_i)_i\ t = \tau$, qui donnent un nouveau nom à un type existant au lieu de définir un nouveau type de donnée), cette situation n'apparaît pas car OCaml interdit déjà les abréviations cycliques. Dans un groupe de types mutuellement récursifs, contenant des abréviations et des types de donnée, tout cycle dans la relation « la définition de `t` mentionne le constructeur de type `u` » doit être « cassé » par au moins une définition de type de données, sinon le groupe de définitions est rejeté. Mais nous ne pouvons pas traiter les constructeurs déboîtés comme les abréviations, car cela voudrait dire qu'ajouter une annotation `[@unboxed]` pourrait transformer un cycle autorisé en un cycle interdit dans de nombreux cas utiles. Par exemple :

```
type 'a thunk = unit -> 'a
type 'a stream =
| Next of ('a * 'a stream) thunk [@unboxed]
| End
```

Cette définition décrit un type de flots (stream) paresseux, qui utilise une abréviation auxiliaire `'a thunk`. Le type `'a stream` est récursif, il apparaît dans le type du constructeur `Next`. Sans annotation de déboîtement, ces définitions sont parfaitement valides. Une approche qui rejette

tous les cycles sauf ceux qui passent par au moins constructeur emboîté n'est pas satisfaisante.

Détecter la répétition d'un type pendant l'expansion ? Une idée naturelle est de garder trace, pendant le calcul de la forme de tête d'un type, des expressions de type dont on est en train de calculer la forme de tête. Si une règle demande de calculer de nouveau l'un des types déjà présents dans cette trace (dans notre exemple : calculer la forme de `int t` pour calculer celle de `int t`), on a trouvé un cycle et le calcul peut s'arrêter avec une erreur.

Remarque. On parle ici d'un test *dynamique* de terminaison, même si le test est effectué pour calculer la forme de tête d'un type, donc pendant le typage du programme fourni par l'utilisateur. En effet, il ne s'agit pas d'une analyse statique qui décide a priori d'accepter ou rejeter des définitions mutuellement récursive, on détecte la non-terminaison *pendant* qu'on déplie les définitions pour en calculer la forme de tête. C'est l'équivalent d'une « instrumentation » ou surveillance (monitoring) d'un programme qui collecte des informations pendant son exécution, pour l'interrompre s'il risque de ne pas respecter une certaine propriété de sûreté.

Hélas, cette méthode ne garantit pas la terminaison en présence de paramètres de type utilisés de façon dite *non-régulière*, comme dans cet exemple :

```
type 'a t = Loop of ('a list) t [@unboxed]
```

Ici, demander la forme de tête de `int t` conduit à demander celle de `int list t`, puis de `int list list t`, etc. Il y a une boucle infinie, mais avec des expressions de type de plus en plus grosses, chacune distincte de toutes les précédentes.

Détecter la répétition d'un constructeur de type pendant l'expansion ? Dans l'exemple précédent, aucune expression de type ne se répète, mais le constructeur de type `t` se répète en tête de chaque type de la trace, et c'est son expansion qui provoque la non-terminaison. On pourrait rendre le test précédent moins fin (rejeter plus de programmes) en arrêtant avec une erreur les expansions qui répètent le même constructeur de tête plusieurs fois.

Cependant, ce critère est maintenant trop grossier, il rejette des définitions sûres et intéressantes. Voici deux exemples, l'un synthétique et l'autre plus réaliste.

```
type 'a id = Id of 'a [@unboxed]
type t = Foo of int id id [@unboxed]

type 'a located = 'a * Location.t
type expr = expr_ located
and expr_ = ... | Name of name [@unboxed] | ...
and name = string located
```

Dans le premier exemple, calculer la forme de tête de `int id id` demande deux expansions successives du constructeur de tête `id`, et serait rejeté par le critère que nous proposons. Le second exemple est plus réaliste ; il montre que des types paramétrés comme `'a id`, ici `'a located`, peuvent être utilisés pour factoriser des aspects spécifiques d'une structure de données complexe, être utilisés dans plusieurs définitions mutuellement récursives (ici les « expressions » et les « noms » d'un petit langage), et parfois se retrouver plusieurs fois le long d'un chemin de définitions à déplier.

Notre solution : garder une trace par sous-expression Notre approche consiste à détecter la répétition d'un constructeur dans une trace, mais en gardant des traces « locales » : chaque sous-expression de type garde trace des expansions qui ont eu lieu avant son apparition, mais pas des expansions qui ont suivi. Cela suffit pour garantir la terminaison (comme nous le verrons ensuite), mais cela autorise les exemples précédents utilisant `int id id` ou `'a located`.

4.2 Digression formelle : la forme de tête comme une forme normale

Les définitions de types (types sommes et abréviations) des langages ML peuvent être vus comme des termes d'un lambda-calcul simplement typé comportant des définitions globales, mutuellement récursives. Par exemple, les définitions suivantes

```
type 'a id = Id of 'a [@unboxed]
type 'a proc =
  | Base of int id [@unboxed]
  | Ask of 'a -> 'a proc [@unboxed]
  | Tell of 'a * 'a proc
```

Peuvent être vues comme les déclarations suivantes :

$$\mathsf{id} := \lambda\alpha.\ \mathsf{Id}^{\mathsf{unboxed}}\ \mathsf{of}\ \alpha$$

$$\mathsf{proc} := \lambda\alpha.\ (\mathsf{Base}^{\mathsf{unboxed}}\ \mathsf{of}\ (\mathsf{int}\ \mathsf{id})) + (\mathsf{Ask}^{\mathsf{unboxed}}\ \mathsf{of}\ (\alpha, \alpha\ \mathsf{proc})\ (\to)) + (\mathsf{Tell}\ \mathsf{of}\ (\alpha, \alpha\ \mathsf{proc})\ (\times))$$

avec des fonctions n-aires dont l'application se note à l'envers $(\tau_i)_i\ t$ (la fonction t appliquée à une famille d'arguments $(\tau_i)_i$), et où les constructeurs de termes $\mathsf{Base}^{\mathsf{unboxed}}\ \mathsf{of}\ \tau$, certains noms de fonctions primitifs $(\to)$, $(\times)$, et la forme binaire $t + u$ sont vus comme des neutres dont l'application ne se réduit pas.

$$\tau := \alpha \mid C\ \mathsf{of}\ \tau \mid C^{\mathsf{unboxed}}\ \mathsf{of}\ \tau \mid (\tau_i)_i\ t \mid \tau + \tau'$$

En particulier, l'expansion d'un constructeur de tête, remplacé par sa définition où l'on remplace les instances de ses paramètres de type, correspond à la règle de β-réduction usuelle :

$$\frac{t := \lambda(\alpha_i)_i.\ \tau'}{(\tau_i)_i\ t \rightsquigarrow_\beta \tau'[(\alpha_i \leftarrow \tau_i)_i]}$$

Avec cette présentation, la forme de tête d'une expression de type peut se lire depuis la *forme normale* de cette expression pour une certaine stratégie de réduction :

— on réduit sous les sommes $t + u$
— on ne réduit pas sous les applications des constructeurs types de base : $(\times)$, $(\to)$, etc.
— on réduit sous les constructeurs déboîtés C^{unboxed}
— on ne réduit pas sous les constructeurs emboîtés

On définit les contextes de réduction $E[\Box]$ ci-dessous, et une notion de forme normale V correspondante :

$$E ::= \Box \mid C^{\mathsf{unboxed}}\ \mathsf{of}\ \tau \mid \tau + E \mid E + \tau \qquad \frac{\tau \rightsquigarrow_\beta \tau'}{E[\tau] \rightsquigarrow E[\tau']}$$
$$V ::= \alpha \mid C\ \mathsf{of}\ \tau \mid C^{\mathsf{unboxed}}\ \mathsf{of}\ V \mid V + V'$$

La forme normale résultante V n'est pas exactement une forme de tête H, elle porte plus d'information. Pour récupérer exactement la forme de tête, on effectue une simple opération d'effacement $\lfloor V \rfloor$:

$$\begin{array}{lcl} \lfloor \alpha \rfloor & := & \top \\ \lfloor C\ \mathsf{of}\ \tau \rfloor & := & C\ \mathsf{of}\ \tau \\ \lfloor C^{\mathsf{unboxed}}\ \mathsf{of}\ V \rfloor & := & \lfloor V \rfloor \\ \lfloor V + V' \rfloor & := & \lfloor V \rfloor + \lfloor V' \rfloor \end{array} \qquad \frac{((\tau_i)_i\ t,\ \widehat{\mathtt{t}}) \in \widehat{\mathtt{T}}}{\lfloor (\tau_i)_i\ t \rfloor := \widehat{\mathtt{t}}}$$

Par exemple, la forme normale de $\mathsf{bool}\,\mathsf{proc}$ vu comme un λ-terme, pour cette stratégie, est :
$V := \mathsf{Base}^{\mathsf{unboxed}}\mathsf{of}(\mathsf{Id}^{\mathsf{unboxed}}\ \mathsf{of}\ \mathsf{int}) + \mathsf{Ask}^{\mathsf{unboxed}}\mathsf{of}(\mathsf{bool}, \mathsf{bool}\ \mathsf{proc})\ (\to) + \mathsf{Tell}\mathsf{of}(\mathsf{bool}, \mathsf{bool}\ \mathsf{proc})\ (\times)$

et l'effacement décrit ci-dessus nous redonne exactement sa forme de tête :

$$\lfloor V \rfloor = \widehat{\texttt{int}} + \widehat{\texttt{function}} + \mathsf{Tell\ of\ bool} \times \mathsf{bool\ proc}$$

Lemme 2. On a $\tau \Rightarrow H$ si et seulement τ a une forme normale V qui s'efface en H.

$$\tau \Rightarrow H \qquad \Longleftrightarrow \qquad \exists V, \quad \tau \rightsquigarrow^* V \quad \wedge \quad \lfloor V \rfloor = H$$

Remarque. Avec la même β-réduction, mais d'autres stratégies de réduction et d'autres analyses des formes normales résultantes, on peut représenter d'autres analyses des expressions de type qui ont besoin de déplier les définitions. Y compris des analyses déjà existantes en OCaml, par exemple : « est-ce que ce type ne contient que des valeurs immédiates ? ». Notre Théorème 1 de terminaison s'applique aussi dans le cas d'une relation de réduction maximale où on peut réduire en profondeur les arguments d'un constructeur, ou sous les constructeurs emboîtés, donc il s'applique en fait à toutes les analyses possibles sur des formes normales de cette grammaire.

4.3 Types annotés par des traces

Nous définissons une nouvelle grammaire d'expressions de types « annotées » $\bar{\tau}$, où chaque expression et sous-expression de type est une paire $\tau \mathbin{@} l$ contenant une trace d'expansion l, qui est une liste de constructeurs de type t.

$$\bar{\tau} \quad ::= \quad \tau \mathbin{@} l \qquad\qquad \tau \quad ::= \quad \alpha \mid (\bar{\tau}_i)_i\ t \qquad\qquad l \quad ::= \quad \emptyset \mid l, t$$

Un exemple de type annoté est $(\alpha \mathbin{@} l_\alpha, \beta \mathbin{@} l_\beta)\ t \ @\ l_t$. L'expression en tête applique le constructeur de type t, avec la trace l_t, et les deux paramètres de t sont aussi annotés avec les traces l_α et l_β.

Remarque. Nous réutilisons le non-terminal τ dans cette grammaire par analogie avec la grammaire τ des expressions de type usuelles, mais ici τ désigne une expression de type dont les sous-expressions sont annotées. Ce que nous voulons dire par τ devrait être clair selon le contexte ; dans la suite du document nous n'utilisons pas la méta-variable $\bar{\tau}$ mais toujours explicitement la paire $\tau \mathbin{@} l$, et les usages « annotés » de τ sont toujours de cette forme.

On peut maintenant raffiner la relation de réduction sur les types annotés, pour formaliser notre algorithme de contrôle dynamique de la terminaison : avant de déplier une définition de type de donnée, on vérifie que le constructeur correspondant n'apparaît pas déjà dans la trace de l'application du constructeur, et sinon on réduit vers un nouveau terme d'erreur `Cycle`.

$$\bar{\tau}_\perp \quad ::= \quad \bar{\tau} \mid \texttt{Cycle} \qquad \frac{t := \lambda(\alpha_i)_i.\ \tau' \qquad t \notin l}{(\tau_i)_i\ t \mathbin{@} l \rightsquigarrow_\beta \tau'[(\alpha_i \leftarrow \tau_i)_i]^{@l,t}} \qquad \frac{t := \lambda(\alpha_i)_i.\ \tau' \qquad t \in l}{(\tau_i)_i\ t \mathbin{@} l \rightsquigarrow_\beta \texttt{Cycle}}$$

$$V_\perp \quad ::= \quad V \mid \texttt{Cycle} \qquad \frac{\tau \rightsquigarrow_\beta \tau'}{E[\tau] \rightsquigarrow E[\tau']} \qquad \frac{\tau \rightsquigarrow_\beta \texttt{Cycle}}{E[\tau] \rightsquigarrow \texttt{Cycle}}$$

La nouvelle définition de la β-réduction remplace la substitution d'expressions de types non-annotées $\tau'_k[(\alpha_i)_i \leftarrow (\tau_i)_i]$ par une « substitution annotante » $\tau'_k[(\alpha_i)_i \leftarrow (\tau_i \mathbin{@} l_i)_i]^{@l,t}$, une méta-opération $\tau[\sigma]^{@l}$ définie ainsi :

$$\begin{aligned} \alpha[\sigma]^{@l} &= \sigma(\alpha) \\ ((\tau_i)_i\ t)[\sigma]^{@l} &= \Big((\tau_i[\sigma]^{@l})\ t\Big) \mathbin{@} l \end{aligned}$$

La substitution annotante $\tau[\sigma]^{@l}$ prend une expression de type non-annotée τ, une substitution σ de variables de type libres vers des expressions de types annotées $\tau_i \mathbin{@} l_i$, et une « trace courante » l. Elle renvoie une expression de type annotée qui utilise les $\tau_i \mathbin{@} l_i$ à la place des variables α, et qui sur toutes les autres sous-expressions de type utilise la trace l. Par exemple :

$$(\alpha \to \mathsf{int})[\alpha \leftarrow \tau_\alpha \mathbin{@} l_\alpha]^{@l} \qquad = \qquad (\tau_\alpha \mathbin{@} l_\alpha \to \mathsf{int} \mathbin{@} l) \mathbin{@} l$$

L'effet de cette substitution annotante dans la règle de β-réduction est le suivant. Quand on expanse une version annotée de $(\tau_i)_i\, t$, les arguments τ_i du constructeur de type t sont déjà annotés, et ils conservent leur annotation dans l'expansion. Par contre, les « nouvelles » sous-expressions de type, qui viennent de la définition du type de donnée t qui s'expanse en un type paramétré τ', ne portaient pas d'annotation. On leur donne, dans le résultat, la trace courante de l'expansion, l, à laquelle on ajoute le constructeur t que l'on vient d'expanser.

Ainsi, à tout moment du calcul, chaque sous-expression de type est annotée par la trace des expansions effectuées jusqu'à son apparition, par expansion d'un constructeur de type.

On peut enfin définir un jugement $\tau \Rightarrow_\bot H_\bot$ de mise en forme de tête contenant ce contrôle de la terminaison. Ci-dessous, l'expression $\tau[\emptyset]^{@\emptyset}$ correspond à annoter en profondeur un type non-annoté τ avec la trace vide $\emptyset$.

$$H_\bot ::= H \mid \texttt{Cycle} \qquad\qquad \lfloor \texttt{Cycle} \rfloor := \texttt{Cycle} \qquad\qquad \frac{\tau[\emptyset]^{@\emptyset} \rightsquigarrow^\star V_\bot}{\tau \Rightarrow_\bot \lfloor V_\bot \rfloor}$$

Cette stratégie de contrôle de la terminaison est *correcte* : les calculs qu'elle produit sans lever l'erreur `Cycle` correspondent à des réductions dans le système non-annoté de départ.

Lemme 3 (Correction du contrôle de terminaison)**.**
Si $\tau \mathbin{@} l \rightsquigarrow^\star \tau' \mathbin{@} l'$, alors les termes non-annotés τ, τ' correspondant sont aussi dans la relation $\tau \rightsquigarrow^\star \tau'$. Par conséquent, si $\tau \Rightarrow_\bot H$ alors $\tau \Rightarrow H$.

Remarque. Notre implémentation n'utilise pas une relation petit-pas comme ici, mais une fonction récursive plus proche d'une relation grand-pas comme le jugement initial : on utilise partout des expressions de type annotées $\tau \mathbin{@} l$, et on utilise la même substitution annotante dans la règle Type. La présentation petit-pas, équivalente, est choisie ici car elle facilite la présentation de la preuve de terminaison dans la section suivante.

Exemple avec cycle Dans le cas de la définition

```
type 'a loop = Loop of 'a list loop [@unboxed]
```

calculer la forme de tête de ce type de donnée demande de normaliser le type annoté

$$(\alpha \mathbin{@} \emptyset)\ \mathsf{loop} \mathbin{@} \emptyset$$

Une étape de β-réduction donne l'expression de type annotée suivante :

$$(\alpha' \ \mathsf{list})\ \mathsf{loop}[\alpha' \leftarrow \alpha \mathbin{@} \emptyset]^{@[\mathsf{loop}]} \qquad = \qquad ((\alpha \mathbin{@} \emptyset)\ \mathsf{list} \mathbin{@} [\mathsf{loop}])\ \mathsf{loop} \mathbin{@} [\mathsf{loop}]$$

(Nous notons $[\mathsf{loop}]$ pour la trace $(\emptyset, \mathsf{loop})$.) Dans cette expression, α garde son annotation de départ, mais les deux nouveaux constructeurs de type list et loop sont maintenant annotés avec une trace qui indique qu'ils ont été introduits par l'expansion du constructeur loop.

Il n'est plus possible d'expanser ce terme car la prémisse $t \notin l$ est invalide : le constructeur loop apparaît dans la trace $[\mathsf{loop}]$. La seule réduction possible est vers le terme d'erreur `Cycle`.

$$\frac{(\alpha\ \mathsf{loop})[\emptyset]^{@\emptyset} = (\alpha \mathbin{@} \emptyset)\ \mathsf{loop} \mathbin{@} \emptyset \rightsquigarrow ((\alpha \mathbin{@} \emptyset)\ \mathsf{list} \mathbin{@} [\mathsf{loop}])\ \mathsf{loop} \mathbin{@} [\mathsf{loop}] \rightsquigarrow \texttt{Cycle}}{\left(\mathsf{type}\ \alpha\ \mathsf{loop}\ =\ \mathsf{Loop}^{\mathsf{unboxed}}\ \mathsf{of}\ \alpha\ \mathsf{list}\ \mathsf{loop}\right) \Rightarrow \texttt{Cycle}}$$

Exemple sans cycle Reprenons notre exemple de l'expression de type problématique `int id id` :

```
type 'a id = Id of 'a [@unboxed]
type t = Foo of int id id [@unboxed]
```

Calculer la forme de tête de `t` revient à normaliser l'expression annotée

$$((\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset$$

qui se réduit en

$$\alpha'[\alpha' \leftarrow (\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset]^{@[\mathsf{id}]} \qquad = \qquad (\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset$$

L'expression de type que nous obtenons vient de l'expansion de id, mais les constructeurs de type qui la composent ne sont pas « nouveaux », ils viennent du paramètre de l'expression de départ où ils étaient annotés, et gardent la même annotation. En particulier, l'expression de type résultante ne porte pas la trace [id], qui empêcherait d'y expanser de nouveau id, mais la trace initiale $\emptyset$. Nous pouvons faire une nouvelle expansion vers une forme normale : $\mathsf{int} \mathbin{@} \emptyset$.

$$\frac{((\mathsf{int}\ \mathsf{id})\ \mathsf{id})[\emptyset]^{@\emptyset} = ((\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset \rightsquigarrow (\mathsf{int} \mathbin{@} \emptyset)\ \mathsf{id} \mathbin{@} \emptyset \rightsquigarrow \mathsf{int} \mathbin{@} \emptyset \qquad \dfrac{(\mathsf{int}, \widehat{\texttt{int}}) \in \widehat{\mathsf{T}}}{\lfloor \mathsf{int} \rfloor = \widehat{\texttt{int}}}}{\left(\mathsf{type\ t} \ =\ \mathsf{Foo}^{\mathsf{unboxed}}\ \mathsf{of\ int\ id\ id}\right) \Rightarrow \widehat{\texttt{int}}}$$

4.4 Preuve de terminaison

Nous présentons notre argument de terminaison dans un cadre plus abstrait de récriture d'arbres : des arbres n-aires dont les nœuds sont étiquetés, avec un process d'expansion d'un arbre vers un nouvel arbre, et une relation d'ordre bien fondée sur les noeuds qui garantit la terminaison des récritures par expansion.

4.4.1 Arbres n-aires

Définition (Arbres n-aires ouverts)**.** Étant donné deux ensembles $\mathcal{N}$ (les étiquettes de noeuds) et $\mathcal{V}$ (les variables), on définit les arbres n-aires ouverts $a \in \mathsf{Tree}(\mathcal{N}, \mathcal{V})$ par la grammaire :

$$\begin{array}{rcll} a \in \mathsf{Tree}(\mathcal{N}, \mathcal{V}) & ::= & \mathsf{node}(c,\ (a_i)_i) & (c \in \mathcal{N}, i \in \{1, 2, \ldots, n\}) \\ & | & \mathsf{var}(x) & (x \in \mathcal{V}) \end{array}$$

Un nœud d'un arbre est formé soit d'un constructeur $c \in \mathcal{N}$ suivi d'un nombre arbitraire (fini) de sous-arbres, soit par une variable $x \in \mathcal{V}$.

Définition (join)**.** Ces arbres ont une structure monadique, on peut définir une opération

$$\mathsf{join} : \mathsf{Tree}(\mathcal{N}, \mathcal{V} \uplus \mathsf{Tree}(\mathcal{N}, \mathcal{V})) \to \mathsf{Tree}(\mathcal{N}, \mathcal{V})$$

$$\frac{}{\mathsf{join}(\mathsf{node}(c,\ (a_i)_i)) := \mathsf{node}(c,\ (\mathsf{join}(a_i))_i)} \qquad \frac{x \in \mathcal{V}}{\mathsf{join}(\mathsf{var}(x)) := \mathsf{var}(x)} \qquad \frac{x \in \mathsf{Tree}(\mathcal{N}, \mathcal{V})}{\mathsf{join}(\mathsf{var}(x)) := x}$$

Définition (π, $b \in_\pi a$, $\mathsf{Subtree}(a)$, $\pi.\pi'$, $\pi' \geq \pi$)**.** On définit un chemin π, la relation $b \in_\pi a$ si π est un chemin du sous-arbre b à son parent a, et le multi-ensemble $\mathsf{Subtree}(a)$ des sous-arbres de a.

$$\begin{array}{rcll} \pi & ::= & \emptyset & \\ & | & \pi.i & (i \in \mathbb{N}) \end{array} \qquad \frac{}{a \in_\emptyset a} \qquad \frac{b \in_\pi a_i}{b \in_{\pi.i} \mathsf{node}(c,\ (a_i)_i)}$$

$$\mathsf{Subtree}(a) := \{\!\{ b \mid \exists \pi,\ b \in_\pi a \}\!\}$$

On peut remarquer que la relation $b \in_\pi a$ est compatible avec la définition naturelle de la concaténation de chemins : si $a_1 \in_{\pi_{12}} a_2$ et $a_2 \in_{\pi_{23}} a_3$, alors $a_1 \in_{\pi_{12}.\pi_{23}} a_3$.

Enfin, on dit que π' est une extension de π, notée $\pi' \geq \pi$, si π' est de la forme $\pi''.\pi$.

$$\pi' \geq \pi \qquad := \qquad \exists \pi'',\ \pi' = \pi''.\pi$$

Définition ($a[\pi \leftarrow b]$)**.** $a[\pi \leftarrow b]$ est l'arbre a où le sous-arbre en π est remplacé par b.

$$\frac{}{a[\emptyset \leftarrow b] := b} \qquad \frac{a_j[\pi \leftarrow b] = a'}{\mathsf{node}(c,\ (a_i)_{i\in I})[\pi.j \leftarrow b] := \mathsf{node}(c,\ \Big((a_i)_{i\in J\setminus\{i\}}, (j \mapsto a')\Big))}$$

Ce cadre général nous intéresse car il modélise notre calcul de forme de tête.

Exemple (Arbres de types annotés)**.** Nous considérons comme des arbres n-aires les expressions de type annotées $\tau \mathbin{@} l$: ces expressions se plongent dans l'ensemble $\mathsf{Tree}(\mathcal{T}, \emptyset)$ pour un ensemble de constructeurs de noeuds $\mathcal{T}$ ayant la grammaire suivante :

$$c ::= \alpha \mathbin{@} l \mid t \mathbin{@} l$$

Définition ($\mathsf{tree}(\tau)$)**.** Nous notons $\mathsf{tree}(\tau)$ pour l'arbre correspondant au type annoté τ.

$$\mathsf{tree}(\alpha \mathbin{@} l) := \mathsf{node}(\alpha \mathbin{@} l, \emptyset) \qquad \mathsf{tree}((\bar{\tau}'_i)_i \mathbin{@} l) := \mathsf{node}(t \mathbin{@} l, (\mathsf{tree}(\bar{\tau}'_i))_i)$$

Par exemple l'expression de type annotée $(\alpha \mathbin{@} l_\alpha, \beta \mathbin{@} l_\beta)\ t \mathbin{@} l_t$ devient l'arbre

$$\mathsf{node}(t \mathbin{@} l_t,\ (0 \mapsto \mathsf{node}(\alpha \mathbin{@} l_\alpha, \emptyset), 1 \mapsto \mathsf{node}(\beta \mathbin{@} l_\beta, \emptyset)))$$

4.4.2 Expansion d'arbre

Nous allons maintenant définir une notion d'expansion sur les arbres, qui généralise l'expansion des définitions dans les expressions de type. Informellement, nous considérons le processus d'expansion suivant :

- Un nœud (en position arbitraire) de l'arbre est choisi pour être expansé ; le sous-arbre dont il est racine est appelé le « sous-arbre expansé ».
- Le processus d'expansion remplace le sous-arbre expansé par un sous-arbre contenant :
  - zéro, un ou plusieurs « nouveaux » nœuds qui ne correspondent pas à une occurrence d'un noeud dans l'arbre de départ, et
  - un nombre arbitraire de copies de certains sous-arbres stricts du nœud expansé.

Un sous-arbre du nœud expansé peut disparaître pendant l'expansion, ou être dupliqué en plusieurs sous-arbres, placés à l'intérieur du sous-arbre remplaçant le sous-arbre expansé.

Définition (Expansion d'arbre)**.** Une expansion d'arbre *en tête* $b \rightsquigarrow_\beta b'$ consiste à remplacer un arbre b par un arbre b' dont certains sous-arbres sont des sous-noeuds *stricts* de b.

$$\frac{b \in \mathsf{Tree}(\mathcal{N}, \mathcal{V}) \qquad b_s \in \mathsf{Tree}(\mathcal{N}, \mathcal{V} \uplus (\mathsf{Subtree}(b) \setminus \{b\}))}{b \rightsquigarrow_\beta \mathsf{join}(b_s)}$$

Une expansion d'arbre $a \rightsquigarrow a'$ est une expansion en tête d'un sous-arbre de a.

$$\frac{b \in_{\pi_b} a \qquad b \rightsquigarrow_\beta b'}{a \rightsquigarrow a[\pi_b \leftarrow b']}$$

Exemple. La Figure 4.4.2 représente un arbre à gauche, et trois expansions possibles de cet arbre à droite, toutes à partir du nœud c, où les « nouveaux nœuds » sont montrés en rouge. Dans la première expansion, le nœud c est remplacé par un nouveau nœud c', auquel sont attachés les enfants du nœud c (ni effacés ni dupliqués). Dans la seconde expansion, il n'y a pas de nouveau nœud, il ne reste à la place de c que son sous-arbre de racine e (et d est effacé). Dans la troisième expansion, il y a deux nouveau nœuds c' et c'', deux copies du sous-arbre de racine e, une copie du sous-arbre de racine f, et d est effacé.

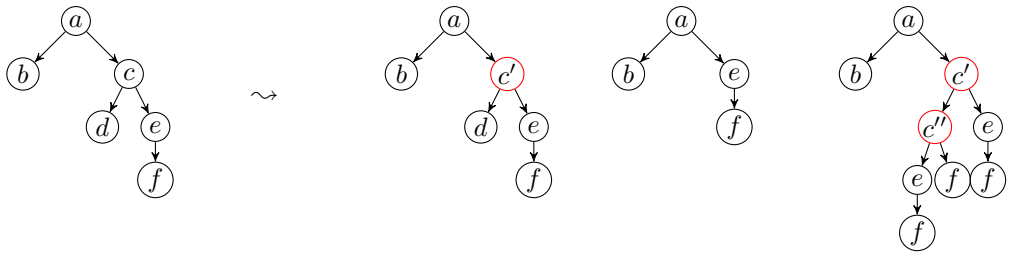


FIGURE 1 – Expansions de sous-arbres

Exemple. Le processus d'expansion d'arbre généralise l'expansion des définitions dans une expression de type : si $\tau \rightsquigarrow \tau'$, alors $\mathsf{tree}(\tau) \rightsquigarrow \mathsf{tree}(\tau')$.

En particulier, à toute séquence infinie d'expansions d'une expression de type correspond une séquence infinie d'expansions sur l'arbre correspondant.

Remarque. Avec nos expressions de type annotées, un type paramétré $(\tau_i)_i\ t$ s'expanse en remplaçant des variables de la définition de t par les paramètres τ_i. Au niveau des arbres cela correspond à des expansions où les « vieux » sous-arbres sont toujours les enfants directs du nœud expansé, et pas des sous-arbres stricts arbitraires. Mais cette notion plus riche d'expansion est utile pour modéliser les contraintes de type du langage OCaml

```
type 'a t = ('b * bool) constraint 'a = 'b * 'b * int
```

qui permettent effectivement de désigner des sous-expressions de type arbitraires.

Définition ($\mathsf{Tree}(\mathcal{N})$)**.** L'ensemble des variables $\mathcal{V}$ est utile pour définir l'expansion par substitution. On ne s'en servira plus par la suite : on notera $\mathsf{Tree}(\mathcal{N})$ pour $\mathsf{Tree}(\mathcal{N}, \emptyset)$.

Définition ($\mathsf{constr}(a)$)**.** Pour $a \in \mathsf{Tree}(\mathcal{N})$, on définit $\mathsf{constr}(a)$ le constructeur de tête de a (qui ne peut pas être une variable, puisqu'on est sur $\mathsf{Tree}(\mathcal{N}, \emptyset)$:

$$\mathsf{constr}(\mathsf{node}(c, (a_i)_i)) := c$$

4.4.3 Terminaison de l'expansion

Définition (Mesure)**.** Si A est un ensemble, on appelle *mesure sur* A le choix d'une fonction $m_A : A \to M$ pour un ensemble $(M, <_M)$ muni d'un ordre bien fondé [4].

Définition (Expansion mesurée)**.** Étant donné une mesure $m_{\mathcal{N}}$ sur $\mathcal{N}$, on dit qu'une expansion $a \rightsquigarrow a' \in \mathsf{Tree}(\mathcal{N})$ est *mesurée* si les « nouveaux » nœuds de l'expansion ont une mesure

4. Un ordre $(<)$ est bien fondé s'il n'existe pas de chaîne infinie descendante, de suite $(m_i)_{i\in\mathbb{N}}$ avec $\forall i \in \mathbb{N},\ m_i > m_{i+1}$.

strictement inférieure au nœud expansé, c'est-à-dire si on a

$$\frac{b_s \in \mathsf{Tree}(\mathcal{N}, \mathsf{Subtree}(a) \setminus \{a\})}{\dfrac{b \in_{\pi_b} a \qquad b \rightsquigarrow_\beta \mathsf{join}(b_s)}{a \rightsquigarrow a[\pi \leftarrow \mathsf{join}(b_s)] = a'}}$$

où les nœuds de b_s sont strictement inférieurs au nœud b :

$$\forall \pi,\ \mathsf{node}(c, \_) \in_\pi b_s \implies m_{\mathcal{N}}(c) < m_{\mathcal{N}}(\mathsf{constr}(b))$$

Lemme 4. L'expansion d'expressions de type annotées est une expansion mesurée.

Démonstration. Une expansion de tête dans un type annoté est de l'une des formes

$$\frac{t := \lambda(\alpha_i)_i.\ \tau' \qquad t \notin l}{(\tau_i)_i\ t \mathbin{@} l \rightsquigarrow_\beta \tau'[(\alpha_i \leftarrow \tau_i)_i]^{@l,t}} \qquad \frac{t := \lambda(\alpha_i)_i.\ \tau' \qquad t \in l}{(\tau_i)_i\ t \mathbin{@} l \rightsquigarrow_\beta \texttt{Cycle}}$$

Comme mesure d'un nœud $t \mathbin{@} l$, on prend tout simplement la trace l, ordonnée par anti-inclusion : $l \leq l'$ si et seulement si $l \supseteq l'$. Par construction, nos traces ne peuvent mentionner qu'au plus une fois chaque constructeur de type, et le nombre de constructeurs de types déclarés dans le programme programme est fini ; il existe donc une taille maximale de traces possibles (le nombre de définitions de type), et l'ordre sur les traces est bien fondé. Enfin, lors de l'expansion d'un nœud $\tau \mathbin{@} l$, les nouveaux nœuds viennent de la substitution $\tau'[(\alpha_i \leftarrow \tau_i)_i]^{@l,t}$, et ils sont donc annotés par une trace strictement plus petite l, t.

Pour les réductions $\tau \rightsquigarrow_\beta \texttt{Cycle}$, on ajoute aussi `Cycle` à cet ensemble de mesures, en étendant l'ordre avec $\texttt{Cycle} < l$ pour tout l ; il reste bien fondé. Pour résumer :

$$\begin{array}{lcl} m(\tau \mathbin{@} l) & := & l \\ m(\texttt{Cycle}) & := & \texttt{Cycle} \end{array} \qquad \frac{l \supseteq l'}{l \leq l'} \qquad \frac{}{\texttt{Cycle} < l}$$

□

Théorème 1 (Terminaison)**.** Si $\mathcal{N}$ est muni d'une mesure $m_{\mathcal{N}}$, les expansions mesurées sur $\mathsf{Tree}(\mathcal{N})$ sont fortement normalisantes : il n'existe pas de séquence infinie d'expansions mesurées.

Pour montrer la terminaison de l'expansion d'arbre, on utilise notre mesure $m_{\mathcal{N}}$ sur les nœuds pour construire une mesure sur les arbres, qui envoie les arbres sur un ensemble muni d'un ordre bien fondé, telle qu'une expansion fait décroire strictement la mesure d'un arbre.

Étant donné un ensemble $(S, <_S)$ muni d'un ordre bien fondé, on peut construire un ordre bien fondé $(<_{\mathcal{P}(S)})$ sur l'ensemble $\mathcal{P}(S)$ des ensembles d'éléments de S, et un ordre bien fondé $(<_{\mathcal{M}(S)})$ sur l'ensemble $\mathcal{M}(S)$ des multi-ensembles d'éléments de S (cet ordre est attribué à Dershowitz, Manna, Huet et Oppen).

— $S_1 <_{\mathcal{P}(S)} S_2$ si, pour tout $a \in S_1$, il existe $b \in S_2$ tel que $a <_S b$.
— $M_1 <_{\mathcal{M}(S)} M_2$ si M_1 est de la forme $M + N_1$ et M_2 est de la forme $M + N_2$ avec $\lfloor N_1 \rfloor_{\mathsf{set}} <_{\mathcal{P}(S)} \lfloor N_2 \rfloor_{\mathsf{set}}$.

À partir d'une mesure $m_{\mathcal{N}} : \mathcal{N} \to (O, <_O)$ sur les étiquettes des nœuds de nos arbres (où $(<_O)$ est bien fondé), on définit pour tout arbre $a \in \mathsf{Tree}(\mathcal{N})$ une mesure $m_{\mathsf{Node}(a)} : \mathsf{Subtree}(a) \to (\mathcal{M}(O), <_{\mathcal{M}(O)})$ sur les nœuds de a, en considérant pour chaque nœud le chemin entre ce nœud (inclus) et la racine de l'arbre, vu comme un multi-ensemble de nœuds.

On définit notre mesure $m_{\mathsf{Node}(a)}(b)$, pour chaque nœud $b \in \mathsf{Subtree}(a)$, comme la somme de la mesure du constructeur de b et d'une mesure $m_{\mathsf{Path}(a)}$ définie par induction sur $b \in_\pi a$.

$$m_{\mathsf{Node}(a)}(b) \quad := \quad \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b))\}\!\} + m_{\mathsf{Path}(a)}(b \in_\pi a) \qquad m_{\mathsf{Path}(a)}\left(a \in_\emptyset a\right) := \emptyset$$

$$m_{\mathsf{Path}(\mathsf{node}(c,\,(a_i)_i))}\left(\frac{b \in_\pi a_i}{b \in_{\pi.i} \mathsf{node}(c,\,(a_i)_i)}\right) := \{\!\{m_{\mathcal{N}}(c)\}\!\} + m_{\mathsf{Path}(a_i)}\left(b \in_\pi a_i\right)$$

Notons que la mesure $m_{\mathsf{Path}(a)}$ (pas $m_{\mathsf{Node}(a)}$) est compatible avec la concaténation de chemins :

$$a_1 \in_{\pi_{12}} a_2 \quad \wedge \quad a_2 \in_{\pi_{23}} a_3$$

$$\implies \qquad m_{\mathsf{Path}(a_3)}(a_1 \in_{\pi_{12}.\pi_{23}} a_3) \quad = \quad m_{\mathsf{Path}(a_2)}(a_1 \in_{\pi_{12}} a_2) + m_{\mathsf{Path}(a_3)}(a_2 \in_{\pi_{23}} a_3)$$

À partir de cette mesure $m_{\mathsf{Node}(a)} : \mathsf{Subtree}(a) \to (\mathcal{M}(O), <_{\mathcal{M}(O)})$ on définit une mesure $m_{\mathsf{Tree}} : \mathsf{Tree}(\mathcal{N}) \to (\mathcal{M}(\mathcal{M}(O)), <_{\mathcal{M}(\mathcal{M}(O))})$ sur les arbres, en considérant pour chaque arbre le multi-ensemble des mesures de ses nœuds. Les arbres sont donc mesurés comme des multi-ensembles de multi-ensembles d'étiquettes munies d'un ordre bien fondé :

$$m_{\mathsf{Tree}}(a) = \{\!\{m_{\mathsf{Node}(a)}(b) \mid b \in \mathsf{Subtree}(a)\}\!\}$$

Il faut montrer que les expansions mesurées pour $m_{\mathcal{N}}$ font strictement décroitre m_{Tree}. Cette deuxième partie de la preuve, moins difficile que ce bon choix de mesure, est faite en Annexe A.1.

4.5 Complétude ? « Oui ! Pour l'instant. »

On sait maintenant que la relation de récriture sur les expressions de type annotées $\tau \mathbin{@} l$ de recherche de preuve de termine nécessairement. En particulier, pour toutes les définitions récursives dont l'expansion ne termine pas, notre algorithme renverra `Cycle` en un temps fini – le nombre d'expansions est même borné par le nombre de constructeurs de type définis dans l'environnement. De plus, pour tous les exemples non-cycliques que nous avons considérés jusque-là, l'algorithme calcule effectivement la forme de tête et ne s'arrête pas (à tort) avec le résultat `Cycle`. Existe-t-il des définitions pour lesquelles l'algorithme renvoie `Cycle`, alors que la séquence d'expansions aurait en fait terminé ?

La réponse à cette question dépend de l'expressivité du langage de déclarations de type. Dans le fragment que nous avons présenté, la réponse est *non* : notre détection de la terminaison est correcte *et* complète. Pour le langage OCaml tout entier, nous conjecturons que notre algorithme est toujours complet ; mais l'ajout de fonctionnalités plus avancées (types de sorte supérieure, etc.) pourrait perdre cette propriété. Sur le long terme, nous nous contentons d'affirmer que l'algorithme termine sur tous les exemples *utiles* que nous avons rencontré, et qu'il pourrait être à raffiner si des fonctionnalités avancées sont ajoutées au langages, qui permettent d'écrire des définitions utiles sur lequel il serait incomplet. Nous considérons donc le résultat de complétude, esquissé ci-dessous, comme une curiosité temporaire.

La complétude de notre algorithme peut sembler surprenante ; d'habitude dans notre domaine un algorithme est soit correct, soit complet, mais jamais les deux a la fois, car le problème sous-jacent est indécidable. En général le problème de la terminaison d'un λ-terme en présence de définitions récursives arbitraires est certainement indécidable. Mais les λ-termes correspondant aux définitions de type ML (voir Section 4.2) des propriétés spécifiques :

— La seule règle de réduction est la β-expansion des fonctions, tous les autres constructeurs de termes sont inertes/neutres.

— On est dans un fragment de *premier ordre* : les termes sont bien typés pour un système de types simples où les arguments d'une fonction ont toujours le type de base α, jamais un type de fonction. Par exemple, un type paramétré $(\alpha, \beta)\, t$ a toujours le type (la sorte) $\star \times \star \to \star$, notre grammaire ne permet pas de placer une variable α en position de constructeur de type.

Dans ce fragment, une β-expansion ne crée jamais de « nouveaux » radicaux (redexes) : les arguments d'une fonction ne sont jamais des λ-abstractions dont la substitution créerait un nouveau radical, les radicaux sont donc déjà présentes dans les arguments expansés ou dans le corps de la fonction.

En particulier, si on a un type $(\tau_i)_i\, t$ dont les arguments τ_i sont normalisants, mais qui n'est pas normalisant lui-même [5], alors la non-terminaison ne dépend pas des τ_i : le terme $(\alpha_i)_i\, t$ ne terminerait pas non plus. La source de non-terminaison (le prochain radical dans la séquence de réductions infinie) vient de la définition du constructeur de type t. Dans notre système annoté, si le type $(\tau_i)_i\, t$ a la trace l, alors le prochain radical aura la trace l, t, et ainsi de suite. On extraire de toute séquence de réduction infinie une sous-séquence où chaque radical a une trace plus grande que le précédent. En conséquence, notre critère dynamique de détection des boucles est complet, il interromprait toute séquence de réduction infinie dans ce langage.

Conclusion

Bien gérer les constructeurs emboîtés demande de résoudre un problème de terminaison en présence de définitions récursives étonnamment intéressant.

Remarque. Nous manquons de place pour parler d'autres aspects de ce travail, que nous avons donc laissés en appendices dans les pages qui suivent.

Acknowledgments We received good feedback on this work from the OCaml community, in particular the excellent comments of Stephen Dolan informed several aspects of our work. Stephen was also effective as a « proof assistant », finding flaws in our first three/four proof attempts. Irène Waldspurger suggested measuring nodes by their path to the root, the key ingredient missing from our previous proofs. Finally, the combative review of Adrien Guatto greatly improved the presentation.

Références

Nicolas Chataing and Gabriel Scherer. Constructor unboxing: modified proposal, 2021.

Simon Colin, Rodolphe Lepigre, and Gabriel Scherer. Unboxing Mutually Recursive Type Definitions in OCaml. In *JFLA 2019*, January 2019.

Torbjörn Granlund and contributors. Gmp, 1991.

Antoine Miné and Xavier Leroy. Zarith, 2012.

Jeremy Yallop. Rfc proposal: constructor unboxing, 2020.

5. Si on a un terme non-normalisant, il existe toujours un sous-terme de cette forme : soit le constructeur de tête a des arguments normalisants, soit l'un des arguments est non-normalisant et on raisonne récursivement sur celui-là.

A Haut niveau (grenier)

A.1 Terminaison

Preuve du théorème de terminaison (Théorème 1). Considérons une expansion mesurée $a \rightsquigarrow a'$, elle est nécessairement de la forme suivante, où un sous-arbre b à un chemin π_b dans a est remplacé par un sous-arbre $\mathsf{join}(b_s)$, où les nœuds de b_s sont de mesure strictement plus petite :

$$\frac{b_s \in \mathsf{Tree}(\mathcal{N}, \mathsf{Subtree}(a) \setminus \{a\}) \qquad \forall \pi, (\mathsf{node}(c, \_)) \in_\pi b_s \implies m_{\mathcal{N}}(c) < m_{\mathcal{N}}(\mathsf{constr}(b))}{\dfrac{b \in_{\pi_b} a \qquad b \rightsquigarrow_\beta \mathsf{join}(b_s)}{a \rightsquigarrow a[\pi_b \leftarrow \mathsf{join}(b_s)] = a'}}$$

On veut montrer que $m_{\mathsf{Tree}}(a) > m_{\mathsf{Tree}}(a')$.

Séparer les nœuds modifiés des nœuds inchangés On peut séparer les chemins π valides sur a et a' en deux ensembles : les chemins qui sont des extensions de π_b ($\pi \geq \pi_b$), qui désignent un sous-arbre du sous-arbre b (dans a) ou du sous-arbre $\mathsf{join}(b_s)$ (dans a'), et les chemins $\pi \not\geq \pi_b$ qui ne sont pas des extensions de π_b.

$$\begin{aligned}
& m_{\mathsf{Tree}}(a) \\
= \;& \{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \in \mathsf{Subtree}(a)\}\!\} \\
= \;& \{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi,\ b' \in_\pi a\}\!\} \\
= \;& \{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi \geq \pi_b,\ b' \in_\pi a\}\!\} + \{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi \not\geq \pi_b,\ b' \in_\pi a\}\!\}
\end{aligned}$$

$$\begin{aligned}
& m_{\mathsf{Tree}}(a') \\
= \;& \{\!\{m_{\mathsf{Node}(a')}(b') \mid \exists b' \exists \pi \geq \pi_b,\ b' \in_\pi a'\}\!\} + \{\!\{m_{\mathsf{Node}(a')}(b') \mid \exists b' \exists \pi \not\geq \pi_b,\ b' \in_\pi a'\}\!\}
\end{aligned}$$

La mesure $m_{\mathsf{Node}(a)}(b')$ d'un sous-arbre b' de a ne dépend que des constructeurs sur le chemin entre la racine a et le sous-arbre b' (inclus). En particulier, si b' est à un chemin $\pi \not\geq \pi_b$ qui n'étend pas π_b, il est aussi présent au même chemin comme sous-arbre de a', où il a la même mesure, puisque les constructeurs de ces arbres diffèrent seulement sous π_b :

$$\{\!\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi \not\geq \pi_b,\ b' \in_\pi a\}\!\} \quad = \quad \{\!\{m_{\mathsf{Node}(a')}(b') \mid \exists b' \exists \pi \not\geq \pi_b,\ b' \in_\pi a'\}\!\}$$

Et il nous reste à montrer, par définition de l'ordre sur les multi-ensembles, la partie $\lfloor N_1 \rfloor_{\mathsf{set}} > \lfloor N_2 \rfloor_{\mathsf{set}}$:

$$\{m_{\mathsf{Node}(a)}(b') \mid \exists b' \exists \pi \geq \pi_b,\ b' \in_\pi a\} \quad >_{\mathcal{P}(\mathcal{M}(O))} \quad \{m_{\mathsf{Node}(a')}(b') \mid \exists b' \exists \pi \geq \pi_b,\ b' \in_\pi a'\}$$

Classification d'un chemin dans $\mathsf{join}(b_s)$ Pour chaque sous-arbre sous π_b dans a', donc dans $\mathsf{join}(b_s)$, il faut montrer qu'il existe un sous-arbre sous π_b dans a, donc dans b, de mesure strictement plus grande.

Remarquons (et prouvons) qu'un sous-arbre b' de a' a un chemin de la forme $\pi_s.\pi_n.\pi_b$, avec :

- le chemin π_b entre b et la racine a,
- Un chemin π_n ($n$ouveaux nœuds) de zéro, un ou plusieurs « nouveaux » nœuds de l'expansion,
- un chemin π_s ($s$ous-arbre), éventuellement vide, dans une copie d'un sous-arbre strict de b.

Quand π_s termine bien dans un sous-arbre copié de b, on définit de plus un chemin π_v ($v$ieux nœuds) qui va de b au premier vieux nœud recopié. (Sinon π_v est choisi vide.)

Par exemple, dans la Figure 4.4.2, certains chemins vers des nœuds de l'expansion de c sont les suivants – pour plus de lisibilité, on écrit les chemins en indiquant le constructeur du nœud plutôt que son indice :

— $[a, c']$, qui est de la forme $\pi_b = [a]$, $\pi_n = [c']$, $\pi_s = []$ avec $\pi_v = []$.
— $[a, c', c'', f]$, qui est de la forme $\pi_b = [a]$, $\pi_n = [c', c'']$, $\pi_s = [f]$ avec $\pi_v = [e]$.

La présence de π_b est par hypothèse, on a supposé $\pi \geq \pi_b$ donc $\pi = \pi'.\pi_b$ pour un certain π' tel que $b' \in_{\pi'} \mathsf{join}(b_s)$. Formellement, on définit comment découper π' en une paire (π_s, π_n) en définissant une fonction $\mathsf{split\_join}\,(b_s,\ b' \in_{\pi'} \mathsf{join}(b_s))$ par induction sur b_s et inversion sur la définition de $\mathsf{join}(b_s)$:

$$\mathsf{split\_join}\left(\mathsf{var}(x),\ \frac{x \in \mathsf{Tree}(\mathcal{N}, \mathcal{V})}{b' \in_{\pi''} \mathsf{join}(\mathsf{var}(x)) = x}\right) := (\pi'', \emptyset)$$

$$\mathsf{split\_join}\left(\mathsf{node}(c,\ \_),\ \frac{}{b' \in_{\emptyset} \mathsf{node}(c,\ \_) = b'}\right) := (\emptyset, \emptyset)$$

$$\mathsf{split\_join}\left(\mathsf{node}(c,\ (a_i)_i),\ \frac{b' \in_{\pi''} \mathsf{join}(a_i)}{b' \in_{\pi''.i} \mathsf{join}(\mathsf{node}(c,\ (a_i)_i))}\right) := (\pi''_s, \pi''_n.i)$$
$$\text{où} \quad (\pi''_s, \pi''_n) = \mathsf{split\_join}\,(a_i,\ b' \in_{\pi''} \mathsf{join}(a_i))$$

Par construction nous avons $\pi_s.\pi_n = \pi'$, donc $b' \in_{\pi_s.\pi_n} \mathsf{join}(b_s)$, et nous avons aussi $b' \in_{\pi_s} \mathsf{join}(b'_s)$ et $b'_s \in_{\pi_n} b_s$ pour un sous-arbre b'_s de b_s tel que :

(1) Ou bien b' n'est pas de la forme $\mathsf{var}(x)$: le chemin π' s'arrête dans b_s avant d'arriver à un nœud $\mathsf{var}(x)$. Dans ce cas b' est un « nouveau nœud » de b_s – en particulier, on sait que l'expansion a généré au moins un nouveau nœud, sinon on aurait $b_s = \mathsf{var}(\_)$. On a $\pi_s = \emptyset$ par définition de $\mathsf{split\_join}\,(\_,\ \_)$, et on définit en plus π_v comme $\emptyset$: il n'y a pas de chemin vers un ancien nœud correspondant dans b.
(2) Ou bien le chemin π' va jusqu'à un nœud $b'_s = \mathsf{var}(b'')$ de b_s : π_n est le chemin entre b'_s et b_s, et π' continue avec un chemin $b' \in_{\pi_s} b''$, éventuellement vide, jusqu'à un b'' sous-arbre strict de b. On définit alors π_v comme le chemin de b'' dans b, non-vide puisque b'' est un sous-arbre strict.

On va conclure la preuve en trouvant un sous-arbre de a sous π_b (donc dans b) dont la mesure est strictement supérieure à celle de notre sous-arbre b' de chemin $\pi_s.\pi_n.\pi_b$ dans a'.

Chemin supérieur Nous choisissons le sous-arbre de chemin $\pi_s.\pi_v.\pi_b$ dans a, que nous appelons b'_v.

Cas (1) Dans le cas (1) ci-dessus ($\pi_s = \pi_v = \emptyset$), b' est un nouveau nœud, et b'_v est le sous-arbre de chemin $\emptyset.\emptyset.\pi_b$, donc π_b : le nœud b'_v est exactement b. Il faut montrer qu'on a $m_{\mathsf{Node}(a)}(b) > m_{\mathsf{Node}(a')}(b')$.

$$\begin{aligned}
&m_{\mathsf{Node}(a)}(b'_v)\\
&= m_{\mathsf{Node}(a)}(b)\\
&= m_{\mathsf{Path}(a)}(\pi_b) + \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b))\}\!\}
\end{aligned}$$

$$\begin{aligned}
&m_{\mathsf{Node}(a')}(b')\\
&= m_{\mathsf{Path}(a')}(\pi_n.\pi_b) + \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b'))\}\!\}\\
&= m_{\mathsf{Path}(\mathsf{join}(b_s))}(\pi_n) + m_{\mathsf{Path}(a')}(\pi_b) + \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b'))\}\!\}\\
&= m_{\mathsf{Path}(\mathsf{join}(b_s))}(\pi_n) + m_{\mathsf{Path}(a)}(\pi_b) + \{\!\{m_{\mathcal{N}}(\mathsf{constr}(b'))\}\!\}
\end{aligned}$$

L'expansion de a à a' est mesurée, donc on a $m_{\mathcal{N}}(\mathsf{constr}(b)) > m_{\mathcal{N}}(\mathsf{constr}(b'))$, qui conclut ce cas.

Cas (2) Dans le cas (2) ci-dessus, b' appartient à b'', un sous-arbre strict de b recopié par l'expansion selon b_s. π_s est le chemin de b' dans b'', et π_v le chemin (non-vide) de b'' dans b. On a :

$$\begin{array}{rcccccccc} m_{\mathsf{Node}(a)}(b'_v) & = & m_{\mathsf{Path}(a)}(\pi_b) & + & m_{\mathsf{Path}(b)}(\pi_v) & + & m_{\mathsf{Path}(b'')}(\pi_s) & + & m_{\mathcal{N}}(b'_v) \\ & & (=) & & & & (=) & & (=) \\ m_{\mathsf{Node}(a')}(b') & = & m_{\mathsf{Path}(a')}(\pi_b) & + & m_{\mathsf{Path}(b_s)}(\pi_n) & + & m_{\mathsf{Path}(b'')}(\pi_s) & + & m_{\mathcal{N}}(b') \end{array}$$

Pour conclure il faut montrer $m_{\mathsf{Path}(b)}(\pi_v) > m_{\mathsf{Path}(b_s)}(\pi_n)$. Cela est vrai car π_v est non-vide, donc $m_{\mathsf{Path}(b)}(\pi_v)$ est une somme qui contient en particulier la mesure de sa racine $m_{\mathcal{N}}(b)$, qui est strictement supérieure à la mesure de tous les nœuds de b_s puisque l'expansion est mesurée. □

B Bas niveau

B.1 Représentation des autres valeurs (que les types sommes)

Nous avons décrit dans l'introduction la façon dont les types sommes sont représentés soit comme des valeurs immédiates, soit par un *bloc* avec des méta-données en en-tête. Plus généralement, toutes les valeurs OCaml sont représentées ainsi. Par exemple, `int`, `bool` et `char` sont représentés comme des valeurs immédiates, alors que `float`, `string`, `int array` ou bien `exn` (le type des exceptions) sont représentés par des blocs.

Plus précisément :

— Les types « produits » (n-uplets (`int * bool * float`), les enregistrements et les tableaux[6]) portent l'étiquette 0, comme le premier constructeur d'un type somme.

— Les autres types de valeurs portent une étiquette spécifique, qui sert au ramasse-miette et aux fonctions de l'environnement d'exécution (comparaison polymorphe, sérialisation etc.) pour les distinguer. Par exemple, les `string` portent l'étiquette `Obj.string_tag = 252`, les objets et les types sommes extensibles (dont `exn`) portent l'étiquette `Obj.object_tag = 248`, et les types « étrangers (custom) », implémentés en C (dont `int32`, `int64`, `nativeint`), portent l'étiquette `Obj.custom_tag = 255`. Ces tags particuliers sont autour de 250, car l'étiquette d'une valeur doit tenir dans un octet, et une plage d'étiquettes entre 0 et `Obj.last_non_constant_constructor_tag = 245` est réservée pour les constructeurs non-constants et les types produits.

Ainsi, la définition suivante est acceptable

```
type tree =
  | Concat of tree * tree
  | Leaf of string [@unboxed]
```

car déboîter le constructeur `Leaf` introduit des valeurs d'étiquette `Obj.string_tag = 252`, alors que les valeurs du constructeur `Concat` portent l'étiquette 0. Mais la définition suivante serait incorrecte :

```
type tree =
  | Concat of tree * tree
  | Leaf of string array [@unboxed]
Error: This declaration is invalid, some [@unboxed] annotations introduce
       overlapping representations.
```

6. Pour les spécialistes : les tableaux de flottants peuvent aussi porter l'étiquette `Obj.double_array_tag = 254`. C'est toujours l'unique tag du type `FloatArray.t`, et c'est aussi un tag possible pour `'a array` à moins que l'option `-no-flat-float-array` ne soit utilisée.

car les `string array` portent l'étiquette 0.

B.2 Représentation de bas niveau des têtes

Dans notre implémentation, nous suivons cette représentation des valeurs en définissant la tête d'une valeur $\texttt{head}(v)$ comme un élément de $\{\texttt{Imm}, \texttt{Block}\} \times \mathbb{Z}_{\texttt{int}}$ (où $\mathbb{Z}_{\texttt{int}}$ est l'ensemble des entiers OCaml de type `int`) ainsi :

— La tête d'une valeur immédiate n est la paire $(\texttt{Imm}, n)$

— La tête d'un bloc de tag t est la paire $(\texttt{Block}, t)$.

De façon équivalente, nous pouvons décrire les têtes par un type de donnée ML (utilisé de nouveau comme un méta-langage) :

```
type imm = Imm of int [@unboxed]
type tag = Tag of int [@unboxed]
type head =
| Imm of imm
| Block of tag
```

Les ensembles de têtes qui approximent les types OCaml peuvent être représentés facilement. Dans chaque domaine (les valeurs immédiates ou les étiquettes de blocs), on représente un ensemble par soit un ensemble fini de cas possibles, soit un élément $\top$ représentant l'ensemble des valeurs possibles du domaine. Nos formes de tête de bas niveau sont décrites ainsi dans notre méta-langage :

```
type 'a shape =
  | Set of 'a list
  | Any
type head_shape = {
  imm: imm shape;
  tag: tag shape;
}
```

Pour les constructeurs emboîtés C, la représentation de bas niveau est déterminée par la présence ou non d'un argument, et leur ordre dans la déclaration de type de donnée : les constructeurs constants ont la tête `Imm n` où n est leur rang (à partir de 0) parmi les constructeurs constants, et pareillement les constructeurs non constants ont la tête `Tag n`.

Pour les autres types, on utilise la forme de tête qui approxime la représentation des valeurs par le compilateur OCaml. Voici quelques exemples de formes de têtes :

— `int` : `{imm = Any; tag = Set []}`

— `bool` : `{imm = Set [Imm 0; Imm 1]; tag = Set []}`

— `int64` : `{imm = Set []; tag = Set [Obj.custom_tag]}`

— `exn` : `{imm = Set []; tag = Set [Obj.object_tag]}`

— `('a * 'b)` : `{imm = Set []; tag = [Tag 0]}`

— `('a option)`, `('a list)` : `{imm = Set [Imm 0]; tag = Set [Tag 0]}`

— `('a -> 'b)` : `{imm = Set []; tag = Set [Tag Obj.closure_tag; Tag Obj.infix_tag]}`[7]

— `('a array)` : `{imm = Set []; tag = Set [Tag 0; Tag Obj.double_array_tag]}`[8]

— `('a Lazy.t)` : `{imm = Any; tag = Any}`[9]

7. `Obj.infix_tag` signale certaines fonctions d'un bloc de fonctions mutuellement récursives.

8. Remarque pour les spécialistes : pour les tableaux, `Obj.double_array_tag` est présent ou non selon que l'option `-no-flat-float-array` est activée. En l'absence de tableaux de flottants plats, on peut définir `type _ arr = Array : 'a array -> 'a arr [@unboxed] | Float : Floatarray.t -> float arr [@unboxed]`.

9. Remarque pour les spécialistes : l'implémentation de référence OCaml utilise une optimisation de raccourci pour les valeurs paresseuses : si on veut construire un glaçon à partir d'une valeur déjà calculée (plutôt qu'un

À l'exception des cas `'a option` et `'a list`, les exemples ci-dessus correspondent tous à des types primitifs du langage, qui déterminent l'ensemble les types primitifs $\widehat{\texttt{t}}$ dans nos jugements de calcul de la forme de tête, et leur représentation de bas niveau.

Ces informations suffisent à calculer la représentation de bas niveau d'une tête h et d'une forme de tête H, et donc de rejeter les définitions de types dont les têtes de bas niveau contiennent des doublons.

B.3 Compilation du filtrage par motif

Le compilateur OCaml stocke, dans ses environnements, des méta-données sur les déclarations de types et leurs constructeurs ; en particulier il stocke déjà, pour les constructeurs emboîtés, leur représentation (valeur immédiate ou étiquette de bloc). Nous étendons ces méta-données pour stocker, pour chaque constructeur déboîté, la forme de tête H de son paramètre de type.

Pour comprendre notre schéma de compilation, il faut connaître deux constructions déjà utilisées par la traduction du filtrage de motif en OCaml, dans la représentation intermédiaire « Lambda » :

— Une construction `switch` sur une valeur OCaml, dont les cas correspondent soit à une valeur immédiate soit à l'étiquette d'un bloc, optionnellement un « cas par défaut » pour les entrées n'ayant pas de cas spécifique.

```
type formula = True | False | And of formula * formula | Or of formula * formula
let rec eval f =
  match f with
  | True -> true
  | False -> false
  | And (f1, f2) -> eval f1 && eval f2
  | Or (f1, f2) -> eval f1 || eval f2

(* ocamlc -dlambda -dno-unique-ids test.ml *)
(switch* f
 case int 0: 1
 case int 1: 0
 case tag 0: (&& (apply eval (field 0 f)) (apply eval (field 1 f)))
 case tag 1: (|| (apply eval (field 0 f)) (apply eval (field 1 f))))
```

(`switch*` plutôt que `switch` indique l'absence de cas par défaut : la liste de cas est exhaustive pour toutes les entrées possibles.)

— Une primitive `isint` qui teste si une valeur est une valeur immédiate ou un bloc. Elle est utilisé dans le cas où au moins l'un des « domaines » du filtrage ne contient qu'un seul cas (et pas de cas par défaut).

```
let rec size f = match f with
  | True | False -> 1
  | And (f1, f2) | Or (f1, f2) -> 1 + size f1 + size f2

(* ocamlc -dlambda -dno-unique-ids test.ml *)
(if (isint f)
  1
  (+ (+ 1 (apply size (field 0 f))) (apply size (field 1 f))))
```

calcul restant à effectuer), il peut être représenté directement par cette valeur sans l'emboîter. Cela est possible à n'importe quelle type, donc les valeurs paresseuses peuvent contenir n'importe quelle tête.

Le compilateur de filtrage par motif génère ces deux constructions à partir de listes de clauses simplifiées, indicées chacune par une constructeur du type filtré (sans nommer ses variables, etc.).

En présence seulement de constructeurs emboîtés, il suffit au compilateur de regarder la représentation (`Imm n` ou `Tag n`). Si le domaine des constructeurs non-constants ne contient qu'un seul cas, il génère un `if (isint arg) ...`, avec un arbre de tests conditionnels bien choisis pour le domaine constant. Sinon, il génère une instruction `switch`, qui est préservée par le compilateur vers machine virtuelle (bytecode) (la machine virtuelle a une instruction `switch`) et transformée ultérieurement en tests bien choisis par le compilateur natif.

Quand on ajoute des constructeurs déboîtés, il faut gérer le fait qu'un `switch` peut contenir des cas correspondant à un constructeur déboîté, dont la forme de tête (stockée dans l'environnement et donc disponible pendant la compilation) indique plusieurs valeurs immédiates ou tags possibles. Par exemple, on pourrait se retrouver avec un `switch` ressemblant au suivant, en présence d'un constructeur constant (`Tag 0`), d'un constructeur déboîté de forme de tête `{imm = Set [Imm 0]; tag = Set [Tag 252]}`, et d'autres cas traités par une clause par défaut :

```
(switch* f
 case tag 0: foo
 case unboxed (int 0 | tag 252): bar
 default: foobar)
```

Notre stratégie de compilation, très simple, consiste à expanser ces cas en plusieurs cas, un par valeur de tête possible :

```
(switch f
 case tag 0: foo
 case int 0: bar
 case tag 252: bar
 default: foobar)
```

Cette transformation duplique des « actions » (les expressions du côté droit), mais le compilateur de filtrage par motif a déjà une machinerie pour garder trace de ces duplications et insérer des constructions de partage.

Enfin, un constructeur déboîté peut contenir la tête `Any` pour l'un des domaines ou les deux. Notre exemple précédent, avec la forme de tête `{imm = Any; tag = Set [Tag 252]}`, donnerait :

```
(switch* f
 case tag 0: foo
 case unboxed (int any | tag 252): bar
 default: foobar)
```

Dans ce cas, nous ne pouvons pas générer seulement la construction `switch`, qui ne permet pas de donner une condition « pour toutes les valeurs immédiates » ou « pour toutes les étiquettes de bloc ». Nous utilisons alors la construction `isint` pour traiter différemment les deux domaines :

```
(if (isint f)
  bar
  (switch* f
   case tag 0: foo
   case tag 252: bar
   default: foobar))
```

Inconvénient : motifs épars Une construction `switch` avec un cas `tag 0` et un cas `tag 252` génère en fait du code un peu décevant sur la machine virtuelle (bytecode), on obtient une instruction avec une table de sauts de 252 octets. Idéalement il faudrait repérer ces cas de tags très élevés et les gérer avec des conditionnelles avant le `switch`. En pratique, la machine virtuelle a maintenant peu d'utilisateurs, donc ce travail n'est pas une priorité.

Inconvénient : tests répétés Notre stratégie, très simple, génère parfois du code décevant, quand un constructeur déboîté a lui-même comme argument un type somme sur lequel on fait de nouveau un filtrage. Par exemple :

```
type t = Int of int | String of string
type u = Unit | Const of t [@unboxed]

let to_string u =
  match u with
  | Unit -> "()"
  | Const (Int n) -> string_of_int n
  | Const (String s) -> s
```

Ce programme va générer deux `switch` imbriqué, alors qu'un seul suffirait :

```
(switch* u
  int 0: "Unit"
  tag 0 | tag 1:
    (switch u
      tag 0: (apply string_of_int (field u 0))
      tag 1: (field u 1)))
```

Régler ce problème demande des changements plus invasifs au compilateur de filtrage par motif, sur lesquels nous n'avons pour l'instant pas travaillé. Les cas d'usage principaux pour le déboîtement de constructeurs que nous connaissons déboîtent un paramètre à un type primitif (`int`, `string`, etc.), et pas un type somme, donc ne rencontrent pas ce problème.

C Travaux futurs

C.1 Portabilité à d'autres implémentations

Un défaut de ce travail est que la décision d'accepter ou rejeter des déclarations de constructeurs déboîtés dépend de détail bas-niveau de l'implémentation du langage, qui n'ont pas de raison d'être communs à différentes implémentations.

Nous aimerions donner une sémantique de plus haut niveau, c'est-à-dire choisir un critère de non-confusion pour les formes de tête de haut niveau qui soit plus simple, moins spécifique, donc portable à plus d'implémentations variées du langage.

Mais toute simplification a pour effet d'effacer des distinctions entre types, et donc d'augmenter les approximations et d'accepter moins d'annotations de déboîtement ; elle a donc un impact en performance pour les utilisateurs de l'implémentation principale.

Nous proposons pour l'instant deux approches pour atténuer ce problème de conception :

— Par défaut, utiliser une représentation bas-niveau des formes de tête un peu moins fine, qui utilise un « plus petit dénominateur commun » entre l'implémentation de référence de OCaml et `js_of_ocaml`, l'implémentation utilisée dont la représentation des valeurs est la plus différente.

— Envisager des annotations qui indiquent explicitement qu'elles ne seront honorées que par certaines annotations, par exemple une annotation `[@unboxed native]` qui ne prend

effet qu'avec l'implémentation de référence.

La différence de représentation principale de `js_of_ocaml` se situe au niveau des types numériques : `int`, `float`, `nativeint` et `int32` ont tous la même représentation sous-jacente dans cette implémentation, ainsi que les constructeurs constants.

Pour implémenter une notion de forme de tête moins fine, qui empêche les confusions entre les types ayant la même représentation avec `js_of_ocaml`, une approche simple est d'ajouter un champ à nos formes de tête, qui garde trace de si la tête inclus un de ces types numériques :

```
type head_shape = {
  imm: imm shape;
  tag: tag shape;
  numeric: bool;
}
```

Deux formes de tête sont en conflit (ne sont pas disjointes) si elles sont en conflit selon la définition habituelle, ou si le champ `numeric` est vrai des deux côtés.

C.2 Types abstraits

Un type abstrait présent dans une implémentation (pas une interface) de module est souvent destiné à être peuplé par du code écrit en C, ou dans un autre langage utilisant la IFE (FFI) OCaml. Nous pourrions laisser l'utilisateur indiquer la forme de tête de ces types abstraits ; il faudrait lui faire confiance (le compilateur ne va pas inspecter le code C pour vérifier), mais de nombreux aspects de l'interface IFE, comme par exemple le type des primitives `external`, reposent déjà sur le fait de faire confiance à la description de l'interaction côté OCaml.

C.3 Signatures, paramètres, modularité

Si un type abstrait peut être annoté avec une forme de tête, cette construction pourrait aussi être utilisée dans les signatures de modules ; il faudra alors savoir vérifier qu'une définition concrète du type vérifie bien la forme de tête utilisée dans la signature.

Enfin, nous pourrions envisager d'annoter un paramètre de type α_i d'une définition pour lui donner une forme de tête plus précise que $\top$, en interdisant les instances du type ne respectant pas cette contrainte.

Pour être pleinement modulaire, il faudrait aussi concevoir un langage pour décrire, pour un type abstrait paramétré $\alpha\, t$, la forme de tête des instances $\tau\, t$ en fonction de la forme de tête du paramètre τ – peut-être une extension du langage des formes de tête avec des variables formelles.

C.4 Cycles bénins

Il y a une différence entre les deux déclarations cycliques suivantes, rejetées par notre critère de terminaison qui échoue avec le résultat `Cycle` :

```
type bad_cycle = Foo | Loop of bad_cycle [@unboxed]

type meh_cycle = Loop' of meh_cycle [@unboxed]
```

Dans le premier cas, `bad_cycle`, autoriser la déclaration de type aboutirait vraiment à des confusions entre `Foo`, `Loop Foo`, `Loop (Loop Foo)`, etc. : elle doit être rejetée à la fois parce qu'elle est cyclique et parce qu'elle est incorrecte.

Dans le second cas, si la définition était acceptée, elle donnerait un type vide : on n'a pas de constructeur non-récursif utilisable comme cas de base, donc aucune valeur de ce type ne peut

être construite [10]. On peut parler d'un cycle « bénin », qui pourrait être accepté sans danger.

Stephen Dolan nous a fait remarquer que certains cas de définitions acceptées peuvent se transformer en cycles bénins quand des définitions de type abstraites sont révélées.

```
type 'a foo
type weird = Loop'' of weird foo [@unboxed]
```

Ce type est accepté par notre analyse, puisqu'on suppose la forme de tête $\top$ pour `'a foo` et, par extension, pour `weird`. Mais si on apprenait ensuite que `'a foo` était en fait définit par `type 'a foo = 'a`, on se rendrait compte qu'on a accepté une définition de cycle bénin.

Nous souhaitons préserver la propriété désirable que révéler la définition d'un type abstrait préserve la validité du programme. Nous devons donc modifier notre prototype pour accepter les cycles bénins.

Il n'est pas tout à fait évident de savoir comment distinguer les deux types de cycles dans notre algorithme. Une approche un peu naïve, et relativement simple à implémenter, consiste à autoriser l'expansion deux fois d'un même constructeur de types dans une branche de calcul (et d'arrêter si une troisième expansion est essayée), au lieu d'échouer avec `Cycle` à la deuxième expansion. Si le cycle n'est pas bénin, les autres formes de tête présentes dans le cycle vont être présentes en deux exemplaires (une par extension), donc créer des confusions et faire rejeter la déclaration.

C.5 Séparabilité

OCaml accepte déjà les déclarations de type à constructeur unique avec l'annotation `[@@unboxed]`. En raison de l'optimisation de tableaux de flottants (donc hors du mode `-no-flat-float-array`), cela demande une analyse spécifique pour garantir que les types déboîtés restent « séparables », c'est-à-dire qu'ils contiennent, dans l'ensemble de leurs valeurs, soit uniquement des nombres flottants soit aucun nombre flottant. (Pour plus de détails, voir Colin, Lepigre, and Scherer (2019).)

Avec un type mono-constructeur, la seule façon de casser la séparabilité est d'utiliser un type abstrait de GADT :

```
type any = Any : 'a -> any [@@unboxed]
Error: This type cannot be unboxed because
       it might contain both float and non-float values,
       depending on the instantiation of the existential variable 'a.
       You should annotate it with [@@ocaml.boxed].
```

Le déboîtement de constructeur permet de créer des types non-séparables sans utiliser de types existentiels :

```
type non_separable = Int of int [@unboxed] | Float of float [@unboxed]
```

Il faut adapter notre travail pour rejeter ces définitions et continuer à garantir la séparabilité. Nous proposons de le faire en ajoutant un booléen à notre définition de la forme de tête, qui trace la séparabilité de la forme de tête calculée :

```
type head_shape = {
  imm: imm shape;
  tag: tag shape;
  separable: bool;
}
```

10. Remarque : `let rec x = Loop' x in x` serait accepté si `Loop'` n'était pas un constructeur déboîté, mais est rejeté quand `Loop'` est marqué `[@unboxed]`.

Un type existentiel de GADT n'est pas séparable, et l'union d'une forme de tête autorisant les flottants avec une forme de tête autorisant autre-chose donne aussi une forme de tête non-séparable. Si la forme de tête de bas niveau calculée pour une définition contient `separable = false`, il faut rejeter la définition.

Remarque. Avant cette extension, on pouvait donner une sémantique à une forme de tête comme un ensemble de valeurs OCaml. Par exemple la sémantique de `{imm = Set []; tag = Set [Tag 0]}` était l'ensemble des blocs dont le tag est 0. Avec cette extension, on trace une propriété `separable: bool` relationnelle, qui parle des ensembles de valeur OCaml et non pas propriété d'une valeur en isolation.

On doit donc interpréter une forme de tête comme un *ensemble d'ensembles* de valeurs OCaml : `{imm = Set []; tag = Set [Tag 0]}` décrit l'ensemble des ensembles qui (1) contiennent uniquement des blocs de tag 0, et (2) sont séparables.

Remarque. Nous espérons que cette analyse dynamique (par dépliement de définitions) de la séparabilité des types pourrait remplacer entièrement l'analyse statique effectuée actuellement Colin, Lepigre, and Scherer (2019).

C.6 Arité

Notre notion de tête de bas niveau ne prend en compte que l'étiquette des blocs, mais on pourrait aussi prendre en compte son arité : deux constructeurs de même étiquette mais d'arité différente sont efficacement distinguables à l'exécution.

Par exemple, le programme suivant est rejeté avec notre présentation usuelle, et serait accepté si l'arité était ajoutée à nos têtes :

```
type 'a pair = 'a * 'a
type 'a triple = 'a * 'a * 'a

type 'a foo =
  | Pair of 'a pair [@unboxed]
  | Triple of 'a triple [@unboxed]
```

C.7 GADTs

On peut calculer plus finement la forme de tête d'un GADT. Considérons par exemple le type suivant :

```
type _ t =
  | Int : int -> int t [@unboxed]
  | Bool : bool -> bool t [@unboxed]
```

Avec notre approche actuelle, cette déclaration est rejetée ; notre analyse considère qu'il y a une confusion entre, par exemple, `Int 0` et `Bool false`, qui ont la même représentation. Mais, les types `int` et `bool` étant incompatibles, ces deux valeurs ont des types différents et ne peuvent pas être confondues l'une avec l'autre.

Nous pouvons donc raffiner notre calcul des formes de tête d'une expression de type $\tau\ t$, quand $\_\ t$ est un GADT. Nous proposons de le calculer ainsi :

1. Éliminer les constructeurs de t dont le type de retour est incompatible avec τ.

   Par exemple, si on cherche la forme de tête de `int t`, on ne garde que le constructeur `Int`, mais pour `'a t` ou `foo t` où `foo` est un type abstrait on garde les deux constructeurs.

2. Grouper les constructeurs restants en « composantes connexes », des ensemble de constructeurs dont les types de retour sont compatibles entre eux.

Un constructeur peut apparaître dans plusieurs composantes. Par exemple, si on rajoutait un constructeur `Id : 'a -> 'a t [@unboxed]` à notre exemple, les composantes connexes seraient `Int, Id` et `Bool, Id`.

3. Calculer la forme de tête de chaque composante connexe.
4. Renvoyer l'union (non-disjointe) des formes de tête obtenues.

Nous pensons que ce raffinement est important en pratique pour les utilisateurs de GADTs. Des types comme `_ t` ci-dessus sont souvent utilisés pour représenter les valeurs d'un mini-langage bien typé, et le fait de pouvoir déboîter toutes ces valeurs peut avoir un impact important sur les performances d'un évaluateur.

Une idée compliquée venant rarement seule, nous n'avons pas été si surpris de découvrir que ce raffinement a un impact sur la stratégie de détection des cycles, qui devient incomplète. Considérons par exemple le GADT suivant :

```
type _ t =
| A : int t -> bool t [@unboxed]
| B : int t
```

Avec notre jugement actuel, calculer la forme de tête de `bool t` demanderait de calculer la forme de tête de `int t`, et échouerait alors avec une erreur `Cycle`. Mais si nous avions accepté d'expanser une deuxième fois le constructeur `t`, nous l'aurions fait à une instance `int t` où seul le constructeur `B` est possible, et nous aurions pu terminer correctement notre calcul.

Notre vérification dynamique de terminaison repose sur l'hypothèse que si un constructeur apparaît dans son expansion, on a nécessairement une boucle infinie d'expansion – indépendamment des instances des paramètres du constructeur. Ce n'est plus vrai avec les GADTs, si nous utilisons ce critère raffiné où les instances des paramètres influent sur l'expansion de la définition.

Heureusement, il est relativement simple de restaurer la complétude dans ce cas. Pour les GADT, au lieu de stocker seulement le constructeur de type, ici `t`, dans la trace d'expansion, nous proposons de stocker le constructeur de type *et* les constructeurs qui ont été sélectionné, car leurs types sont compatibles avec l'instance spécifique `foo t` que l'on a expansé. Si `t` apparaît de nouveau en tête de l'expression de type à une instance `bar t`, on calcule de nouveau l'ensemble des constructeurs compatibles, et on échoue avec `Cycle` seulement si `t` existe déjà avec cet ensemble dans la trace. Ce critère préserve la terminaison, car il y a un nombre fini de sous-ensembles de constructeurs possibles pour `t`, et nous pensons qu'il restaure notre résultat de complétude pour les GADTs.

Inférer et vérifier les tailles de tableaux avec des types polymorphes

Jean-Louis Colaço[1], Baptiste Pauget[1,3], and Marc Pouzet[2,3]

[1] Ansys, Toulouse, France
jean-louis.colaco@ansys.com
baptiste.pauget@ansys.com
[2] Ecole normale supérieure – PSL university, Paris, France
marc.pouzet@ens.fr
[3] INRIA, Paris, France

Résumé

Cet article présente un système de vérification et d'inférence statique des tailles de tableaux dans un langage fonctionnel strict statiquement typé. Plutôt que de s'en remettre à des types dépendants, nous proposons un système de types proche de celui de ML. Le polymorphisme sert à définir des fonctions génériques en type et en taille. L'inférence permet une écriture plus légère des opérations classiques de traitement du signal — application point-à-point, accumulation, projection, transposée, convolution — et de leur composition ; c'est un atout clef de la solution proposée. Pour obtenir un bon compromis entre expressivité du langage des tailles et décidabilité de la vérification et de l'inférence, notre solution repose sur deux éléments : (i) un langage de types où les tailles sont des polynômes multi-variés et (ii) l'insertion de points de coercition explicites entre tailles dans le programme source. Lorsque le programme est bien typé, il s'exécute sans erreur de taille en dehors de ces points de coercition. Deux usages de la proposition faite ici peuvent être envisagés : (i) la génération de code défensif aux points de coercition ou, (ii) pour les applications critiques ou intensives, la vérification statique des coercitions en les limitant à des expressions évaluables à la compilation ou par d'autres moyens de vérification formelle.

L'article définit le langage d'entrée, sa sémantique dynamique, son système de types et montre sa correction. Il est accompagné d'une implémentation en OCaml, dont le code source est accessible publiquement.

1 Introduction

On s'intéresse ici à la programmation, dans un langage de haut niveau, de systèmes temps réel soumis à des contraintes de sûreté fortes, comme ceux que l'on trouve dans l'avionique, le rail et l'automobile (e.g., commande de vol, freinage, moteur électrique). Le langage Scade [4], par exemple, est utilisé depuis plus de vingt ans pour programmer du logiciel temps réel embarqué critique. Il hérite des principes et du style de programmation de Lustre [10] : on écrit un modèle idéal synchrone dit *zero-délai* en composant des fonctions de suites et on confie au compilateur le soin de vérifier des propriétés de sûreté du programme source — par exemple qu'il est déterministe et qu'il peut être exécuté en temps et mémoire calculables statiquement — et de générer le code exécutable. La confiance donnée à son compilateur permet d'éviter d'avoir à démontrer *a posteriori* que le code est correct vis-à-vis du programme source. Elle s'appuie sur une description formelle précise de la sémantique du langage et de son compilateur. [1]

Les applications temps réel modernes combinent du code de contrôle complexe (avec un niveau d'imbrication important d'automates hiérarchiques) et du calcul intensif utilisant des

[1] Il ne s'agit cependant pas d'un compilateur prouvé formellement comme c'est le cas pour CompCert [18].

tableaux. Cet article étudie ce dernier aspect. On cherche à exprimer ces calculs dans le cadre d'un langage purement fonctionnel tel que Scade, en offrant une expressivité suffisante et dont la sûreté peut être assurée à la compilation par des moyens modulaires peu coûteux.

L'utilisation de tableaux dans un langage de programmation introduit des accès dynamiques à la mémoire dont la sûreté n'est en général pas vérifiée par le compilateur. Ces accès doivent respecter les bornes des tableaux, sous peine de produire, au mieux un arrêt à l'exécution, au pire une corruption silencieuse de la mémoire. Cette propriété peut être assurée à l'exécution de différentes façons : par l'ajout de code défensif et la levée d'exceptions (comme le fait OCaml, par exemple), en saturant la valeur de l'indice [9] ou en ajoutant une valeur par défaut [4] — deux solutions suivies dans plusieurs compilateurs synchrones (e.g., Heptagon, Lustre V6 et Scade). Il en résulte une moindre efficacité du code généré et l'introduction potentielle de code mort.[2] Assurer à la compilation la bonne utilisation des tableaux, c'est-à-dire de l'absence d'accès hors de leur domaine, apporte une garantie de sûreté et améliore les perspectives d'optimisations. C'est ce que permet notamment le langage Spark [1], construit sur le langage Ada.

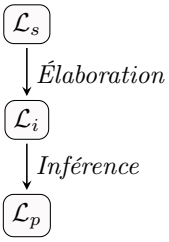
Les applications ciblées proviennent du traitement du signal et de l'IA, en particulier l'implémentation de réseaux de neurones. Les opérations de base sont celles de l'algèbre linéaire (addition et produit de tableaux multi-dimentionnels, etc.). Certains algorithmes requièrent une forme limitée de récursivité sur les tailles de tableaux : c'est le cas de la transformée de Fourier rapide, ce qui dépasse l'expressivité actuelle de Scade. D'autres, comme l'échantillonnage, nécessitent des relations non linéaires entre les tailles. Dans un langage explicitement typé, la spécification des tailles dans les types conduit à des annotations longues et répétées. Afin d'y remédier, il est nécessaire d'offrir un moyen d'inférer automatiquement les types et les tailles en s'assurant de leur bonne utilisation.

L'expression d'opérations sur des tableaux dans un langage purement fonctionnel ainsi que la vérification par typage de leur bonne utilisation sont deux questions anciennes qui ont été étudiées largement. Nous y revenons dans la partie 7. Plutôt que de s'en remettre à l'utilisation d'un système de types dépendants, la solution proposée s'appuie sur deux éléments : (i) un langage de types avec polymorphisme où les tailles sont des polynômes multi-variés. Le polymorphisme sert à définir des fonctions génériques en type et en taille. L'inférence permet une écriture plus légère des opérations classiques de traitement du signal et de leur composition. (ii) L'insertion de points de coercition explicites entre tailles dans le programme source. Lorsque le programme est bien typé, il s'exécute sans erreur de taille en dehors de ces points de coercitions. Pour des applications critiques, la vérification statique des coercitions pourra être faite a posteriori en les limitant à des expressions évaluables à la compilation ou en utilisant d'autres moyens de vérification formelle.

Cette étude est menée sur un langage purement fonctionnel et elle s'appuie sur un prototype indépendant de Scade dont le code source et les exemples sont accessibles ici : `https://gitlab.inria.fr/parkas/jfla-2022`. Ce langage ne contient pas de constructions temporelles et se concentre sur les opérations de manipulation de tableaux. L'article est organisé ainsi : la partie 2 donne un aperçu général de la contribution proposée. Le langage et sa sémantique sont définis dans la partie 3. Le système de types est défini dans la partie 4. La partie 5 détaille l'inférence des types et des tailles. Un langage de plus haut niveau est esquissé dans la partie 6. Nous discutons des travaux connexes et concluons dans la partie 7.

[2] Ce dernier point n'est pas anodin : la couverture du code, activité exigée pour la certification d'applications critiques, est plus délicate à assurer en présence de code mort, car celui-ci ne peut être couvert par aucun test.

2 Vue d'ensemble



La proposition faite dans cet article définit trois langages dont l'articulation est résumée ci-contre. Le langage de *profondeur* $\mathcal{L}_p$, explicitement typé, pour lequel une sémantique sera définie, contient les constructions nécessaires à l'introduction de tailles dans les types. Ce langage représente les tableaux par des fonctions dont le domaine est un intervalle commençant à zéro. Pour rendre les annotations optionnelles, une inférence des types et des tailles est construite sur un langage *intermédiaire* $\mathcal{L}_i$, variante de $\mathcal{L}_p$. Enfin, le langage de *surface* $\mathcal{L}_s$ apporte une couche de commodités syntaxiques. Il couvre en particulier les notations usuelles de tableaux et s'élabore vers $\mathcal{L}_i$.

Afin d'illustrer la représentation des tailles dans les types, les exemples suivants sont écrits dans $\mathcal{L}_i$. Le lecteur impatient trouvera dans la Figure 8 ces fonctions écrites dans $\mathcal{L}_s$.

Tableaux intentionnels. Dans de nombreux langages inventés pour le calcul scientifique (dont Sisal [6], pour n'en citer qu'un), les manipulations explicites d'indices sont remplacées par des opérateurs sur les tableaux, appelés aussi combinateurs [15, 13]. Ils fournissent des schémas d'accès prédéfinis toujours corrects. Le langage Scade [4] suit la même approche. Certaines primitives nécessitent cependant que des tailles coïncident, ce qui ne peut être imposé sans exprimer les tailles dans les types. Dans $\mathcal{L}_p$, le langage des tailles est séparé de celui des expressions et possède son propre ensemble de variables. Ainsi, l'application point-à-point d'une fonction (`map`), sa version binaire (`map2`) et la réduction de tableaux (`fold`), trois opérateurs qui s'expriment en Scade, auront ici le schéma de type suivant :

$$\begin{aligned}
&\texttt{val map} &&: \forall \iota.\ \forall \alpha, \beta.\ (\alpha \to \beta) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\,\alpha \to \texttt{[}\iota\texttt{]}\,\beta \\
&\texttt{val fold} &&: \forall \iota.\ \forall \alpha, \beta.\ (\alpha \to \beta \to \alpha) \to \texttt{<}\iota\texttt{>} \to \alpha \to \texttt{[}\iota\texttt{]}\,\beta \to \alpha \\
&\texttt{val map2} &&: \forall \iota.\ \forall \alpha, \beta, \gamma.\ (\alpha \to \beta \to \gamma) \to \texttt{<}\iota\texttt{>} \to \texttt{[}\iota\texttt{]}\,\alpha \to \texttt{[}\iota\texttt{]}\,\beta \to \texttt{[}\iota\texttt{]}\,\gamma
\end{aligned}$$

Le second argument de chacun de ces itérateurs, de type $\texttt{<}\iota\texttt{>}$ (lire *taille* ι), est appelé *paramètre de taille*. Il s'agit d'un entier dont la valeur est égale à celle de la *variable de taille* ι. Pour un type τ, le type $\texttt{[}\iota\texttt{]}\,\tau$ est celui des tableaux de taille ι d'éléments de type τ. L'application partielle —`map` f <42>— attend donc un tableau de taille 42 uniquement. Ces types sont universellement quantifiés en tailles (ι) et en types (α, β, γ). Contrairement à un système avec types dépendants où `map` aurait pour signature $\forall \alpha, \beta.\ (\alpha \to \beta) \to \Pi\, n\texttt{:int}.\ \texttt{[}n\texttt{]}\,\alpha \to \texttt{[}n\texttt{]}\,\beta$, les arguments de taille n'ont pas besoin d'être nommés et liés dans les types. Dans $\mathcal{L}_i$, le produit scalaire s'écrit :

$$\texttt{let dot\_product} : \_ = \lambda u\texttt{:}\_.\ \lambda v\texttt{:}\_.\ \texttt{fold}\ (\texttt{+})\ \texttt{<}\_\texttt{>}\ 0\ (\texttt{map2}\ (\texttt{*})\ \texttt{<}\_\texttt{>}\ u\ v)$$

Les trois premiers "trous" (`_`) symbolisent des types à inférer. Les deux suivants sont contenus dans la construction $\texttt{<}\cdot\texttt{>}$ qui désigne la valeur d'une taille et qu'il faudra ici inférer. Ces paramètres de taille des itérateurs n'ont pas besoin d'être spécifiés car leur valeur est contrainte par la taille des tableaux. Pour la fonction `dot_product`, l'inférence détermine le type $\forall \iota.\ \texttt{[}\iota\texttt{]int} \to \texttt{[}\iota\texttt{]int} \to \texttt{int}$. Lorsque des variables de taille n'apparaissent pas dans le type final ou l'environnement, il est nécessaire de les spécifier. Ainsi, l'expression —`fold` (+) <_> 0 (λi: [_]. 2)—, de type `int`, est rejetée car la taille du tableau —λi: [_]. 2—, dont tous les éléments valent 2, n'est pas contrainte. Cela conduirait ici à une expression de valeur arbitraire.

Supposons que l'on dispose d'une primitive `window`, illustrée dans la Figure 6, qui définit une fenêtre glissante de taille κ, de pas 1. À partir d'un tableau t de taille $\iota+\kappa-1$, elle construit une matrice de ι lignes et κ colonnes en faisant glisser une fenêtre de longueur κ le long de t.

$$\texttt{val window} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \to \texttt{[}\iota + \kappa - 1\texttt{]}\,\alpha \to \texttt{[}\iota\texttt{]}\,\texttt{[}\kappa\texttt{]}\,\alpha$$

Ici, la variable de taille ι n'apparaît pas comme paramètre de taille. Elle devra donc être déduite des types. De plus, la relation entre les tailles des tableaux utilise des opérations arithmétiques. Il est ainsi possible de définir une convolution de noyau k :

$$\texttt{let convolution} : \_ = \lambda k{:}\_.\ \lambda i{:}\_.\ \texttt{map}\ (\texttt{dot\_product}\ k)\ \texttt{<\_>}\ (\texttt{window}\ \texttt{<\_>}\ i)$$

Cette fois encore, l'inférence parvient à compléter les parties manquantes, instancier les déclarations référencées et trouve le type $\forall \iota, \kappa.\ [\kappa]\texttt{int} \to [\iota]\texttt{int} \to [\iota - \kappa + 1]\texttt{int}$.

Tableaux extensionnels. Lorsque les opérateurs intentionnels ne suffisent pas ou pour les programmer, il est nécessaire d'accéder explicitement aux éléments des tableaux. Dans $\mathcal{L}_i$, l'itérateur `map` peut être défini ainsi :

$$\texttt{let map} : \_ = \lambda f{:}\_.\ \lambda n{:}\texttt{<}\iota\texttt{>}.\ \lambda X{:}\_.\ \lambda i{:}[\iota].\ f\ (X\ i)$$

Le type du dernier argument, $[\iota]$ (lire *indice de taille* ι) représente les entiers positifs strictement inférieurs à la valeur de la taille ι. Les tableaux étant vus comme des fonctions, le type $[\eta]\tau$ est en réalité une abréviation pour le type $[\eta] \to \tau$. L'application —$X\ i$— lit donc le tableau X de taille ι à l'indice i, et respecte par construction les bornes. Ici, le paramètre n sert uniquement à introduire un argument permettant de contraindre la variable de taille ι.

Définir l'opérateur de fenêtre glissante requiert un calcul sur les indices :

$$\texttt{let window} : \_ = \lambda k{:}\texttt{<}\kappa\texttt{>}.\ \lambda X{:}[\iota + \kappa - 1]\_.\ \lambda i{:}[\iota].\ \lambda j{:}[\kappa].\ X\ (i + j \triangleright [\_])$$

L'accès au tableau X utilise une *coercition*, notée $\triangleright$. Pour un terme e et un type τ, $e \triangleright \tau$ désigne une assertion sur les tailles qui apparaissent dans le type de e, laissant uniquement au typage la vérification que le type de e et τ ont la même structure. Ici, pour que l'application soit correcte, la valeur de $i + j$ doit être bornée supérieurement par une taille. Il est inutile de la spécifier car elle peut être déduite du type de X, d'où l'écriture —$X\ (i + j \triangleright [\_])$—. Les coercitions permettent d'utiliser des opérations arithmétiques quelconques qui dépassent l'expressivité du langage des tailles, constitué de polynômes multivariés. Ainsi, l'expression —$4 \triangleright [3]$— est bien typée, mais elle échouera inévitablement. Les erreurs qui résultent des coercitions doivent être éliminées par un autre mécanisme que le typage. Dans le contexte de l'embarqué critique, il est possible de limiter les coercitions à des expressions évaluables à la compilation.

3 Langage de profondeur

L'élimination statique des erreurs liées aux tableaux s'appuie sur la définition d'un langage d'expressions, purement fonctionnel, d'ordre supérieur, explicitement typé, doté de constructions polymorphes et évalué strictement. Il a pour but de traiter tailles et types sur un pied d'égalité, tant du point de vue du système de types que de l'inférence. Ce langage est volontairement restreint aux constructions essentielles de manipulation de tableaux.

Sans recourir à un langage aussi abstrait que HM(X)[24], nous garderons de la souplesse dans la définition d'*opérateurs*. Cette paramétrisation requiert trois composants — syntaxe, sémantique et typage — qui seront définis précisément.

3.1 Syntaxe

La syntaxe de $\mathcal{L}_p$, le langage explicitement typé, est résumée dans la Figure 1. Par la suite, n désignera un entier relatif. Les tailles, les types et leurs variables seront désignés par des lettres grecque, tandis que les termes et leurs variables seront nommés par des lettres latines.

η	::=	*Tailles*	e	::=	*Termes*
	ι	variable		x	variable
	n	constante		$e\ e$	application
	η `+` η	somme		$\lambda x{:}\tau.\ e$	abstraction
	η `*` η	produit		`true` | `false`	booléen
				n	entier
τ	::=	*Types*		o	opérateur
	α	variable		e `[`η`]`	application de taille
	`<`η`>`	singleton		e `[`τ`]`	application de type
	`[`η`]`	intervalle		$\Lambda\iota.\ e$	abstraction de taille
	`int`	entier		$\Lambda\alpha.\ e$	abstraction de type
	`bool`	booléen		`fix` $x{:}\sigma = e$	point-fixe
	$\tau \to \tau$	fonction		`let` $x{:}\sigma = e$ `in` e	définition locale
				`let size` $\iota = e$ `in` e	déf. de taille
σ	::=	*Schémas de type*		`<`η`>`	taille
	τ	type simple		$e \triangleright \tau$	coercition
	$\forall\iota.\ \sigma$	quantif. de taille		`case` e `then` e `else` e	déf. par cas
	$\forall\alpha.\ \sigma$	quantif. de type		`.`	branche morte

Figure 1: Syntaxe abstraite du langage explicitement typé $\mathcal{L}_p$

$$\langle \forall\iota.\ \sigma \rangle = \overline{\langle\sigma\rangle}^{\iota} \qquad \langle \lambda x{:}\tau.\ e \rangle = \langle\tau\rangle \cup \overline{\langle e\rangle}^{x}$$
$$\langle \forall\alpha.\ \sigma \rangle = \overline{\langle\sigma\rangle}^{\alpha} \qquad \langle \texttt{fix}\ x{:}\sigma = e \rangle = \langle\sigma\rangle \cup \overline{\langle e\rangle}^{x}$$
$$\langle \Lambda\iota.\ e \rangle = \overline{\langle e\rangle}^{\iota} \qquad \langle \texttt{let}\ x{:}\sigma = e_1\ \texttt{in}\ e_2 \rangle = \langle\sigma\rangle \cup \langle e_1\rangle \cup \overline{\langle e_2\rangle}^{x}$$
$$\langle \Lambda\alpha.\ e \rangle = \overline{\langle e\rangle}^{\alpha} \qquad \langle \texttt{let size}\ \iota = e_1\ \texttt{in}\ e_2 \rangle = \langle e_1\rangle \cup \overline{\langle e_2\rangle}^{\iota}$$

Figure 2: Variables libres des objets de la syntaxe. $\overline{S}^{v}$ désigne l'ensemble $S \setminus \{v\}$. Dans les autres cas, $\langle o \rangle$ est construit inductivement par union des variables libres des sous-objets.

Espaces de noms, variables libres et substitutions. Les tailles, types et termes possèdent chacun leur propre espace de noms de variables, respectivement $\mathcal{V}_\eta$, $\mathcal{V}_\tau$ et $\mathcal{V}_e$, supposés disjoints. Pour un objet de la syntaxe o, $\langle o \rangle \in \mathcal{V}_\eta \cup \mathcal{V}_\tau \cup \mathcal{V}_e$ désigne l'ensemble des *variables libres*, défini dans la Figure 2 et $\langle o \rangle_e$ (*resp.* $\langle o \rangle_\eta$, $\langle o \rangle_\tau$) l'ensemble $\langle o \rangle \cap \mathcal{V}_e$ (*resp.* $\mathcal{V}_\eta$, $\mathcal{V}_\tau$). Un objet o est *clos* si $\langle o \rangle = \emptyset$. Les substitutions sont définies pour les différentes catégories syntaxiques et les couples variable/élément. Elle seront notées uniformément $\cdot\{\cdot/\cdot\}$. Ainsi, $e\{\eta/\iota\}$ désigne la substitution dans le terme e de toutes les occurrences libres de la variable de taille ι par la taille η, y compris dans les tailles et types contenus dans e.

Tailles, types et schémas. L'ensemble des tailles $\mathcal{P} := \mathbb{Z}[\mathcal{V}_\eta]$ est celui des polynômes multivariés à coefficients entiers. Cette classe d'expressions arithmétiques présente l'avantage d'admettre pour forme normale une somme pondérée de produits de variables, ce qui permet de comparer symboliquement des tailles structurellement différentes (*eg.* $(\iota-1)^2-1 = \iota*(\iota-2)$). Pour les applications visées, ce langage semble suffire à exprimer les propriétés des tailles des opérations de tableaux.

En plus des types usuels du λ-calcul simplement typé (variables, types entier et booléen, fonctions), deux *raffinements* du type entier, au sens de [8, 31, 20, 7], introduisent les tailles dans

les types. Pour un entier n, $<n>$ désigne le singleton $\{n\}$, tandis que $[n]$ représente l'intervalle $[\![0, n-1]\!]$. Pour une taille générique, ces types dépendent de la valuation des variables de taille. Le langage des types ne contient pas de constructeur de type pour les tableaux : le type «tableau de taille η d'éléments de type τ» est représenté par $[\eta] \to \tau$, noté également $[\eta]\tau$[3]. Le type singleton $<\eta>$, ne contenant qu'une seule valeur, est utilisable comme expression [30, 7]. Il aide également l'inférence par l'introduction de contraintes sur les tailles (partie 5).

Suivant les restrictions introduites dans les langages de la famille ML, les types sont quantifiés universellement et doivent être en forme prénexe selon des variables de taille ($\forall\iota.\ \sigma$) et de type ($\forall\alpha.\ \sigma$). Par la suite, de telles constructions similaires pour des tailles et des types (quantification, abstraction, application ou substitution) seront factorisées en écrivant : $\forall\iota|\alpha.\ \sigma$.

Termes. Le cœur *a la* ML de notre langage fonctionnel (variables, abstractions, applications et déclarations locales polymorphes) est enrichi d'un opérateur de point-fixe polymorphe —`fix` $x{:}\sigma = e$—, extension très étudiée [21, 22] qui importe particulièrement pour la définition de fonctions récursives sur des tableaux (voir l'annexe D).

Les seules valeurs de base sont les entiers et les booléens. La composante syntaxique de la paramétrisation des opérateurs est constituée d'un ensemble d'identifiants noté $\mathcal{O}$, supposé disjoint de celui des variables. Cet ensemble contient *a minima* les opérations arithmétiques entières (en particulier l'addition et la multiplication), les comparaisons entières et une égalité polymorphe. Ces opérateurs seront notés avec les symboles usuels, et leur application écrite en forme infixe.

Afin de définir une notion de portée des variables de taille et de type, le polymorphisme est rendu explicite par des abstractions de taille et de type (*généralisations*) —$\Lambda\iota|\alpha.\ e$— et les applications correspondantes (*instanciations*) —$e\ [\eta|\tau]$—.

Bien que séparés, les mondes des tailles et des expressions communiquent. La valeur d'une taille peut être utilisée comme terme —$<\eta>$—, tandis que l'utilisation d'un entier comme taille s'exprime par la quantification existentielle d'une variable de taille —`let size` $\iota = e_1$ `in` e_2—.

Lorsqu'il est impossible de vérifier formellement un raffinement (*eg.* égalité de tailles), une coercition —$e \triangleright \tau$—, semblable à celle de Futhark [12], insère une vérification *post-typage* des raffinements. Enfin, une définition par cas —`case` e_1 `then` e_2 `else` e_3—, qui peut être rendue partielle par un terme d'erreur —.— sur l'une des branches, conditionne l'évaluation d'expressions, par exemple pour les cas terminaux de fonctions récursives.

Précédence et notations. Les expressions seront implicitement parenthésées par les règles suivantes : les abstractions et généralisations ont la plus faible précédence, suivies des introductions de variables (de taille, de type ou d'expression). Viennent ensuite la définition par cas, puis la coercition et enfin les application et instanciations, associatives à gauche. Ainsi, $e_1\ e_2\ e_3$ désigne $(e_1\ e_2)\ e_3$ et $\lambda x{:}\tau.\ e\ x \triangleright \tau' =$ a le sens de $\lambda x{:}\tau.\ ((e\ x) \triangleright \tau')$.

3.2 Sémantique à grands pas

La sémantique de $\mathcal{L}_p$ —$e \rightsquigarrow v^\star$— associe à certaines expressions closes une *valeur*. L'ensemble des valeurs (noté $\mathcal{V}$) est le sous-ensemble des expressions présenté dans la Figure 3, augmenté d'une valeur spécifique $\star$ (lire «erreur») qui représente une erreur *post-typage* explicite. Ces erreurs correspondent à des propriétés que l'on peut vérifier statiquement par des mécanismes autres que le typage, en imposant par exemple aux expressions dont elles dépendent d'être évaluables à la compilation.

[3] Utilisée dans Futhark[13], cette notation convient aux tableaux multidimensionnels : $[\eta_1, \eta_2]\tau := [\eta_1]\,[\eta_2]\,\tau$

Opérateurs. Le volet sémantique de la paramétrisation des opérateurs, fortement inspiré de celle des destructeurs de HM(X)[25], comprend une fonction d'arité $\nu : \mathcal{O} \longrightarrow \mathbb{N}^*$ ainsi qu'une relation de réduction $\overset{o}{\rightsquigarrow}$.

En plus des constructeurs (n, `true`, `false`, λ, Λ), l'ensemble des valeurs est constitué des applications partielles d'opérateurs instanciés (la classe *Instance d'opérateur*) dont les arguments ont été réduits à des valeurs. Pour une instance d'opérateur p, l'arité de l'opérateur instancié sera notée $\nu(p)$. Les applications totales d'opérateurs sont réduites à l'aide de $\overset{o}{\rightsquigarrow}$. Cette sémantique des opérateurs est supposée totale sur le domaine de valeurs défini par leur type (voir partie 4), et ne peut pas produire $\star$. Cela exclut par exemple la détection des divisions par zéro, ou des racines carrées de nombres négatifs. Comme dans [15], seules les erreurs *structurelles* sont représentées par $\star$.

p	$::=$	$o \mid p\ [\eta] \mid p\ [\tau]$	*Instance d'op.*
v	$::=$		*Valeur*
	$\mid$	$n \mid$ `true` $\mid$ `false`	constantes
	$\mid$	$p\ v_1\ \ldots\ v_i, \quad i < \nu(p)$	app. partielle
	$\mid$	$\lambda x{:}\tau.\ e$	abstraction
	$\mid$	$\Lambda\alpha.\ e$	abstraction de type
	$\mid$	$\Lambda\iota.\ e$	abstraction de taille
$v^\star$	$::=$	$v \mid \star$	*Valeur optionnelle*

Figure 3: Valeurs et expressions

Erreurs et réduction. Le système de réduction associé est présenté dans la Figure 4. Afin d'alléger la formalisation, les règles *étoilées* (de la forme $\frac{\_\ \ldots\ \_}{\star}$) propagent implicitement les erreurs : si l'un des antécédents est réduit à $\star$, l'expression entière est réduite elle aussi à $\star$. Cela correspond à l'ajout de règles partielles dont la dernière prémisse et l'expression totale sont réduites à $\star$, (cette construction est détaillée dans l'annexe A).

Propriétés. Cette sémantique impose une évaluation stricte : les arguments des applications de fonctions ou d'opérateurs sont réduits avant d'être substitués (règles β-RED et TOTOP). Les règles relatives à la définition par cas (TCASE, FCASE) n'évaluent, elles, qu'une seule des deux branches ce qui permet d'empêcher la propagation d'erreurs depuis la branche non choisie. Associée à la construction de branche morte, dont la sémantique produit $\star$ (règle ERR), cette définition par cas permet d'exprimer des gardes : la sémantique de `case` P `then` e `else` . n'évaluera l'expression e qui si le prédicat booléen P est valide.

Ce système de réduction est partiel : certains termes ne peuvent pas être réduits, on parlera de termes *bloqués*. En effet, les règles β-RED, INST, CTRUE, CFALSE et LETS restreignent la forme de la valeur obtenue par la réduction de l'une de leurs sous-expressions. Enfin, les règles de la sémantique des coercitions requièrent une adéquation de la valeur et du type.

Raffinements et coercitions. La présence de types raffinés empêche la définition d'une sémantique indépendante des types. En effet, ces raffinements discriminent des valeurs de même forme (type de base sous-jacent identique) et doivent être vérifiés en plusieurs endroits. Par exemple, le terme $—(\lambda x : [4].\ e)\ 8—$ ne devrait pas avoir de sémantique car l'argument $—8—$ n'est pas du type de la variable $x : [4]$. De manière générale, il est nécessaire, lorsque des substitutions sont utilisées dans la réduction, de s'assurer que la valeur remplaçante satisfait le raffinement du type de la variable substituée. Cela concerne les règles β-RED, LET et FIX.

Pour restreindre les occurrences de $\star$ aux coercitions *explicites*, deux degrés sont distingués : les coercitions *faibles* $—\triangleright_w—$ (que l'on note simplement $\triangleright$ dans la syntaxe de $\mathcal{L}_p$ et $\mathcal{L}_i$), peuvent échouer et produire $\star$ tandis que les coercitions *fortes* $—\triangleright_s—$ (introduites uniquement par les règles de typages β-RED, LET et FIX) n'ont de sémantique que si elles réussissent. Étant fixé un degré k, les règles de réduction des coercitions $—v \triangleright_k \sigma \rightsquigarrow v^*—$ vérifient dans les cas de

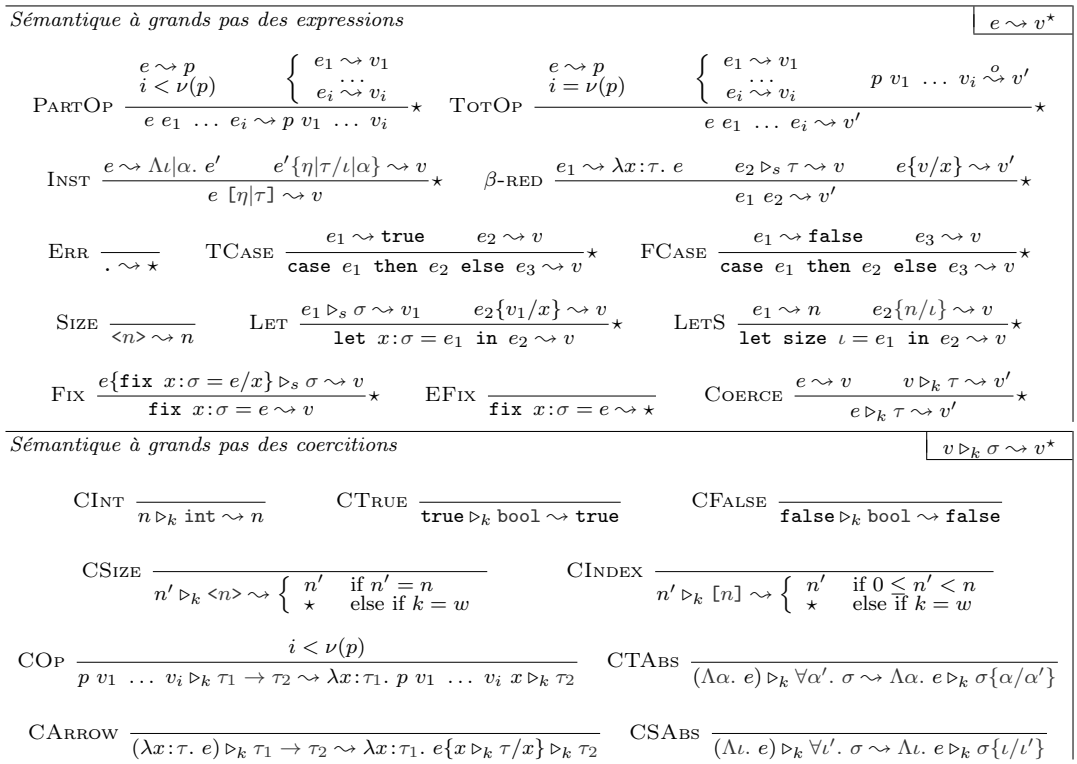


Figure 4: Règles de déduction de la sémantique de $\mathcal{L}_p$

base le respect des raffinements (CSize, CIndex), et ne peuvent échouer que si $k = w$. Pour les abstractions, applications partielles d'opérateurs et généralisations, les vérifications sont retardées à l'appel ou instanciation.

Rappelons notre objectif initial d'empêcher les accès incorrects aux tableaux. Par leur encodage sous la forme de fonctions, une valeur de type $[\eta]\tau$ est de la forme $\lambda i{:}[\eta].\ e$. L'accès aux éléments est représenté par l'application, qui vérifie les raffinements et assure donc le respect des bornes. L'existence d'une sémantique garantit donc la correction des accès.

Récursion et convergence. Pour les distinguer des termes bloqués, ceux qui divergent doivent posséder une sémantique[4]. Pour cela, la règle EFix arrête la réduction par la levée d'une erreur et peut s'appliquer à n'importe quelle occurrence d'un point-fixe. Elle correspond à des récursions statiques, déroulées à la compilation (un nombre limité de fois). Dans l'embarqué critique, seuls ces usages de récursions sont autorisés afin de borner statiquement la pile et le temps d'exécution.

Les règles Fix et EFix rendent la sémantique de $\mathcal{L}_p$ non déterministe, car elles permettent d'interrompre des récursions bornées. Il est aisé de vérifier que les autres règles sont dirigées par la syntaxe, ce qui permet d'énoncer une propriété de déterminisme affaibli, prouvée dans l'annexe A : il existe au plus une valeur v ($v \neq \star$) telle que $e \rightsquigarrow v$.

[4]Dans les sémantiques à petits pas, les termes divergents produisent une suite infinie de réductions élémentaires, ce qui les distinguent des termes bloqués, dont la réduction s'achève sans trouver de valeur.

4 Système de types

Une discipline de types, inspirée de celle d'Hindley et de Milner [14] (HM), sélectionne des termes pour lesquels une sémantique existe, c'est à dire des valeurs ou des expressions pouvant être réduites, possiblement à $\star$.

4.1 Jugements et déductions

Environnement. Les expressions sont typées dans un environnement Γ défini comme un triplet $(\Gamma_\eta, \Gamma_\tau, \Gamma_e)$ où Γ_η (*resp.* Γ_τ) collecte les variables de taille (*resp.* type) définies et Γ_e associe à certaines variables d'expression un schéma de type. Dans la suite, nous supposerons que les termes ont été renommés de façon à éviter les conflits de noms. L'environnement n'est donc pas ordonné et nous noterons uniformément $\Gamma, x : \sigma$; Γ, ι ou Γ, α l'extension de la composante appropriée, laquelle ne contient pas la variable insérée x, ι ou α.

Jugements. Le jugement de typage —$\Gamma \vdash e : \sigma$— se lit «dans l'environnement Γ, l'expression e a le schéma de type σ». Cette relation suppose implicitement la bonne formation de σ et de e, c'est à dire que leurs variables libres doivent être définies dans Γ. La relation de sous-typage —$\tau_1 <: \tau_2$— introduit ou élimine (selon la variance) des raffinements. Celle de coercition —$\sigma_1 \approx \sigma_2$— définit une équivalence de types ignorant les tailles. Dans la suite, les environnements vides seront omis. Les règles de déductions de ces relations sont présentées dans la Figure 5.

Opérateurs. Dernière facette de notre paramétrisation, chaque opérateur o se voit attribuer un schéma de type, noté $\sigma(o)$. Celui-ci permet de définir l'ultime hypothèse sur le domaine de la sémantique $\overset{o}{\rightsquigarrow}$: pour chaque opérateur instancié p, elle doit être définie pour toutes les valeurs $v_1, \ldots, v_{\nu(p)}$ telles que $\vdash p\ v_1\ \ldots\ v_{\nu(p)} : \tau$ et vérifier $p\ v_1\ \ldots\ v_{\nu(p)} \overset{o}{\rightsquigarrow} v \implies \vdash v : \tau$.

Déductions. Dans $\mathcal{L}_p$, le polymorphisme est explicite. Les constructions associées sont typées à l'aide des règles Gen et Inst. Elles généralisent et instancient en taille et en type.

La quantification existentielle de variables de taille, typée par la règle LetSize est locale car la variable introduite ne peut pas apparaître librement dans le type de l'expression, puisque celui-ci est bien formé dans l'environnement de typage non étendu.

La règle Fix permet une récursion polymorphe. En effet, la variable introduite possède un schéma de type. Elle pourra donc être instanciée différemment lors de ses utilisations récursives.

L'égalité de types, nécessaire par exemple dans les règles SRefl et CRefl, requiert l'identité formelle des tailles qui y apparaissent. Ainsi, les types $[\iota]$ et $[2 - \iota]$ sont différents, bien que toute instanciation telle que $\iota = 1$ les rende identiques.

Raffinements, sous-typage et coercitions. Le sous-typage permet uniquement d'insérer ou de supprimer des raffinements, selon la variance. Ainsi, $[\eta]$ <: `int` et `int` $\to \alpha <: [\eta]\alpha$, mais la déclaration suivante est mal typée, bien que pour toute taille η, $\eta < \eta + 1$:

$$\texttt{let drop\_last} : \forall\iota.\ \forall\alpha.\ [\iota+1]\alpha \to [\iota]\alpha = \Lambda\iota.\ \Lambda\alpha.\ \lambda X : [\iota+1]\alpha.\ X \qquad \text{(Erreur)}$$

Cette restriction permet à l'inférence des tailles de résoudre des égalités plutôt que des inégalités. Les raffinements sont également modifiés par les coercitions, qui vérifient : (i) que des entiers satisfont des raffinements (règles CSize et CIndex), ou (ii) que les tailles effectives, c'est à dire après instanciation, coïncident avec les tailles attendues (règle Coerce et relation $\tau_1 \approx \tau_2$).

Typage des expressions $\Gamma \vdash e : \sigma$

$$\textsc{Var}\ \frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \textsc{Bool}\ \frac{}{\Gamma \vdash \texttt{true}|\texttt{false} : \texttt{bool}} \qquad \textsc{Int}\ \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \textsc{SubType}\ \frac{\Gamma \vdash e : \tau \qquad \tau <: \tau'}{\Gamma \vdash e : \tau'}$$

$$\textsc{Size}\ \frac{}{\Gamma \vdash <\eta> : <\eta>} \qquad \textsc{Op}\ \frac{}{\Gamma \vdash o : \sigma(o)} \qquad \textsc{Err}\ \frac{}{\Gamma \vdash \_ : \sigma} \qquad \textsc{Case}\ \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{case}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}$$

$$\textsc{CSize}\ \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash e \triangleright <\eta> : <\eta>} \qquad \textsc{CIndex}\ \frac{\Gamma \vdash e : \texttt{int}}{\Gamma \vdash e \triangleright [\eta] : [\eta]} \qquad \textsc{Coerce}\ \frac{\Gamma \vdash e : \tau \qquad \tau' \approx \tau}{\Gamma \vdash e \triangleright \tau' : \tau'}$$

$$\textsc{Let}\ \frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x{:}\sigma = e_1\ \texttt{in}\ e_2 : \tau} \qquad \textsc{LetSize}\ \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma, \iota \vdash e_2 : \tau}{\Gamma \vdash \texttt{let size}\ \iota = e_1\ \texttt{in}\ e_2 : \tau}$$

$$\textsc{Abs}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x{:}\tau.\ e : \tau \to \tau'} \qquad \textsc{Fix}\ \frac{\Gamma, x : \sigma \vdash e : \sigma}{\Gamma \vdash \texttt{fix}\ x{:}\sigma = e : \sigma} \qquad \textsc{Gen}\ \frac{\Gamma, \iota|\alpha \vdash e : \sigma}{\Gamma \vdash \Lambda\iota|\alpha.\ e : \forall\iota|\alpha.\ \sigma}$$

$$\textsc{App}\ \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau} \qquad \textsc{Inst}\ \frac{\Gamma \vdash e : \forall\iota|\alpha.\ \sigma}{\Gamma \vdash e\ [\eta|\tau] : \sigma\{\eta|\tau/\iota|\alpha\}}$$

Sous-typage $\tau_1 <: \tau_2$

$$\textsc{SSize}\ \frac{}{<\eta> <: \texttt{int}} \qquad \textsc{SIndex}\ \frac{}{[\eta] <: \texttt{int}}$$

$$\textsc{SRefl}\ \frac{}{\tau <: \tau} \qquad \textsc{SArrow}\ \frac{\tau_2 <: \tau_1 \qquad \tau_1' <: \tau_2'}{\tau_1 \to \tau_1' <: \tau_2 \to \tau_2'}$$

Coercition des tailles $\tau_1 \approx \tau_2$

$$\textsc{CSize}\ \frac{}{<\eta_1> \approx <\eta_2>} \qquad \textsc{CIndex}\ \frac{}{[\eta_1] \approx [\eta_2]}$$

$$\textsc{CRefl}\ \frac{}{\tau \approx \tau} \qquad \textsc{CArrow}\ \frac{\tau_1 \approx \tau_1' \qquad \tau_2 \approx \tau_2'}{\tau_1 \to \tau_2 \approx \tau_1' \to \tau_2'}$$

Figure 5: Système de types de $\mathcal{L}_p$

Les constantes entières, de type `int` ne peuvent être utilisées comme indices. Il est nécessaire, pour obtenir des valeurs de type $[\eta]$ d'utiliser une coercition. Le système de type présenté ne contraint pas le signe des tailles. Des tableaux de taille négative peuvent donc être déclarés. Ils ne peuvent cependant pas être lus (et se comportent donc comme des tableaux vides), car une coercition vers un indice de taille négative échouera systématiquement.

Calculs sur les tailles. Les opérations polynomiales peuvent être traitées spécifiquement par la règle ci-contre (ou $\bullet$ désigne +, - ou $*$). Cela permet de construire des expressions de type $<\eta>$ à partir de paramètres de tailles.

Typage des expressions (extension) $\Gamma \vdash e : \sigma$

$$\textsc{Add,Sub,Mul}\ \frac{\Gamma \vdash e_1 : <\eta_1> \qquad \Gamma \vdash e_2 : <\eta_2>}{\Gamma \vdash e_1 \bullet e_2 : <\eta_1 \bullet \eta_2>}$$

4.2 Propriétés du typage

Équivalence de types. Suivant la définition de [22], un schéma de type σ' est une *instance générique* de σ s'il existe une substitution des variables liées de σ ne capturant aucune de ses variables libres permettant d'obtenir σ', à l'ordre des quantifications près. Lorsque deux schémas de type sont des instances génériques l'un de l'autre, ils sont dits *équivalents*. Pour les types de ML, dont l'égalité est structurelle, cette relation coïncide avec un renommage des variables liées. Lorsque les tailles sont considérées, leur égalité extensionnelle élargit cette équivalence. Les types de $\mathcal{L}_s$ ne sont donc pas uniques : bien qu'ils soient structurellement différents, les deux schémas de type suivants sont équivalents.

$$\texttt{val concat} : \forall\iota_1, \iota_2.\ \forall\alpha.\ [\iota_1]\alpha \to [\iota_2]\alpha \to [\iota_1 + \iota_2]\alpha$$
$$\texttt{val concat} : \forall\iota_1, \iota_2.\ \forall\alpha.\ [\iota_1]\alpha \to [\iota_2 - \iota_1]\alpha \to [\iota_2]\alpha$$

Types principaux. L'utilisation de polynômes comme langage de tailles permet des contraintes non linéaires pour lesquelles plusieurs solutions peuvent exister. Cela empêche l'existence d'un type le plus général. Ainsi, la fonction (sans intérêt) suivante, où ι doit être inférée

$$\texttt{let zero} : <\iota> \to <0> = \lambda n{:}<\iota>.\ (n-1) * (n-2)$$

est correctement typé pour $\iota = 1$ ou $\iota = 2$. Les deux types de `zero` qui en résultent ($<1> \to <0>$ et $<2> \to <0>$) ne sont ni équivalents ni ne sont des instances génériques d'un même schéma de type. L'inférence, détaillée dans la partie 5, rejettera une telle définition.

Correction. Ce système de types vérifie deux propriétés : le type est préservé par réduction et le typage est correct vis-à-vis de la sémantique. Elles s'expriment formellement par :

$$\forall e\ \sigma\ v,\quad \left.\begin{array}{c}\vdash e : \sigma \\ e \rightsquigarrow v\end{array}\right\} \Longrightarrow \vdash v : \sigma \qquad \text{(Préservation)}$$

$$\forall e\ \sigma,\ \ \vdash e : \sigma \implies \exists v^\star \in \mathcal{V},\ e \rightsquigarrow v^\star \qquad \text{(Correction)}$$

La preuve de ces propriétés (voir l'annexe A) utilise une extension générique de sémantiques à grands pas développée dans [5]. Comme énoncé sous-partie 3.2, l'existence d'une sémantique garantit une exécution sans erreur (en particulier aux accès de tableaux) à l'exception des valeurs $\star$ produites par certaines constructions. Le typage implique donc la même propriété.

5 Inférence

Les annotations de types deviennent encombrantes lorsque les expressions de tailles grossissent ; il est souhaitable de les inférer. Cependant, au vu de la richesse de notre système de types, impossible d'étendre ici les propriétés fondamentales dont jouit l'inférence pour la discipline de types de HM [14] (existence de types principaux, complétude et correction). En effet, la récursion polymorphe seule rend l'inférence indécidable [11]. De plus, l'utilisation de raffinements, réminiscence de types dépendants [30], limite également les perspectives de reconstruction des types. Enfin, la manipulation d'expressions arithmétiques non linéaires laisse, elle aussi, peut d'espoir de solution totale au problème d'inférence. L'inférence cherchera seulement un type *pertinent*, qu'il sera possible d'ajuster par l'ajout d'annotations.

5.1 Aperçu de l'inférence

Comme le font différents travaux sur l'inférence de types étendus [30, 17, 26], notre stratégie de reconstruction de types procède graduellement : (i) Les types (sans raffinement) sont inférés, par unification structurelle. (ii) Les raffinements du type entier sont sélectionnés par propagation locale des annotations. (iii) Les tailles sont inférées par résolution de contraintes.

Langage $\mathcal{L}_i$. Les termes implicitement typés sont exprimés dans le langage $\mathcal{L}_i$, variante du langage $\mathcal{L}_p$. Les *trous* (`_`) laissés à la place de tailles ou de types représentent des variables libres à inférer. Les termes sont implicitement généralisés aux `let` et les instanciations explicites sont interdites. Les généralisations peuvent apparaître, mais sont dotées d'une sémantique légèrement différente de celles de $\mathcal{L}_p$ (voir la sous-partie 5.3).

Inférence par collecte de contraintes. Les trois passes d'inférence ont été implémentées selon le même schéma : le terme est parcouru afin d'en extraire des contraintes dont le système

résultant est résolu en certaines constructions du langage [17]. Cela permet de traiter de façon similaire les tailles et les types, tant pour l'unification que pour la généralisation et l'instanciation. La principale limitation de cette stratégie tient à une localisation des erreurs plus approximative. Si elle n'apporte rien à l'inférence des types[5] pour laquelle l'ordre des contraintes n'a pas d'importance, la collecte des contraintes de taille offre une vue d'ensemble du système qui aide à leur résolution.

Les contraintes provenant de contextes de sous-typage sont distinguées de celles qui émanent des coercitions. Les variables libres, qui proviennent des *trous*, sont également collectées. Aux points de généralisation (`let`, abstractions de taille ou type), le système de contraintes est résolu, par des processus d'unification propres à chaque phase, présentés ci-dessous.

5.2 Résolution de contraintes

(i) Types. Première étape du processus d'inférence, les contraintes sont résolues pour les types sans raffinements. Pour cela, un type $\overline{\texttt{int}}$ désigne le type entier dont le raffinement reste à déterminer. La résolution de contraintes procède par unification structurelle des types, qui échoue selon les conditions habituelles (*e.g.* inégalité des constructeurs de tête, types cycliques). Ici, les contraintes provenant de sous-typage et de coercitions sont traitées uniformément.

À l'issue de cette passe d'inférence, les occurrences du type $\overline{\texttt{int}}$ sont remplacées par des variables de type fraîches, qui seront inférées dans la phase suivante.

(ii) Raffinements. Cette seconde phase choisit l'un des types `int`, `<_>` ou `[_]` pour chaque variable de type introduite à la place de $\overline{\texttt{int}}$ par l'inférence des types, sans en inférer les tailles. Pour illustrer la résolution de contraintes de raffinement, la définition suivante applique une négation booléenne à chaque élément d'un tableau :

$$\texttt{let not\_array} : \_ = \lambda X : \_.\ \lambda i : \texttt{[\_]}.\ \texttt{not}\ (X\ i)$$

L'inférence des types produit le terme suivant, dont les contraintes sont résumées ci dessous :

$$\texttt{let not\_array} : (\alpha \to \texttt{bool}) \to \beta \to \texttt{bool} = \lambda X : \gamma \to \texttt{bool}.\ \lambda i : \texttt{[\_]}.\ \texttt{not}\ (X\ i)$$

$$\texttt{[\_]} <: \gamma \tag{1}$$

$$(\gamma \to \texttt{bool}) \to \texttt{[\_]} \to \texttt{bool} <: (\alpha \to \texttt{bool}) \to \beta \to \texttt{bool} \tag{2}$$

La Contrainte 1 provient de l'application qui impose que le type de l'argument i soit un sous-type de celui du domaine de X. La Contrainte 2 est induite par la déclaration qui requiert que le type de l'expression soit un sous-type de l'annotation. Par la règle SArrow cette dernière se décompose en deux contraintes non triviales : $\gamma <: \alpha$ et $\beta <: \texttt{[\_]}$.

Les contraintes de la forme $\alpha <: \texttt{<\_>}$; $\alpha <: \texttt{[\_]}$ ou $\texttt{int} <: \alpha$ imposent le raffinement de la variable α. Elles sont considérées en premier afin de réduire le nombre de variable à inférer. Dans l'exemple, seule la variable β peut être définie de cette façon.

Remplacer les variables restantes par `int` produirait un terme bien raffiné dont le type ne serait pas aussi général qu'attendu. Pour résoudre les contraintes restantes (ici, $\texttt{[\_]} <: \gamma$ et $\gamma <: \alpha$), les variables qui n'apparaissent à droite ou à gauche que d'une seule inégalité sont remplacées par l'autre membre. Cette propagation locale de raffinements permet d'inférer le type `[_]bool` $\to$ `[_]bool` pour la fonction `idBArray`, sans laquelle elle se verrait attribuer le type moins général (`int` $\to$ `bool`) $\to$ `[_]bool`, et ne serait pas utilisable avec un tableau.

[5] Les algorithmes d'unification destructrice obtiennent les mêmes résultats et localisent mieux les erreurs.

(iii) Tailles. Bien qu'il ne soit pas possible de résoudre formellement un système d'équations polynomiales entières, les relations entre les tailles de tableaux dans les fonctions sont en pratique simples. Pour présenter les stratégies de résolution les primitives suivantes seront utilisées :

$$\texttt{val window} : \forall \iota, \kappa.\ \forall \alpha.\ \texttt{<}\kappa\texttt{>} \to [\iota + \kappa - 1]\,\alpha \to [\iota]\,[\kappa]\,\alpha$$
$$\texttt{val sample} : \forall \iota, \delta.\ \forall \alpha.\ \texttt{<}\delta\texttt{>} \to [(\iota - 1) * \delta + 1]\,\alpha \to [\iota]\,\alpha$$

La fenêtre glissante, présentée ci-dessus, contraint la taille des tableaux d'entrée et de sortie de façon à imposer une lecture complète de l'entrée. De manière similaire, `sample` extrait un élément sur δ, en lisant le premier et le dernier. La condition de divisibilité est ici encodée dans la taille de l'entrée. La composition de ces fonctions définit l'opérateur d'échantillonnage général `pack`, illustré dans la Figure 6, qui sélectionne ι fenêtres de taille κ, régulièrement espacées et couvrant les extrémités du tableau d'entrée de taille ν. Pour garantir cette propriété, le pas δ entre les fenêtres doit vérifier la relation $\delta * (\iota - 1) = \nu - \kappa$. Étant donnée la définition implicitement typée :

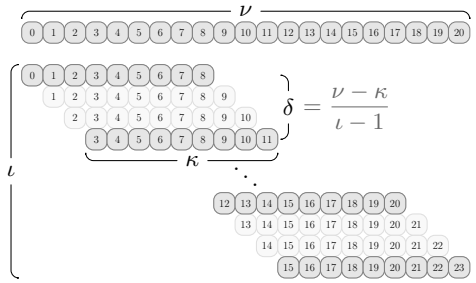


Figure 6: Opérateur `pack`

$$\texttt{let pack} : \_ = \lambda x\texttt{:}\_.\ \texttt{sample <\_> (window <\_> } x\texttt{)}$$

le terme suivant, instancié explicitement, est produit par les deux premières phases d'inférence :

$$\texttt{let pack} : [\nu]\,\alpha \to [\iota]\,[\kappa]\,\alpha = \lambda x\texttt{:}\,[\nu_i]\,\alpha.$$
$$\texttt{sample } [\iota_s]\ [\delta_s]\ [[\kappa'_w]\,\alpha]\ \texttt{<}\kappa_1\texttt{>}\ (\texttt{window } [\iota_w]\ [\kappa_w]\ [\alpha]\ \texttt{<}\kappa_2\texttt{>}\ x)$$

Il est associé aux contraintes de taille résumées ici sous la forme d'égalité pour plus de lisibilité :

$$\kappa_w = \kappa_2 \qquad \nu_i = \iota_w + \kappa_w - 1 \qquad \kappa_w = \kappa'_w \qquad \kappa'_w = \kappa$$
$$\delta_s = \kappa_1 \qquad \iota_w = (\iota_s - 1) * \delta_s + 1 \qquad \iota_s = \iota \qquad \nu_i = \nu$$

Elles sont toutes de la forme $v - P = 0$ où $v \in \mathcal{V}_\eta$, $P \in \mathcal{P}$ et $v \notin \langle P \rangle_\eta$[6]. Pour de telles contraintes, la **stratégie *d'élimination de variables isolées*** substitue v par P dans le système ce qui n'en modifie pas les solutions. Elle permet ici d'inférer le type :

$$\texttt{val pack} : \forall \iota, \kappa, \delta.\ \forall \alpha.\ [\iota * \delta - \delta + \kappa]\,\alpha \to [\iota]\,[\kappa]\,\alpha$$

où la contrainte énoncée sur les tailles et le pas apparaît dans le type. S'il est théoriquement possible d'étendre cette stratégie à des contraintes de la forme $(v - P) * Q$ où $Q \in \mathcal{P}$ ne possède pas de racines entières, une telle factorisation n'est pas facilement dérivable algorithmiquement et ne suffirait pas pour certaines contraintes non linéaires.

L'opérateur `split`, restriction de `pack` où les fenêtres forment une partition du tableau d'entrée (transformation d'un vecteur en matrice) est défini par une annotation de type :

$$\texttt{let split} : [\iota * \kappa]\,\alpha \to [\iota]\,[\kappa]\,\alpha = \texttt{pack}$$

Après instanciation de `pack` par des variables ι_p, κ_p et δ_p, et substitution des variables isolées, l'unique contrainte $(\iota - 1) * \delta_p + \kappa = \iota * \kappa$ est obtenue. Celle-ci (équivalente à $(\iota - 1) * (\delta_p - \kappa) = 0$) est satisfaite si $\iota = 1$ ou $\delta_p = \kappa$. Choisir l'une ou l'autre des égalités pour en dériver une substitution réduit l'ensemble de solutions du système original.

La **stratégie *d'unification de polynômes semblables*** choisit, pour des contraintes c

[6] Ce type de contraintes provient en particulier des variables libres introduites dans les *trous*.

vérifiant : $\exists! \iota_1, \iota_2 \in \mathcal{V}_\eta,\ c\{\iota_1/\iota_2\} = 0$, de substituer ι_2 par ι_1 (δ_p par κ dans l'exemple). Cela permet de simplifier des contraintes de la forme $\eta_1 = \eta_2$, où les deux tailles ne diffèrent structurellement que d'une seule variable. Ces contraintes résultent notamment d'expressions partielles des tailles. Lorsque ces substitutions ne sont pas pertinentes, l'ajout d'annotations pourra orienter l'inférence. Algorithmiquement, l'exploration exhaustive des paires de variables est envisageable car chaque contrainte ne possède que peu de variables libres. Cette stratégie permet à la fonction `split` d'obtenir le type :

$$\texttt{val split} : \forall \iota, \kappa.\ \forall \alpha.\ [\iota * \kappa]\alpha \to [\iota]\,[\kappa]\,\alpha$$

Les contraintes issues de coercitions sont considérées en dernier car ces vérifications post-typage n'ont pas d'obligation à être résolues. Plutôt que de rassembler ces primitives de manipulations de tableaux dans une bibliothèque, elles sont en pratique prédéfinies et traitées spécialement, ce qui réduit le nombre de coercitions à vérifier et permet des optimisations.

5.3 Polymorphisme

Le langage propose un polymorphisme unifié de taille et de type. À la manière des langages de la famille ML, les expressions sont généralisées aux déclarations et les variables sont instanciées à leurs utilisations. De plus, comme le fait le langage OCaml [19, Chapter 5], ces mécanismes implicites sont complétés par plusieurs constructions. (i) Des *annotations de schéma de types* dans les déclarations locales et points-fixe. Elles sont nécessaires pour les récursions polymorphes. (ii) Des *abstractions explicites* de type et de taille introduisant des variables localement abstraites dans un terme. Ces variables seront donc généralisées à la déclaration englobante. (iii) Des *variables anonymes* de taille et de type (préfixées par `'` dans la syntaxe concrète). Elles sont généralisées aux déclarations globales uniquement.

Contrairement au cas des types[7], les variables de taille généralisées qui n'apparaissent pas dans le type conduisent à des expressions non définies. Ainsi la déclaration suivante est rejetée :

$$\texttt{let even} : \_ = \texttt{fold}\ (+)\ \texttt{<\_>}\ 0\ (\lambda i{:}\texttt{[\_]}.\ 2) \qquad \text{(Erreur)}$$

car la taille du tableau (et de l'itération), dont dépend la valeur de l'expression, n'est pas contrainte. L'ajout d'un paramètre de taille produit le type $\forall \iota.\ \texttt{<}\iota\texttt{>} \to \texttt{int}$, ce qui permet de contraindre la variable ι et rend le terme valide :

$$\texttt{let even} : \_ = \lambda n{:}\texttt{<\_>}.\ \texttt{fold}\ (+)\ n\ 0\ (\lambda i{:}\texttt{[\_]}.\ 2)$$

6 Langage de tableaux

Le langage de plus haut niveau $\mathcal{L}_s$ a pour but d'introduire les annotations nécessaires à l'inférence. Il cherche également à se rapprocher des constructions de tableaux de Scade [4]. Son étude est encore incomplète et sa forme sera uniquement esquissée dans cette partie.

Différenciation des arguments. L'ajout des annotations de raffinements est structurée par une séparation syntaxique de trois types d'arguments. Avec les contractions habituelles entre déclarations et abstractions (n-aires), la concaténation de deux tableaux s'exprime par :

```
let concat «n,p» (X:[n]_, Y:[p]_) [i<n+p] =
   case i < n then X[i] else Y[i-n]
```

Suivant les règles de la Figure 7, ce terme est élaboré vers la définition suivante de $\mathcal{L}_i$:

[7] Des variables de type non contraintes proviennent de code mort, ce que l'on peut vouloir interdire également.

Paramètres de taille	*Arguments généraux*	*Indices de tableaux*
$\lambda\text{«}n\text{»}.\ e$	$\lambda(x:\tau).\ e$	$\lambda[i<\eta].\ e$
$\Lambda n.\ \lambda n{:}\text{<}n\text{>}.\ e$	$\lambda x{:}\tau.\ e$	$\lambda i{:}[\eta].\ e$

(*a*) Abstractions

	Paramètres de taille	*Arguments généraux*	*Indices de tableaux*
$\mathcal{L}_s$	$f\ \text{«}\eta\text{»}$	$f\ (e)$	$f\ [e]$
$\mathcal{L}_i$	$f\ \text{<}\eta\text{>}$	$f\ e$	$f\ (e \triangleright [\_])$

(*b*) Applications

Figure 7: Élaboration de $\mathcal{L}_s$ dans $\mathcal{L}_i$

```
let window «k» (X:[k+'n-1]_) [i<'n, j<k] = X [i+j]
let dot_product (u,v) = fold (+) (0, map (*) (u,v))
let convolution (k,i) = map (dot_product (k,_)) (window (i))
```

Figure 8: Convolution dans $\mathcal{L}_s$

$$\texttt{let concat} : \_ = \Lambda n.\ \lambda n{:}\text{<}n\text{>}.\ \Lambda p.\ \lambda p{:}\text{<}p\text{>}.\ \lambda X:[n]\ \_.\ \lambda Y:[p]\ \_.\ \lambda i:[n+p].\ \texttt{case}\ i<n\ \texttt{then}\ X\ (i \triangleright [\_])\ \texttt{else}\ Y\ (i-n \triangleright [\_])$$

L'inférence détermine le type attendu $\forall \iota_1, \iota_2.\ \forall \alpha.\ \text{<}\iota_1\text{>} \to \text{<}\iota_2\text{>} \to [\iota_1]\alpha \to [\iota_2]\alpha \to [\iota_1+\iota_2]\alpha$. L'abstraction de *paramètres de taille* (`«n,p»`) introduit simultanément une variable localement abstraite de taille et une variable d'expression du même nom, permettant une utilisation transparente dans les termes et les tailles. Les accès aux tableaux (applications d'*indices de tableaux*) insèrent une coercition vers le type `[_]`, indice dont la taille sera déterminée par l'inférence.

Stratification des types. $\mathcal{L}_s$ distingue les types de données (dont les tableaux) de ceux des fonctions, de la forme $\text{<}\eta_1,\ldots,\eta_k\text{>} \to \tau_1^i,\ldots,\tau_m^i \to \tau_1^o,\ldots,\tau_n^o$, dont les paramètres de tailles se trouvent en tête et où les types du domaine et du codomaine sont des types de données. Cette restriction empêche l'ordre supérieur et la currification qui, bien qu'explorés dans [2, 3], ne sont pas utilisés dans les langages synchrones pour l'embarqué critique. Cela permet :

(i) La définition d'une notion d'*arité* de fonction, grâce à laquelle la spécification des paramètres de taille est optionnelle. Une passe d'inférence détermine les arités des fonctions et insère le nombre d'arguments nécessaires (sans en spécifier les tailles).

(ii) Des *itérateurs n-aires* traités spécifiquement, qui unifient les définitions précédentes (`map`, `map2`) et possèdent un argument fonctionnel supplémentaire en tête.

Les exemples introduits dans la partie 2 écrits dans $\mathcal{L}_s$ sont présentés dans la Figure 8. La syntaxe `dot_product (k,_)` désigne l'application partielle de `dot_product k`. Dans ces termes, les paramètres de taille laissés à l'inférence sont totalement omis.

7 Travaux connexes, discussion et perspectives

L'étude des types dépendants généraux [31] et leur restriction aux tableaux [30] offre un cadre formel pour l'ajout de prédicats complexes dans les types, au prix d'annotations supplémentaires. La gestion des tailles dans le langage proposé dans cet article permet d'en simplifier l'utilisation. Elle s'articule autour des éléments suivants.

(i) *Un système élémentaire de raffinement des types.* Plusieurs techniques d'inférence de raffinements généraux ont été proposées [8, 17, 26] à l'aide d'outils de résolution de contraintes (solveurs SMT). Ici, les seuls raffinements singletons ($\text{<}\eta\text{>}$) et intervalles ($[\eta]$) suffisent à exprimer des utilisations intentionnelles et extensionnelles des tableaux. En l'absence de sous-

typage entre raffinements qui nécessiterait des preuves d'implications, cette restriction permet de vérifier formellement les raffinements et de les propager simplement pendant l'inférence.

(ii) *Un langage des tailles séparé de celui des termes.* Il permet de spécifier des tailles par du polymorphisme proche de celui des types plutôt que par des types dépendants. Ce langage, constitué de polynômes multivariés à coefficients entiers, est adapté aux manipulations formelles grâce à la forme canonique des tailles, en particulier pour l'inférence. Il étend l'expressivité d'autres systèmes [30, 29] et permet d'exprimer des opérations non-linéaires nécessaires aux algorithmes de traitement du signal ou pour écrire un réseau de neurones artificiels. Plutôt que de considérer les tailles comme des constantes (explicites ou abstraites) ou des variables, comme c'est le cas dans [12], les tailles sont ici spécifiées par des contraintes entre méta-variables.

(iii) *Des obligations de preuve explicites (coercitions).* Ces vérifications pallient aux limites du système de types dans la manipulation formelle des tailles. Elles permettent de dépasser les restrictions liées au langage des tailles. Leur résolution est indépendante de la première phase d'analyse statique (inférence, vérification). Comme dans [15], plusieurs implémentations sont envisageables : code défensif, méthodes formelles plus avancées, stratification de l'exécution, etc. Ces coercitions proviennent en particulier des utilisations extensionnelles (par indices) des tableaux, auxquelles il est préférable de substituer des opérations intentionnelles, qui n'en nécessitent pas et apportent de meilleures perspectives de compilation et d'optimisation. Les travaux sur λ^H [7] et SaC [28] proposent des coercitions similaires, introduites systématiquement aux applications par la compilation. Comme pour Futhark [12], nous avons choisi de les rendre explicites afin d'augmenter les garanties apportées par le typage : les erreurs de coercitions ne peuvent provenir que de celles qui sont écrites explicitement dans le programme.

La solution proposée ressemble par plusieurs aspects à l'ajout de dimensions dans les types présenté dans [16]. Un polymorphisme sur les variables de dimensions y est considéré. La structure du langage des dimensions est cependant plus simple que dans notre cas. Il ne contient qu'une seule opération interne associative et commutative, la multiplication, ce qui permet de définir des types principaux et préserve les propriétés de l'inférence de HM. Les difficultés posées par ce système de type enrichi sont similaires aux nôtres : pas d'unicité des types principaux et le besoin de récursion polymorphe, voire de types dépendants.

Plusieurs travaux autour de SaC proposent des idées qu'il serait certainement possible d'adapter. (i) Dans [29], le polymorphisme de forme (nombre de dimensions) est exploré. Construit sur des types dépendants, il permet de définir des algorithmes génériques sur le nombre de dimensions, grâce un double niveau de description des tableaux par leur rang (nombre de dimensions) et leur forme (taille de chaque dimension). (ii) La définition de tableaux par compréhension est étudiée dans [27]. L'inférence des tailles y est guidée par les accès aux tableaux et évite la spécification des tailles lors des manipulations extensionnelles.

Les perspectives de ce travail sont multiples. Il faudra s'assurer de la compatibilité des constructions de tableaux avec les constructions temporelles propres aux langages synchrones. La mise en place d'un système de types permettant de stratifier l'évaluation, à la manière de [23] est également envisagée. Elle permettrait de restreindre les coercitions impossibles à vérifier à des expressions évaluables à la compilation. Enfin, l'étude de la compilation efficace vers des cibles parallèles utilisées pour le calcul intensif embarqué est prévue.

Remerciements

Nous remercions chaleureusement les relecteurs dont les remarques ont permis d'améliorer la version finale de cet article.

Bibliographie

[1] John Gilbert Presslie Barnes. *High integrity software: the spark approach to safety and security: sample chapters*. Pearson Education, 2003.

[2] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Philadelphia, Pensylvania, May 1996.

[3] Jean-Louis Colaço, Alain Girault, Grégoire Hamon, and Marc Pouzet. Towards a higher-order synchronous data-flow language. In *Proceedings of the 4th ACM international Conference on Embedded Software*, pages 230–239, 2004.

[4] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. Scade 6: A Formal Language for Embedded Critical Software Development. In *Eleventh International Symposium on Theoretical Aspect of Software Engineering (TASE)*, Sophia Antipolis, France, September 13-15 2017.

[5] Francesco Dagnino, Viviana Bono, Elena Zucca, and Mariangiola Dezani-Ciancaglini. Soundness conditions for big-step semantics. In *ESOP*, pages 169–196, 2020.

[6] John T Feo, David C Cann, and Rodney R Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, 1990.

[7] Cormac Flanagan. Hybrid type checking. In *Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, 2006.

[8] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, 1991.

[9] Léonard Gérard, Adrien Guatto, Cédric Pasteur, and Marc Pouzet. A Modular Memory Optimization for Synchronous Data-Flow Languages. Application to Arrays in a Lustre Compiler. In *Languages, Compilers and Tools for Embedded Systems (LCTES'12)*, Beijing, June 12-13 2012. ACM. Best paper award.

[10] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[11] Fritz Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(2):253–289, 1993.

[12] Troels Henriksen and Martin Elsman. Towards size-dependent types for array programming. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 1–14, 2021.

[13] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. Futhark: Purely functional GPU-programming with nested parallelism and in-place array updates. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 556–571, New York, NY, USA, 2017. ACM.

[14] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[15] C Barry Jay and Milan Sekanina. Shape checking of array programs. Technical report, Citeseer, 1996.

[16] Andrew Kennedy. Dimension types. In *European Symposium on Programming*, pages 348–362. Springer, 1994.

[17] Kenneth Knowles and Cormac Flanagan. Type reconstruction for general refinement types. In *European Symposium on Programming*, pages 505–519. Springer, 2007.

[18] 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, 8th European Congress*, 2016.

[19] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.11. *Available electronically at https://coq. inria. fr/refman*, 2020.

[20] Yitzhak Mandelbaum, David Walker, and Robert Harper. An effective theory of type refinements.

ACM SIGPLAN Notices, 38(9):213–225, 2003.

[21] Lambert Meertens. Incremental polymorphic type checking in B. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 265–275, 1983.

[22] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, pages 217–228. Springer, 1984.

[23] Hanne R Nielson and Flemming Nielson. Automatic binding time analysis for a typed λ-calculus. *Science of computer programming*, 10(2):139–176, 1988.

[24] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and practice of object systems*, 5(1):35–55, 1999.

[25] François Pottier and Didier Rémy. The essence of ML type inference. 2005.

[26] Patrick M Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 159–169, 2008.

[27] Artjoms Sinkarovs, Sven-Bodo Scholz, Robert Stewart, and Hans-Nikolai Vießmann. Recursive array comprehensions in a call-by-value language. In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages, IFL 2017, Bristol, UK, August 30 - September 01, 2017*, pages 5:1–5:12, 2017.

[28] F. Tang and C. Grelck. User-defined shape constraints in Sac. In R. Hinze, editor, *24th International Symposium on Implementation and Application of Functional Languages (IFL'12), Oxford, UK*. University of Oxford, 2012.

[29] Kai Trojahner and Clemens Grelck. Dependently typed array programs don't go wrong. *The Journal of Logic and Algebraic Programming*, 78(7):643–664, 2009.

[30] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 249–257, 1998.

[31] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, 1999.

Annexes

A Propriétés de la sémantique

Propagation d'erreurs. Nous détaillons ici la construction des règles de propagation d'erreurs pour les règles *étoilées* de la sémantique (Figure 4).

La règle $\dfrac{e_1 \rightsquigarrow v_1 \qquad e_2 \rightsquigarrow v_2 \qquad \dots \qquad e_n \rightsquigarrow v_n}{e \rightsquigarrow v}\star$ introduit les règles de propagation suivantes :

$$\frac{e_1 \rightsquigarrow \star}{e \rightsquigarrow \star} \qquad \frac{e_1 \rightsquigarrow v_1 \qquad e_2 \rightsquigarrow \star}{e \rightsquigarrow \star} \qquad \dots \qquad \frac{e_1 \rightsquigarrow v_1 \qquad e_2 \rightsquigarrow v_2 \qquad \dots \qquad e_n \rightsquigarrow \star}{e \rightsquigarrow \star}$$

Dans le cas des règles TCase et FCase, les deux premières règles de propagation sont identiques :

$$\frac{e_1 \rightsquigarrow \star}{\texttt{case } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \rightsquigarrow \star}$$

Elles sont considérées indiscernables et n'introduisent pas de non déterminisme dans la dérivation de la sémantique.

Déterminisme affaibli. À cause des règles liées au point-fixe, la sémantique définie dans la sous-partie 3.2 n'est pas déterministe. Cependant, elle ne peut associer à une expression deux valeurs distinctes différentes de $\star$:

$$\forall e,\ \forall v_1, v_2 \in \mathcal{V} \setminus \{\star\},\ e \rightsquigarrow v_1 \wedge e \rightsquigarrow v_2 \implies v_1 = v_2$$

Preuve. Supposons $v_1, v_2 \neq \star$ vérifiant $e \rightsquigarrow v_1 \wedge e \rightsquigarrow v_2$. Par induction sur les arbres de déductions (finis) des sémantiques, et en raisonnant par cas sur la forme des expressions.

- Point fixe. `fix` $x{:}\sigma = e'$
  Deux règles peuvent s'appliquer, Fix et EFix. Par hypothèses, $v_1 \neq \star$, $v_2 \neq \star$ donc la déduction utilise des instances de la règle Fix (identiques) pour lesquelles l'hypothèse d'induction s'applique.
- Définition par cas. `case` e_1 `then` e_2 `else` e_3
  Si $e_1 \rightsquigarrow \star$ alors $e \rightsquigarrow \star$ donc, par hypothèse, $e_1 \rightsquigarrow v$, $v \neq \star$. Par induction, dans les deux cas, la condition se réduit vers des valeurs identiques donc une seule règle s'applique.
- Cas restants : La sémantique est dirigée par la syntaxe. En effet, une seule règle est applicable pour chaque construction, et celles-ci déterminent entièrement les prémisses à partir de la conséquence.

B Propriétés du typage

La preuve de la correction du typage par rapport à une sémantique à grands pas ne peut pas être dérivée des propriétés habituelles de préservation du type et de progrès de la réduction, utiles pour les sémantiques à petits pas, car il n'y a pas de différence entre termes bloqués et divergents. Cependant, une analyse générale des conditions de correction pour des sémantiques à grands pas [5] permet de déduire ces résultats de propriétés similaires. Dans le cas des sémantiques non déterministes, il est utile de distinguer deux sortes de corrections, nommées selon l'article cité précédemment :

- *Soundness-must* : aucune réduction possible n'est bloquée
- *Soundness-may* : une réduction possible au moins n'est pas bloquée

Pour la sémantique de $\mathcal{L}_p$, nous prouverons la correction forte (*Soundness-must*). Cette preuve nécessite les lemmes usuels suivants, et les propriétés (S1), (S2) et (S3), détaillées ci-dessous.

B.1 Lemmes initiaux des preuves sur la sémantique

Normalisation des arbres de typage. La règle de sous-typage (SubType) peut être utilisée à tout moment dans la construction du type. Afin de simplifier les raisonnements par cas suivants, les arbres de déduction seront normalisés afin de restreindre les occurrences de SubType aux seuls expressions x ; <η> ; $p\ v_1\ \ldots\ v_k$ avec $k < \nu(p)$ et `fix` $x : \sigma = e$, typées avec les règles Var, Size, Op et Fix. Pour cela, la relation de sous typage est étendue aux schémas de type à l'aide de la règles ci-contre.

Sous-typage étendu $\sigma_1 <: \sigma_2$

$$\text{STGen}\ \frac{\sigma_1 <: \sigma_2\{\iota_1|\alpha_1/\iota_2|\alpha_2\}}{\Lambda\iota_1|\alpha_1.\ \sigma_1 <: \Lambda\iota_2|\alpha_2.\ \sigma_2}$$

Par la transitivité de la relation de sous-typage, il est possible de normaliser les arbres de déduction afin qu'ils ne contiennent pas deux occurrences consécutives de la règle SubType.

Ainsi, la règle de réécriture suivante permet d'éliminer les occurrences successives. La notation $S_1 \ltimes_\tau S_2$ désigne la preuve de sous-typage construite (par transitivité) à partir des preuves de sous-typage vers le type intermédiaire commun τ.

$$\textsc{SubType}\ \dfrac{\textsc{SubType}\ \dfrac{P \quad \overline{\Gamma \vdash e : \tau_2} \qquad S_2 \quad \overline{\tau_2 <: \tau_1}}{\Gamma \vdash e : \tau_1} \qquad S_1 \quad \overline{\tau_1 <: \tau}}{\Gamma \vdash e : \tau} \longrightarrow \textsc{SubType}\ \dfrac{P \quad \overline{\Gamma \vdash e : \tau_2} \qquad S_1 \ltimes_{\tau_1} S_2 \quad \overline{\tau_2 <: \tau}}{\Gamma \vdash e : \tau}$$

Il reste à montrer comment les occurrences de SubType se distribuent sur les prémisses de la règle utilisée au dessus de celle de sous-typage. Nous ne détaillerons ici que le cas de la règle App et le traitement des coercitions.

$$\textsc{SubType}\ \dfrac{\textsc{App}\ \dfrac{P_1 \quad \overline{\Gamma \vdash e_1 : \tau'' \to \tau'} \qquad P_2 \quad \overline{\Gamma \vdash e_2 : \tau''}}{\Gamma \vdash e_1\ e_2 : \tau'} \qquad S \quad \overline{\tau' <: \tau}}{\Gamma \vdash e_1\ e_2 : \tau}$$

$$\downarrow$$

$$\textsc{App}\ \dfrac{\textsc{SubType}\ \dfrac{P_1 \quad \overline{\Gamma \vdash e_1 : \tau'' \to \tau'} \qquad \textsc{SArrow}\ \dfrac{\textsc{SRefl}\ \overline{\tau'' <: \tau''} \qquad S \quad \overline{\tau' <: \tau}}{\tau'' \to \tau <: \tau'' \to \tau'}}{\Gamma \vdash e_1 : \tau'' \to \tau} \qquad P_2 \quad \overline{\Gamma \vdash e_2 : \tau''}}{\Gamma \vdash e_1\ e_2 : \tau}$$

Le cas des coercitions entières est simplifié par le fait que, pour $\overline{\eta}$ dans $\{\texttt{<}\eta\texttt{>}, \texttt{[}\eta\texttt{]}\}$, les seuls types τ vérifiants $\overline{\eta} <: \tau$ sont $\overline{\eta}$ et `int`. Les coercitions suivies de sous-typage peuvent donc se réécrire de la manière suivante :

$$\textsc{SubType}\ \dfrac{\textsc{CSize}\ \dfrac{P \quad \overline{\Gamma \vdash e : \texttt{int}}}{\Gamma \vdash e : \overline{\eta}} \qquad S \quad \overline{\overline{\eta} <: \tau}}{\Gamma \vdash e : \tau} \longrightarrow \begin{cases} \dfrac{P \quad \overline{\Gamma \vdash e : \texttt{int}}}{\Gamma \vdash e : \texttt{int}} & \text{if } \tau = \texttt{int} \\ \textsc{CSize}\ \dfrac{P \quad \overline{\Gamma \vdash e : \texttt{int}}}{\Gamma \vdash e : \overline{\eta}} & \text{if } \tau = \overline{\eta} \end{cases}$$

Pour les coercitions générales, il est nécessaire de modifier le type impliqué (τ') dans la coercition : les raffinements qui sont effacés du type par le sous-typage sont supprimés dans la nouvelle version ($\widetilde{\tau}'$) tandis que les raffinements introduits sont ajoutés et laissés intacts par la coercition. Les dérivations de sous-typage et d'équivalence de types peuvent être construites à partir des dérivations originales. Par exemple, si $\tau = \texttt{[}\eta\texttt{]}\alpha \to \texttt{int}$; $\tau' = \texttt{[}\eta\texttt{]}\alpha \to \texttt{<}\eta\texttt{>}$ et $\tau'' = \texttt{[}\eta'\texttt{]}\alpha \to \texttt{<}\eta'\texttt{>}$ alors $\widetilde{\tau}' = \texttt{[}\eta'\texttt{]}\alpha \to \texttt{int}$

$$\textsc{SbT}\ \dfrac{\textsc{Coe}\ \dfrac{P \quad \overline{\Gamma \vdash e : \tau''} \qquad E \quad \overline{\tau'' \approx \tau'}}{\Gamma \vdash e \triangleright \tau' : \tau'} \qquad S \quad \overline{\tau' <: \tau}}{\Gamma \vdash e \triangleright \tau' : \tau} \longrightarrow \textsc{Coe}\ \dfrac{\textsc{SbT}\ \dfrac{P \quad \overline{\Gamma \vdash e : \tau''} \qquad \widetilde{S} \quad \overline{\widetilde{\tau}' <: \tau''}}{\Gamma \vdash e : \widetilde{\tau}'} \qquad \widetilde{E} \quad \overline{\widetilde{\tau}' \approx \tau}}{\Gamma \vdash e \triangleright \tau : \tau}$$

Lemme d'inversion. (Dérivation des propriétés des prémisses sous réserve de typage)
Soit Γ, e, σ tel que $\Gamma \vdash e : \sigma$

1. Si $e = x$, alors $\Gamma_e(x) <: \sigma$
2. Si $e = e_1\ e_2$, alors $\sigma = \tau$ il existe τ' tel que $\Gamma, x : \tau \vdash e_2 : \tau' \to \tau$ et $\Gamma, x : \tau \vdash e_1 : \tau'$
3. Si $e = \lambda x{:}\tau.\ e'$, alors $\sigma = \tau_1 \to \tau_2$ et $\Gamma, x : \tau \vdash e' : \sigma$
4. Si $e = \texttt{true}|\texttt{false}$, alors $\sigma = \texttt{bool}$
5. Si $e = n$, alors $\sigma \in \{\texttt{int}, \texttt{<\_>}, \texttt{[\_]}\}$
6. Si $e = e'\ \texttt{[}\eta\texttt{]}$, alors il existe ι, σ' tel que $\Gamma \vdash e' : \Lambda\iota.\ \sigma'$ et $\sigma'\{\eta/\iota\} = \sigma$
7. Si $e = e'\ \texttt{[}\tau\texttt{]}$, alors il existe α, σ' tel que $\Gamma \vdash e' : \Lambda\alpha.\ \sigma'$ et $\sigma'\{\alpha/\tau\} = \sigma$
8. Si $e = \Lambda\iota.\ e'$, alors $\sigma = \Lambda\iota.\ \sigma'$ et $\Gamma, \iota \vdash e' : \sigma'$
9. Si $e = \Lambda\alpha.\ e'$, alors $\sigma = \Lambda\alpha.\ \sigma'$ et $\Gamma, \alpha \vdash e' : \sigma'$
10. Si $e = \texttt{fix}\ \ x{:}\sigma' = e'$, alors $\sigma = \sigma'$ et $\Gamma, x : \sigma' \vdash e' : \sigma$
11. etc

Preuve. Analyse par cas sur la forme des expressions.

Lemme de substitution. (Préservation du type par substitution)

$$\left.\begin{array}{rl} \Gamma, x : \sigma' & \vdash e : \sigma \\ \Gamma & \vdash e' : \sigma' \end{array}\right\} \Longrightarrow \Gamma \vdash e\{e'/x\} : \sigma$$

Preuve. Un arbre de dérivation fini de $\Gamma \vdash e\{e'/x\} : \sigma$ est obtenu en replaçant dans une dérivation finie de $\Gamma, x : \sigma' \vdash e : \sigma$ toutes les occurrences de la règle VAR (en nombre fini) par une dérivation de $\Gamma \vdash e' : \sigma'$ (finie). Il faut cependant modifier les environnements pour chaque occurence car ils peuvent être étendus. Cela est valide grâce au lemme suivant : le typage est préservé par extension de l'environnement. En effet, les variables ajoutées sont fraîches et ne peuvent masquer de variables existantes.

Lemme de forme canonique. (Dériver la former des valeurs de leur type)
Pour tout schéma de type σ, on note $\{\!|\sigma|\!\} = \{v \in \mathcal{V} | \vdash v : \sigma\}$. Alors,

$$\begin{aligned} \{\!|\texttt{int}|\!\} &= \{n \mid n \in \mathbb{Z}\} \\ \{\!|\texttt{bool}|\!\} &= \{\texttt{true}, \texttt{false}\} \\ \{\!|\cdot \to \cdot|\!\} &= \{\lambda\cdot{:}\cdot.\ \cdot\} \cup \{p\ v_1\ \ldots\ v_k \mid k < \nu(p)\} \\ \{\!|\forall\ \cdot\ .\ \cdot|\!\} &\subset \{\Lambda\ \cdot\ .\ \cdot\} \cup \{p\} \end{aligned}$$

Pour les fonctions, il est nécessaire de considérer les opérateurs partiellement appliqués. Pour les types polymorphes, seule l'inclusion est vérifiée si certains opérateurs sont monomorphes.
Preuve. Analyse par cas sur la forme des valeurs.

B.2 Propriétés du système

Les sémantiques construites automatiquement dans [5] permettent de prouver la correction et la préservation du type à partir de trois propriétés locales détaillées ci-dessous. Ces propriétés ne nécessitent pas d'induction (laquelle est construite dans la preuve générique).

Pour les présenter (spécialisées pour notre sémantique et notre système de types), la notation d'instance de règle en ligne suivante est reprise de l'article :

$$(e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n, e_{n+1} \rightsquigarrow v_{n+1}, e) \equiv \frac{e_1 \rightsquigarrow v_1 \quad \ldots \quad e_n \rightsquigarrow v_n \quad e_{n+1} \rightsquigarrow v_{n+1}}{e \rightsquigarrow v_{n+1}}$$

Où $e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n$ sont les *prémisses* et $e_{n+1} \rightsquigarrow v_{n+1}$ est la *continuation*, qui produit la valeur résultant de la réduction. Pour les règles ne possédant pas cette forme, une continuation triviale $v_{n+1} \rightsquigarrow v_{n+1}$ peut être ajoutée.

(S1) Préservation locale. Pour toute instance $(e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n, e_{n+1} \rightsquigarrow v_{n+1}, e)$, telle que $\vdash e : \sigma$, il existe $\sigma_1, \ldots, \sigma_{n+1}$ avec $\sigma_{n+1} = \sigma$ vérifiant :

$$\forall k \in [\![1, n+1]\!],\ (\forall h \in [\![1, k-1]\!],\ \vdash v_h : \sigma_h) \implies \vdash e_k : \sigma_k$$

Preuve. Par cas sur les instances de règles de la sémantique (en utilisant le lemme d'inversion).

- Règle β-RED : $e = e_1\ e_2$. On suppose qu'il existe σ tel que $\vdash e : \sigma$.
  Par le lemme d'inversion (2), $\sigma = \tau'$ et $\vdash e_1 : \tau \to \tau'$ et $\vdash e_2 : \tau$
  - Prémisse $e_1 \rightsquigarrow \lambda x{:}\tau.\ e$. On a bien $\vdash e_1 : \tau \to \tau'$
  - Prémisse $e_2 \triangleright_s \tau \rightsquigarrow v$. On a bien $\vdash e_2 \triangleright_s \tau : \tau$
  - Prémisse $e\{v/x\} \rightsquigarrow v'$. Par hypothèse, $\vdash \lambda x{:}\tau.\ e : \tau -> \tau'$ et $\vdash v : \tau'$. Par le lemme d'inversion (4), $x : \tau \vdash e : \tau'$, donc par le lemme de substitution $\vdash e\{v/x\} : \tau'$
- Règles PARTOP, TOTOP : raisonnement similaire
- Règle INST : $e = e'\ [\eta|\tau]$. On suppose qu'il existe σ tel que $\vdash e : \sigma$.
  Par le lemme d'inversion (6,7), $\vdash e_1 : \Lambda\iota|\alpha.\ \sigma'$ et $\sigma'\{\eta|\tau/\iota|\alpha\} = \sigma$
  - Prémisse $e_1 \rightsquigarrow \Lambda\iota|\alpha.\ e$. On a bien $\vdash e_1 : \Lambda\iota|\alpha.\ \sigma'$
  - Prémisse $e\{\eta|\tau/\iota|\alpha\} \rightsquigarrow v'$. Par préservation du typage par substitution de variables de taille ou de type, on a bien $\vdash e\{\eta|\tau/\iota|\alpha\} : \sigma'\{\eta|\tau/\iota|\alpha\}$
- Règles TCASE, FCASE : application directe du lemme d'inversion
- Règle SIZE : $e = \texttt{<}n\texttt{>}$ Une prémisse (continuation) $n \rightsquigarrow n$ et $\vdash n : \texttt{int}$
- Règles ERR, EFIX : trivial
- Règle LET : $e = \texttt{let}\ x{:}\sigma' = e_1\ \texttt{in}\ e_2$. Par le lemme d'inversion :
  - Prémisse $e_2 \triangleright_s \sigma' \rightsquigarrow v_1$: On a bien $\vdash e : \sigma'$
  - Prémisse $e\{v_1/x\} \rightsquigarrow v$. Par hypothèse, $\vdash v : \sigma'$. Par le lemme d'inversion (4), $x : \sigma \vdash e : \sigma'$, donc par le lemme de substitution $\vdash e_2\{v_1/x\} : \tau'$
- Règle LETS, FIX : similaire
- Règle COERCE : $e = e' \triangleright \tau$ On suppose qu'il existe σ tel que $\vdash e : \sigma$.
  Par le lemme d'inversion, il existe σ' tel que $\vdash e : \sigma'$ et $\sigma \approx \sigma'$.

(S2) Progrès existentiel. Pour tout $e \notin \mathcal{V}$, si il existe σ tel que $\vdash e : \sigma$, alors il existe une instance de règle de la forme $(j_1, \ldots, j_n, j_{n+1}, e)$
Preuve. Traitement par cas trivial sur les expressions

(S3) Progrès universel. Pour toute règle $(e_1 \rightsquigarrow v_1, \ldots, e_n \rightsquigarrow v_n, e_{n+1} \rightsquigarrow v_{n+1}, e)$, si il existe σ tel que $\vdash e : \sigma$, alors pour tout $k \in [\![1, n+1]\!]$,

si pour tout $h < k$, $e_h \rightsquigarrow v_h$ et $e_k \rightsquigarrow v$, alors il existe une règle $(j'_1, \ldots, j'_{n'}, j_{n'+1}, e')$ telle que
$\forall h < k,\ j'_h = j_h$, $e' = e$, et $j_k = e'' \rightsquigarrow v$

Intuitivement, il faut montrer que les sous-expressions s'évaluent vers des résultats qui satisfont leur utilisation (forme d'expression attendue).
Preuve. Analyse par cas sur les instances des règles de la sémantique.

- TCase, FCase. Par le typage (règle Case), $\vdash e_1 : \texttt{bool}$. Donc par le lemme de forme canonique $e_1 \rightsquigarrow \texttt{true}|\texttt{false}|\star$. Donc l'une des deux règles s'applique. Pour la continuation, v peut être instancié librement.
- Règle β-red : Par le typage, (règle App), $\vdash e_1 : \tau_1 \to \tau_2$. Donc par le lemme de forme canonique $e_1 \rightsquigarrow v^\star$ où $v^\star$ a la forme:
  - $\star$: une règle de la propagation d'erreur de β-red s'applique alors
  - $\lambda x{:}\tau.\ e$: la règle β-red s'applique alors
  - $p\ v_1\ \ldots\ v_k$, où $k < \nu(p)$: Alors l'une des règles PartOp ou TotOp s'applique.

  La deuxième valeur peut être instanciée librement.
- Règle β-red : similaire
- Règle LetS : Obtenir un entier est garanti par le lemme de forme canonique et le fait que $\vdash e_1 : \texttt{int}$. La construction de la déduction n'est donc pas bloquée.
- Les autres règles n'imposent pas de forme particulière sur les résultats des réductions de leur sous-expressions.

Théorème. L'article [5] montre les implications suivantes, qui vérifie les résultats annoncés dans la sous-partie 4.2 :

$$(S1) \implies \text{(Préservation du type)}$$
$$(S1) + (S2) + (S3) \implies \text{(Correction du typage)}$$

C Calculs sur les indices

Afin de réduire le nombre de coercitions nécessaires, il est possible d'enrichir le système de type de quelques règles spécifiques aux entiers raffinés et aux opérations arithmétiques. Le cas du type singleton a été brièvement présenté (partie 4). Introduire des règles similaires pour le calcul sur les indices ne semble cependant pas pertinent. Bien qu'il soit possible d'ajouter les

règles dérivées des inclusions suivantes (dans lesquelles les types sont vus comme leur ensemble de valeurs) :

$$\texttt{[}\eta_1\texttt{]} + \texttt{[}\eta_2\texttt{]} \subset \texttt{[}\eta_1 + \eta_2 - 1\texttt{]} \qquad \texttt{[}\eta_1\texttt{]} + \texttt{<}\eta_2\texttt{>} \subset \texttt{[}\eta_1 + \eta_2\texttt{]}$$
$$\texttt{[}\eta_1\texttt{]} * \texttt{[}\eta_2\texttt{]} \subset \texttt{[}\eta_1 * \eta_2 - \eta_1 - \eta_2\texttt{]} \qquad \texttt{[}\eta_1\texttt{]} * \texttt{<}\eta_2\texttt{>} \subset \texttt{[}\eta_1 * \eta_2 - \eta_2 - 1\texttt{]}$$

il ne sera pas possible de gérer de soustractions, car le type `[_]` ne permet pas de définir de borne inférieure non nulle. De plus, le cas de la concaténation échappera également au système de types puisqu'il requiert la prise en compte des gardes de la définition par cas.

La vérification de ces propriétés relève d'une interprétation abstraite (à l'aide d'intervalles, par exemple). Par soucis de simplicité du système de types, ces extensions ont été écartées, au profit de l'ajout de primitives considérées comme correctes (`window`, `sample`, ...), qu'il est donc inutile de vérifier et dont l'expressivité est semblable.

Sauf à utiliser directement des indices, il est donc nécessaire d'insérer des coercitions pour accéder aux tableaux. Celles-ci seront plus facilement vérifiables du fait des informations de tailles.

D Transformée de Fourier rapide

Pour un tableau de taille N, sa transformée de Fourier discrète est définie par :

$$P_k = \sum_{n<N} X_n e^{-\frac{2i\pi}{N}nk}$$

Son calcul suivant cette formule requiert $\mathcal{O}(N^2)$ opérations. En effectuant le changement de variable suivant, permettant d'interpréter un vecteur comme une matrice :

$$\left\{ \begin{array}{rcl} N & = & N_1 N_2 \\ n & = & n_2 N_1 + n_1 \\ k & = & k_1 N_2 + k_2 \end{array} \right.$$

cette expression peut être décomposée de la façon suivante :

$$P_{k_1N_2+k_2} = \underbrace{\sum_{n_1<N_1} \underbrace{\left[e^{-\frac{2i\pi}{N}n_1k_2}\right]}_{2} \underbrace{\left(\sum_{n_2<N_2} X_{n_2N_1+n_1} e^{-\frac{2i\pi}{N_2}n_2k_2}\right)}_{1} e^{-\frac{2i\pi}{N_1}n_1k_1}}_{3}$$

Où l'on reconnaît deux transformées de Fourier (1 et 3) appliquées à chaque ligne ou colonne, séparée d'une multiplication points à points (2). De cette formule est dérivé l'algorithme de Fourier-Tukey, dont la complexité pour une entrée dont la taille est une puissance de 2 est $\mathcal{O}(N \log N)$.

Pour l'implémenter, supposons disposer d'un type `complex` et des opérations associées.

```
val exp : complex → complex
val (+) : complex → complex → complex
val (*) : complex → complex → complex
val (/) : complex → complex → complex
```

Rappelons les schémas de type des opérations de tableaux qui seront utilisées :

```
val split : ∀ι, κ. ∀α. [ι ∗ κ]α → [ι][κ]α
val flatten : ∀ι, κ. ∀α. [ι][κ]α → [ι ∗ κ]α
val transpose : ∀ι1, ι2. ∀α. [ι1][ι2]α → [ι2][ι1]α
```

Définissons une recherche de factorisation (naïve), renvoyant 1 si l'argument est premier ou le plus petit diviseur sinon, ainsi qu'une multiplication matrice-vecteur.

```
let div : _ = λn:_.
   (fix f:_ = λi:_.
      case i ∗ i > n then 1 else
      case n%i = 0 then i else f (i + 1)
   ) 2

let dot_product : _ = λu:_. λv:_. fold (+) <_> 0 (map2 (∗) <_> u v)
let mat_vec : _ = λA:_. λu:_. map (λx:_. dot_product x u) A
```

La transformée de Fourier rapide s'écrit alors :

```
let fft : _ =
   fix f:∀ι. <ι> → [ι]_ → [ι]_ = λn:_. λx:_.
      let tw:_ = λk:[_]. λj:[_]. exp (2i ∗ pi ∗ k ∗ j/n) in
      let size κ = div n in
      case <κ> = 1 then mat_vec tw x else
         let size δ = n/<κ> in
         let x:_ = x ▷ [κ ∗ δ]_ in
         let x:_ = split x in
         let x:_ = map (f <_>) <κ> x in
         let x:_ = transpose (map2 (map2 (∗) <_>) <_> tw x) in
         let x:_ = map (f <_>) <δ> x in
         let x:_ = flatten x in
         x ▷ [ι]_
```

Dans le langage de surface (et en syntaxe concrete), on écrit :

```
let vec_vec (u,v) = fold (+) (map (*) (u,v))
let mat_vec (A,u) = map (dot_product (_,u)) (A)
let rec fft «n» (x:[n]_): [n]_ =
   let tw [k,j] = exp (2i*pi*k*j/n) in
   let size k = div (n) in
   case k = 1 then mat_vec (tw,x) else
      let size d = n/k in
      let x = (x :> [k*d]_) in   // Coercition
      let x = split (x) in
      let x = map fft «k» (x) in
      let x = transpose (map (map (*)) (tw,x)) in
      let x = map fft «d» (x) in
      let x = flatten (x) in
      (x :> [n]_)                // Coercition
```

Remarques : (i) Il est nécessaire de préciser les occurrences de n dans le type de la valeur (fonction) récursive définie. (ii) le tableau tw est instancié de deux façons : `[ι][ι]complex` et `[κ][δ]complex`

Formalising Futures and Promises in Viper*

Cinzia Di Giusto[1], Loïc Germerie Guizouarn[1],
Ludovic Henrio[2], and Etienne Lozes[1]

[1] Université Côte d'Azur, Nice, France
{cinzia.di-giusto,loic.germerie-guizouarn,etienne.lozes}@univ-cotedazur.fr
[2] Université de Lyon, EnsL, UCBL, CNRS, Inria, LIP, Lyon, France
ludovic.henrio@cnrs.fr

Abstract

Futures and promises are respectively a read-only and a write-once pointer to a placeholder in memory. They are used to transfer information between threads in the context of asynchronous concurrent programming. Futures and promises simplify the implementation of synchronisation mechanisms between threads. Nonetheless they can be error prone as data races may arise when references are shared and transferred. We aim at providing a formal tool to detect those errors. Hence, in this paper we propose a proof of concept by implementing the future/promise mechanism in Viper: a verification infrastructure, that provides a way to reason about resource ownership in programs.

1 Introduction

Futures and promises are synchronisation primitives that are common in many programming languages. A future is a read-only placeholder for a value to be computed. It is usually filled with the result value of an asynchronous execution. A future can however be explicitly paired with a promise, i.e., a writable single assignment container which is used to set the value of the associated future.

Somehow like a communication channel that may carry only one message, a future/promise pair is used to transfer information between threads in the context of parallel executions. As an example, consider the Viper code of Listing 1. Technical details as well as necessary logical specification, represented in grey, can be left aside for now. Here, the same object (`p`) is used to act both as a future (read pointer) and a promise (write-once pointer). We will explain both the technical details and the way we represent the promise/future pair in the next section. Intuitively, a thread executing the method `m` is spawned and "returns" its result by resolving the promise `p`, while the main thread "awaits" this result with a blocking call to `GET` on future `p`. Later on, a second call to `GET` can be performed, it is non-blocking, and returns the same value as the first get. On the other side, multiple resolves on the same promise should be forbidden as a promise is a single-write entity. This helps to ensure determinacy of programs using futures. Aside from the issue of double resolves, the programmer also has to deal with standard pitfalls of concurrent programming like deadlocks and data races.

In this work we propose a Viper [24] library for future-manipulating programs that ensures standard safety properties, including memory safety, absence of races, and absence of double resolves. We introduce the idea of associating a "resource invariant" with each future/promise pair: the resolve primitive "inhales" the resource invariant, and the get primitive "exhales" it (in the simple case where there is just one get).

*This work has been supported by the French government, through the EUR DS4H Investments in the Future project managed by the National Research Agency (ANR) with the reference number ANR-17-EURE- 0004.

```
1 field value: Int
2
3 method m(p: Ref, x: Ref)
4   requires promise(p)
5   requires unfolding promise(p) in p.inv_id == 0
6   requires acc(x.value)
7 {
8   x.value := x.value * 2
9   RESOLVE(p, x)
10 }
11
12 method main {
13   var p: Ref
14   var x: Ref
15   var a: Ref
16   var b: Ref
17   x := new(value)
18   p := new_promise(0)
19   x.value := 4
20   m(p, x)
21   GET(p, a)
22   GET(p, b)
23 }
```

Listing 1: Simple example with resolve, get, and asynchronous call

The promises we present in this paper are similar to the one implemented by `std::promise` in C++. The programmer can create a new promise, get the future associated to this promise, and set the value of the promise. Setting the value can happen only once, any later attempt will throw an error. In Java, the Future interface defines the consumer end of the concept, while the `CompletableFuture` implementation adds an explicit producer end placeholder. Method `get()` is a blocking read into the shared value, and `complete()` is the method allowing to set this value. An arbitrary number of calls to `complete()` are allowed, the method will return true for the first one and false for all subsequent ones, not modifying the first set value. The same two pointers approach is implemented in Rust: the crate `future` offers the function `promise` which creates a `Promise` and a `Complete`, where the `Promise` is the readonly placeholder and `Complete` is the write only pointer. This terminology differs from ours but the principle is the same: we get dissociated write and read pointer to a placeholder. As a last example of future/promise implementations that is worth mentioning is the JavaScript take. Promises work more as what we described as plain futures earlier, the difference being that instead of a `return` statement they use a `resolve` function which is a parameter of the spawn thread. The need for an explicit call to `resolve` makes our approach more suitable to reason about JavaScript programs than another one where a future would be completed implicitly at the end of a thread.

Our goal is to propose a general approach to promises and future so that all above concepts may be formalised and checked against properties. This paper represents a first step towards this objective. We propose a proof of concept by using the features of the verification framework Viper. The implementation of the library as well as the examples we discuss in this paper are available at [1].

Outline. We begin with a short introduction to Viper in Section 2. In Section 3 we present the promise/future mechanism, the challenges of its verification, and its implementation in Viper. In Section 4 we showcase our implementation on a classical producer-consumer example and a binary tree merging algorithm, both from Blelloch and Reid-Miller [4]. Finally, in Section 5 we discuss some limitations and future works.

Related works. Although not explicitly described in this paper, our final goal is to ensure properties of the promise/future mechanism via a suitable separation logic: as a matter of fact rules of the logic are represented here by proper pre and post conditions in the tool we have used. Hence the closest group of related works concerns the treatment of synchronisation mechanism in separation logic. Indeed, there has been a lot of work about axiomatising various specific synchronisation primitives, including locks [12, 14], channels [32, 21, 22, 15], or barriers [16, 17], to only quote a few. Parkinson pointed out that: "*there is a disturbing trend for each new library or concurrency primitive to require a new separation logic*" [26], and advocated that "*adding new concurrency libraries should simply be a matter of verification, not of new logics or meta-theory*". Research therefore also focused on extending the expressive power of separation logic with features like permissions [5], rely-guarantee reasoning [31], higher-order [25, 27, 30], or views [8], only quoting a few of these extensions. These ideas are now well integrated in separation logic frameworks like Iris [19], Verifast [18] or Gillian [29]. Permission reasoning can even be encoded in tool that does not support this feature natively, like Why3 [10]. An example of such an encoding can be found in [6].

For what concerns, more specifically, futures and promises, a Hoare logic for reasoning about shared futures has been defined by Din and Owe [7]; the logic bases on "communication histories" of the objects but does not deal with resource ownership in a way comparable with separation logic. More recently, Gardner *et al.* addressed the verification of JavaScript programs that manipulate promises [28]; their approach builds on *transpiling* javascript code to the intermediate JSIL language, for which a symbolic execution engine has been developed. It seems rather technically involved for a newcomer to understand this transpiling process and guess how the original program should be specified to be correctly symbolically executed.

Finally, the promises and future mechanisms presents some similarities with prophecy variables [20], we leave for future work to proper study the connections between those two concepts.

2 Viper

Viper [24], standing for Verification Infrastructure for Permission-based Reasoning, is a verification infrastructure. More precisely, it offers an intermediate language and a verification condition generator. Programs written in Viper are checked against logical specifications, allowing to reason about ownership of resources.

A Viper program consists of a sequence of declaration of memory reference fields, predicates, functions and methods. Each memory reference is instantiated with all the declared fields. Permissions specify which fields and predicates may be accessed: `write` denotes full ownership, `none` no ownership and `wildcard` "some amount" of permission. Method declaration typically begins with a set of pre and post-conditions. Pre-conditions give the permissions as well as the properties (logical assertions) required to execute the body of the method, while post-conditions represent what the method call will ensure, in terms of permissions or properties, after its executions. The grey lines of Listing 1 are an illustration of those concepts. The keyword `acc` at line 6 is an assertion specifying total permission on `x.value`. Without this

pre-condition, method `m` could not perform the update at line 8. The pre-condition at line 4 specifies full permission on a predicate: `promise(x)`.

Predicates allow to define composed assertion parametrised by some value (in `promise(x)`, for instance, `x`, the reference that is a promise). They describe some permissions, but in order to use those permissions they must be "unfolded". The `unfold` keyword transfers the ownership of resources from the predicate to the program state. The converse action, `fold`, asserts ownership of a predicate whose body is satisfied and consumes the resources captured by it (hence the program state loses all permissions on those resources). The last pre-condition, at line 5, uses this mechanism: it requires a specific field to be equal to 0, and to gain the right to access this field, a permission on this field contained in `promise(p)` is obtained by temporarily unfolding this predicate. Why this pre-condition is useful will be explained later.

To verify a program, Viper checks all methods independently. Permission on resources accessed in the body of a method must be granted by the pre-conditions of this method, and the program state at the end of the body must satisfy the post-conditions. A method call is only interpreted with respect to its logical specifications: it asserts and consumes the pre-conditions, and assumes the post-conditions. For example, in Listing 1, after the call to `m` in method `main`, because permission on `x.value` is required by `m` and not ensured as post-condition, this field cannot be accessed after line 20.

3 Verifying Programs Using Promises

In this section we will explain how promises and futures work, and how we can verify programs that use them. We will then present our implementation of these synchronisation primitives in Viper.

3.1 Promises and Futures

To encode the use of promise/future pairs, we need four ingredients:

- a way to start an asynchronous computation,
- a way to create the pair of pointers, we call `new_promise` the statement that performs this operation,
- a way to assign a value to the promise, we call this operation `resolve`,
- a way to read the value of a resolved future, we call this operation `get`.

The first two elements are self-explanatory. By definition, only one `resolve` is allowed for each promise. This prevents race-conditions between accesses to the promise. The `get` primitive is blocking until the future is assigned a value through its associated promise, then it returns this value.

A key consideration is that this communication mechanism is used to transfer information between asynchronous threads. This implies that in a heap modifying language, the transferred information could be a reference to memory. To verify programs using this mechanism, we have to be able to reason about how ownership is transferred alongside the information. We will focus on a case where the only way to synchronise and transfer information between asynchronous threads is the resolution of a promise followed by the `get` operation on its associated future. In the example discussed in the Introduction (Listing 1), to prevent data race we must ensure that `m` has exclusive access to `x.value`. But after the line `GET(p, a)`, we need to acknowledge

```
predicate promise(x: Ref){
  acc(x.locker, wildcard) &&
  acc(x.resolved, 1/2) &&
  acc(x.consumed, 1/2) &&
  acc(x.fut_value, 1/2) &&
  x.consumed == none &&
  x.resolved == false &&
  acc (x.inv_id, wildcard)
}

predicate future(x: Ref){
  acc(x.locker, wildcard) &&
  acc(x.inv_id, wildcard)
}

predicate resolved_future(x: Ref) {
  acc(x.locker, wildcard)  &&
  acc(x.resolved, wildcard) &&
  acc(x.fut_value, wildcard) &&
  x.resolved
}
```

Listing 2: Viper promise, future and resolved future predicates

the fact that `m` gave up the lock it had on this field, and allow subsequent instructions to access it.

To tackle this task, we introduce the concept of promise invariant, a logical specification of the information a future can contain. Each promise is associated with such an invariant. Resolving the promise asserts and consumes the invariant, while getting the future assumes it.

There are two caveats to take into consideration for this last operation: a future may be shared between threads, and a future may be read more than once in the same thread. To deal with the first case, we must ensure that the permission obtained on the invariant is equal to the permission owned on the future when calling `get`. As long as we can ensure no more than full ownership on a future is shared between every threads, we know that no more than full ownership on the invariant will be acquired.

If there are multiple calls to `get`, the same permission must not be obtained twice. This can be achieved by logically making the distinction between a future pointer that has not been read yet, and one that has been read at least once. We call a future pointer on which `get` was already called a "resolved future". The `get` operation consumes some permission on a future, and provides a similar amount of permission on a resolved future. We can call `get` on a resolved future as well, this operation does not consume the permission on this object, neither it provides permission on the promise invariant.

3.2 Implementation

The encoding of promises and futures is done via a single memory reference with 5 fields: `fut_value`, `resolved`, `consumed`, `inv_id` and `locker`. Once computed, the result is written in, and read from the field `fut_value`. Field `resolved` is a boolean flag set to true when the

```
1 predicate ip(v: Ref, prom: Ref){
2   acc(prom.inv_id, wildcard) &&
3   (prom.inv_id == 0 ? acc(v.value) :
4    prom.inv_id == 1 ? (v == null ? true :
5                                    acc(v.value) &&
6                                    acc(v.next) &&
7                                    future(v.next) &&
8                                    unfolding future(v.next) in v.next.inv_id == 1) :
9    prom.inv_id == 2 ? tree(v) :
10   true)
11 }
```

Listing 3: Example of promise invariant implementation

promise is resolved. Field `consumed` is a logical variable used to remember how much ownership of the promise invariant has been consumed by the primitive get. The last two fields are used to implement the interplay between futures and promises, allowing the predicates we will define in the following paragraphs to ensure consistency of the reference through resolve and get calls.

More precisely, the role of promises and futures is encoded by three predicates. For a given reference `x`, `promise(x)` gives permission to treat `x` as a promise; `future(x)` gives permission to treat `x` as a future that has not been read yet, that is calling `get` on this future will assume the resource invariant; and `resolved_fut(x)` gives right to "get" a future without gaining permission on the promise invariant. These three predicates are shown in Listing 2.

To ensure consistency of the state of the reference representing the future, the permission to its fields are guarded by predicate `lock_inv`. It is a lock invariant, meaning that only one branch of a parallel composition may use the permission it provides. This predicate is shown in Listing 4, and it ensures that the thread owning it has a partial possession of the reference's fields. This allows a get call to access its fields to read them easily. When combined with the permission contained in the `promise(x)` predicate, the permissions provided by `lock_inv(x)` allow to modify fields `resolved` and `fut_value`. The disjunction at line 7 captures the different permissions that may be owned depending on whether the future is resolved or not. In the first case, this helps keeping track on the amount of permission still available for the promise invariant. The two macros `LOCK(x)` and `UNLOCK(x)` represent the locking (respectively unlocking) mechanism, providing or capturing the resources described by `lock_inv(x)`.

The last predicate we will mention here is `ip(val, fut)` (presented in Listing 3). It is the predicate that represents the permission transferred through a promise resolution. To be able to have a different invariant for different promises, the resources described by this predicate depend on the value of the field `inv_id`. This field is initialised when creating a new promise. The `ip` predicate must be defined for all the different promise invariant needed in a program. The implementation of the predicate in Listing 3 corresponds to what we needed for all the examples of this paper. For instance in Listing 1, the parameter of `new_promise` that was omitted was 0, to specify right to read and write on field `value` is transmitted alongside the reference used to resolve the promise. In the same listing, the pre-condition at line 5 shows the way we can specify what a method ensures about the result it provides through the resolution of a promise.

Next we comment on the encoding of the primitives for manipulating promises and futures: Listing 5 is the implementation of `new_promise`. It allocates the reference, and initialises all

```
predicate lock_inv(x: Ref){
  acc(x.fut_value, 1/2) &&
  acc(x.consumed, 1/2) &&
  x.consumed <= write &&
  x.consumed >= none &&
  acc(x.resolved, 1/2) &&
  (x.resolved ?
    acc(x.consumed, 1/2) &&
    acc(x.fut_value, wildcard) &&
    acc(future(x), x.consumed) &&
    acc(ip(x.fut_value, x), write - x.consumed) &&
    acc(x.resolved, wildcard)
  : true)
}
```

Listing 4: Viper Lock invariant

```
method new_promise(inv_number : Int) returns (res : Ref)
  ensures promise(res) && future(res)
  ensures unfolding future(res) in res.inv_id == inv_number
  ensures unfolding promise(res) in res.inv_id == inv_number
{
  res := new(locker, resolved, consumed, fut_value, inv_id)
  res.resolved := false
  res.consumed := none
  res.inv_id := inv_number
  UNLOCK(res)
  fold promise(res)
  fold future(res)
}
```

Listing 5: Viper promise creation implementation

the fields. It folds the lock invariant in its initial state, and unlocks it. Its last operation is to fold the two predicates ensured by this method: `promise()` and `future()`.

Listing 6 depicts method `resolve(pro, val)`. It requires predicate `promise(pro)` and the promise invariant `ip(val, pro)`. This method has no post-condition, so its calls consume permissions on the promise and the promise invariant. Notice how, thanks to the full ownership on fields `fut_value` and `resolved` provided by the combination of the lock invariant and the promise predicate, the values of those fields are updated, to the computed `v` and `true` respectively. Since the field `resolved` is now true, the ownership of the promise invariant has to be captured when folding the lock invariant. It also means that the fraction of permission on the field `consumed` that was part of the predicate `promise` is now available in the lock invariant, allowing the following get calls to update this field.

Since calls to `resolve` require the promise invariant `ip`, this predicate always has to be folded prior to resolving a promise. Because `ip` depends on `p.inv_id`, folding it requires access to this field. This is granted by unfolding the `promise(x)` predicate. The program state must

```
1 method resolve(x: Ref, v: Ref)
2   requires promise(x)
3   requires ip(v, x)
4 {
5   LOCK(x)
6   unfold lock_inv(x)
7   x.resolved := true
8   x.fut_value := v
9   UNLOCK(x)
10 }
```

Listing 6: Viper resolve implementation

contain this predicate to resolve the promise anyway, but this operation needs to be done explicitly. To simplify writing programs using futures in Viper, we offer a macro doing these operation automatically. We can therefore simply write `RESOLVE(promise, value)`, as it was shown in Listing 1.

The implementation of the `get` primitive is a recursive active wait: if the future is resolved, the value is returned, otherwise get is recursively called with the same arguments. This implementation is shown in Listing 7. The presence of logical annotations renders the code more involved. Indeed, they take into account that the get primitive can be used both when the promise has still to be resolved, or when it is resolved. We therefore need to mimic a disjunction in both the pre and post conditions of this method.

To this aim, we add the parameter `quantity`, which represents the amount of permission on the `future(x)` predicate that will be consumed, and therefore the amount of permission that will be returned on the promise invariant. If this parameter is set to `none`, then we are encoding the case where there are multiple calls to `get` in the same thread. In this case the pre-condition is to have permission on the `resolved_future` predicate, ensuring this is indeed not the first call to `get`. As expected, no permission on the promise invariant is provided in this situation. Moreover, as we have to unfold the predicates manually, this disjunction appears in the code as well, as seen with the branching at line 12. Depending on the quantity parameter we know which permission was required, and therefore which predicate has to be unfolded. Finally, the disjunction also appears when folding the `future` predicate at line 24, but in a different way because here predicate `resolved_future` needs to be folded in both cases. Notice that a partial ownership (a wildcard permission) on `resolved_future` is always enough.

When the parameter `quantity` is different from `none`, some permission on the promise invariant must be released. This permission comes from the predicate `lock_inv` (Listing 4). Observe that when folding this predicate, the amount of permission on the promise invariant (`ip`) that is not captured back into the predicate is equal to the value that was added to the field `x.consumed`. This is also equal to the amount of permission that was captured on `future(x)`, because of line 10. This ensures that the amount of permission released on `ip(x.value, x)` cannot exceed the amount of permission that was held on `future(x)`.

The last post-condition of `get`, at line 9 ensures that the results of two consecutive gets are indeed the same memory reference. Because this is the way `get` will be used most of the time, we provide a `GET(f, res)` macro that corresponds to a `res := get(f, quantity)` statement with `quantity` being set to the amount of permission held on `future(f)` in the current program state. This macro also deals with unfolding the promise invariant if applicable, and as shown

```
method get(x: Ref, quantity: Perm) returns (v: Ref)
  requires quantity >= none && quantity <= write
  requires quantity > none ?
    acc(future(x), quantity)
  : acc(resolved_future(x), wildcard)
  ensures acc(ip(v, x), quantity)
  ensures quantity acc(resolved_future(x), wildcard)
  ensures unfolding acc(resolved_future(x), wildcard) in x.resolved
  ensures unfolding acc(resolved_future(x), wildcard)
    in x.resolved && v == x.fut_value
{
  if (quantity > none) {
    unfold acc(future(x), quantity)
  } else {
    unfold acc(resolved_future(x), wildcard)
  }
  LOCK(x)
  if (!x.resolved) {
    UNLOCK(x)
    fold acc(future(x), quantity)
    v := get(x, quantity)
  } else {
    v := x.fut_value
    if (quantity > none) {
      fold acc(future(x), quantity)
    }
    x.consumed := x.consumed + quantity
    assume x.consumed <= write
    UNLOCK(x)
    fold acc(resolved_future(x), quantity)
  }
}
```

Listing 7: Viper get implementation

in Listing 8, it would be cumbersome to have to write all those lines for each `get` call.

Finally, to implement asynchronous calls to methods, observe that we only need to consume the permission described by the pre-condition of the method we call, and the post-condition is empty. This means that if we can verify that 1) a method can run with its specified pre-condition 2) the asynchronous call can be done in a given environment, and 3) that all subsequent instructions do not need the resources consumed by this call, we know that the called method can run in parallel with the code that called it. The first and second point are checked automatically by the Viper infrastructure. The last point follows from the fact that, in our encoding, asynchronous methods do not have post-conditions. This implies that the resources mentioned in their pre-conditions will be removed from the environment from which they are called, and subsequent instructions will not be able to use permission on those resources.

4 Case Studies

```
define GET(fut, res) {
  var futperm: Perm
  futperm := perm(future(fut))
  unfold acc(future(fut), futperm)
  fold acc(future(fut), futperm)
  res := get(fut, perm(future(fut)))
  unfold acc(ip(res, fut), futperm)
}

define RESOLVE(p, x) {
  unfold promise(p)
  fold ip(x, p)
  fold promise(p)
  resolve(p, x)
}
```

Listing 8: Implementation of the macros simplifying get and resolve

```
method produce(n: Int, p: Ref)
  requires promise(p)
  requires unfolding promise(p) in p.inv_id == 1
{
  if (n == 0) {
    RESOLVE(p, null)
  }
  else {
    var cell: Ref
    cell := new(value, next)
    cell.value := n - 1
    var x: Ref
    x := new_promise(1)
    cell.next := x
    RESOLVE(p, cell)
    produce(n - 1, x)
  }
}
```

Listing 9: Producer/Consumer example: producer

To illustrate how our implementation works, we give two examples inspired by [4]. The first one is a producer consumer scheme, and the second a binary tree merging algorithm.

4.1 Producer Consumer

The producer provides a list, built cell by cell, and the consumer does a computation on the produced list. Here, the produced list is composed by integers from $n-1$ to 0, and the consumer simply sums up the values of all the cells. The cells produced differ from the typical list cells in the fact that the pointer to the next element is a future rather than an actual memory cell.

```
method consume(l: Ref, p: Ref, accu: Int)
  requires future(l)
  requires unfolding future(l) in l.inv_id == 1
  requires promise(p)
  requires unfolding promise(p) in p.inv_id == 0
{
  var cell: Ref
  GET(l, cell)
  if (cell == null) {
    var res: Ref
    res := new(value)
    res.value := accu
    RESOLVE(p, res)
  }
  else {
    consume(cell.next, p, cell.value + accu)
  }
}
```

Listing 10: Producer/Consumer example: consumer

```
method main() {
  var x: Ref
  x := new_promise(1)

  var p: Ref
  p := new_promise(0)

  consume(x, p, 0)
  produce(2, x)

  var res: Ref
  GET(p, res)
}
```

Listing 11: Producer/Consumer example: interaction between the two actors

This allows the consumer to run in parallel with the producer: the producer can return a cell as soon as its value is computed. The consumer can therefore compute a partial result for each cell as soon as it is produced. Viper code for the producer can be seen in Listing 9.

The implementation of the consumer we propose in Listing 10, illustrates one of the perks of having an explicit promise pointer to write the result of a future. Indeed, notice how the promise `p` that is used to collect the result of the computation of the consumer is passed through successive recursive calls and is only resolved by the last call. Using futures implicitly resolved with the termination of the asynchronous thread would lead to a result nested in as many futures as there were recursive calls.

Finally, Listing 11 shows how the producer and the consumer can work together. Notice that as mentioned in the previous section, because both `produce` and `consume` only have pre-

```
split(splitter, tree) :=
  if (tree == empty) then return empty
  else
    if (tree.root > splitter) then
      split_l, split_r := split(splitter, tree.left)
      return (split_l, new tree(tree.root, split_r, tree.right))
    else
      split_l, split_r := split(splitter, tree.right)
      return (new tree(tree.root, tree.left, split_l), split_r)

merge(t1, t2) :=
  if (t1 == empty) return t2
  elif (t2 == empty) return t1
  else
    split_l, split_r := split(t1.root, t2)
    return new tree(t1.root, merge(t1.left, split_l), merge(t1.right, split_r))
```

Listing 12: Tree merging algorithm

conditions, we can consider them as asynchronous methods. The order of the calls to those methods does not make any difference.

4.2 Tree Merging

The next example we implemented in Viper using promises and futures, is a binary search tree merging algorithm. As customary, in a binary search tree with root r, all the values in the left subtree are less than or equal to r, and all the values in the right subtree are greater than r. The algorithm merges two trees $t1$ and $t2$ into a new binary search tree. It is composed by two methods, *split* and *merge*. The method *split* takes a tree t and a value *splitter*, and returns two subtrees of t such that all values of t that were less than or equal to *splitter* are in the first subtree, and all values of t that were greater than *splitter* are in the second subtree. This method proceeds by recursively splitting the left or right subtree of t depending on whether r is greater than *splitter* or not. The method *merge* splits $t2$ using the root of $t1$ as splitter value. It then recursively merges the first half of the split to the left subtree of $t1$, and the second half to the right subtree of $t1$. The algorithm for those methods is shown in Listing 12.

The idea behind the introduction of futures in this algorithm is similar to the one of the previous example. Indeed, notice that the tree structure is recursive in the same way as the the list used in the producer consumer example. Each call of the merge algorithm builds one tree, its root is known from the arguments, and the subtrees are computed by the recursive calls. The intuition is to make the recursive calls asynchronous and to provide the tree computed at each step right away, delaying the subtrees as futures. In a similar way, all the recursive calls to method split can be done asynchronously. As the trees are split from top to bottom, their data will be accessed in the same order as the one of their computation.

To encode this example in our paradigm with explicit promises, we added one promise per return value for the two methods. The method *merge* has to instantiate two new promises before calling *split*, and another one before its recursive call. Notice that the method *split* only has to instantiate one new promise: in each case, it returns one of the results of its recursive call as it is. It means that this recursive call can be completely in charge of fulfilling one of the

```
field value: Ref
field left: Ref
field right: Ref

field type_of_tree: Int
/*  type_of_tree = 0 -> Empty
    type_of_tree = 1 -> Tree
    type_of_tree = 2 -> Fut of tree */

predicate tree(x: Ref){
  acc(x.type_of_tree) &&
  x.type_of_tree < 3 &&
  (x.type_of_tree == 1 ?          // We have an actual tree
      (acc(x.value) && acc(x.left) && acc(x.right) &&
      tree(x.left) && tree(x.right))
    : (x.type_of_tree == 2 ?      // We have a future on a tree
          future(x) &&
          unfolding future(x) in x.inv_id == 2
        : (x.type_of_tree == 0 ?  // We have an empty tree
            true
          : false)))
}
```

Listing 13: Tree predicate for the tree merging example

promises it had in its arguments. The full implementation in Viper of both methods of this example using promises is available at [1].

To represent the trees, we used references, and in addition to the `value` field that we had from the first example, representing here the root of a tree, we needed two fields to represent the left and right subtrees. We chose to represent the trees using an enumerated type, as it would be done in a functional language. A tree may be an empty tree; an actual tree with a value, a left, and right subtrees; or a future of a tree. To encode this type we used another field: `type_of_tree`. Its integer value for a given reference tells us which type of tree this reference is. The last element required to encode the trees is a predicate encapsulating the permissions on the fields of a tree reference. This predicate, alongside with the fields declaration, is shown in Listing 13. It is a parametric predicate depending on the value of the field `type_of_tree`. If the reference is an actual tree (immediately containing data, case where `type_of_tree` = 1), the predicate provides permission on all the fields composing the tree and ensures recursively that the left and right subtrees are trees as well. If the reference is an empty tree (`type_of_tree` = 0), the predicate does not provide any permission other than the one on field `type_of_tree`. If the reference is a future of a tree (`type_of_tree` = 2), the `tree` predicate ensures that the value this future will be resolved with, will be a tree. This is ensured by enforcing the field `inv_id` of the future to be 2, as we defined `ip(x, f)` to be `tree(x)` if `f.inv_id` is 2. Note that in this last case, the reference we deal with is a future on a tree and a tree itself. Not all futures on trees are themselves trees. For a reference to be considered as a tree, permission on its field `type_of_tree` must be available.

One interesting consideration is that, as our algorithm takes such trees as input, we have therefore to take into account that the input might be either an immediate tree or a future on

```
method split(splitter: Int, current_tree: Ref, pl: Ref, pr: Ref)
  requires tree(current_tree) && promise(pl) && promise(pr)
  requires unfolding promise(pl) in pl.inv_id == 2
  requires unfolding promise(pr) in pr.inv_id == 2
{
  unfold tree(current_tree)
  if (current_tree.type_of_tree == 0) {
    ...
  }
  else {
    if (current_tree.type_of_tree == 1) {
      ...
    } else {
      var actual_tree: Ref
      GET(current_tree, actual_tree)
      split(splitter, actual_tree, pl, pr)
    }
  }
}
```

Listing 14: Removing the future layers with recursive calls

a tree. Recursively, because a tree can be a future on a tree, the actual data of the tree (that is an empty tree or a root and a pair of pointers) may be nested in an arbitrary large number of future layers. We will show two ways we used to deal with this situation in the implementation of the two methods of the tree merging algorithm.

The first one is used in the implementation of the method *split*. We built this method around a disjunction on the value of the field `type_of_tree` of the input tree. The last case of this disjunction is illustrated in Listing 14, and is the case where the input tree is a future. The idea is to use a `get` call to wait for the future to be resolved and retrieve its value, and then to recursively call `split` again with this value. Because of the promise invariant we know that this value is a tree, and if it was again a future on a tree, the recursive call would again fall in the same case of the disjunction, triggering a new recursive call. The same idea would apply to this new recursive call, and so on until the value gotten from the future is not itself a future.

The approach we used in method merge differs because here there are two trees as input, and using the same disjunction is not ideal. Instead, we used a while loop to call `get` as many times as required on the trees given as argument. The implementation of this technique is shown in Listing 15. After the while loops, we know that the references `actual_t1` and `actual_t2` contain actual data, and are not futures.

5 Concluding Remarks

In this paper we have introduced the concept of promise invariant, allowing to reason about the resource transferred between asynchronous threads through promise resolution. We have proposed an implementation of the promise/future mechanism in Viper, based on this promise invariant. Two case studies showcasing how our Viper library can be used to verify actual programs conclude our presentation.

```
1 method merge(t1: Ref, t2: Ref, p: Ref)
2   requires tree(t1) && tree(t2)
3   requires promise(p)
4   requires unfolding promise(p) in p.inv_id == 2
5 {
6   var actual_t1: Ref
7   var actual_t2: Ref
8
9   actual_t1 := t1
10  actual_t2 := t2
11
12  while (unfolding tree(actual_t1) in actual_t1.type_of_tree == 2)
13    invariant tree(actual_t1)
14  {
15    unfold tree(actual_t1)
16    var temp: Ref
17    GET(actual_t1, temp)
18    actual_t1 := temp
19  }
20
21  while (unfolding tree(actual_t2) in actual_t2.type_of_tree == 2)
22    invariant tree(actual_t2)
23  {
24    unfold tree(actual_t2)
25    var temp: Ref
26    GET(actual_t2, temp)
27    actual_t2 := temp
28  }
29  ...
30 }
```

Listing 15: Removing the future layers with while loops

We conclude with some considerations about limitations of our current proposition and future works.

Implementation. One limitation of our implementation is that it does not rely on Viper verification to ensure that the quantity of permission on the promise invariant released by successive calls to `get` is not more than `write`. Instead, we need to use an `assume` primitive, as it is shown at line 28 in Listing 7. This primitive instructs the verification tool to "assume" the assertion is true, even if it cannot verify it.

Viper. Since Viper does not implement higher order predicates, the implementation of the promise invariant is not ideal. In fact we have to define the predicate `ip` in the same file as the promise/futures handling primitives. Because the definition of this predicate has to take into account all the different invariants needed to prove a program, the proof of the program has to be done in the same file as the implementation of the primitives, and we cannot provide a library working as a black box. This limitation also prevents us from implementing a logical connection between the arguments of a method and the resource invariant of a promise it resolves. For

instance, in a simple program as the one in Listing 1, we cannot design a promise invariant allowing us to assert that `a.value` is equal to 2 times what `x.value` was before calling `m`.

Explicit and implicit futures. Terminology and implementation highly vary depending on the chosen programming language. In some languages, a future is created for each spawned thread, whereas the promise is implicitly filled by the return statement of the thread. In some other languages, the usage of futures is even more implicit: no explicit get instruction is needed to access the value stored in the future, and the compiler inserts the missing get when it guesses that it is needed. The respective advantages of explicit versus implicit futures is discussed in [9]. It would be interesting to explore how our logic could be adapted to deal with such implicit futures.

Data race freedom and absence of race condition. Separation logic is known to ensure data race freedom: in a proved program, all memory accesses are performed on owned locations. But still the program can have "race conditions" in the sense that its outcome can be non-deterministically depending on the scheduler. For some synchronisation primitives, like locks, data races and race conditions are two different things. But for some other synchronisation primitives, it can be expected that the ownership discipline imposed by separation logic enforces a form of determinism. This has been formalised for instance for channel communications in [11]. It would be interesting to see if a form of determinism can be enforced for a toy imperative, parallel programming language with pointers and futures.

Cost models and parallel time complexity. Blelloch and Reid-Miller argue in [4] that futures allow to give efficient parallel implementations of sequential algorithms with very light modifications; in their paper, they study the parallel time complexity of some algorithms based on future. This is one of these tree manipulating algorithms that we implemented and proved correct in Viper. It would be interesting to go further and also establish the parallel complexity of this algorithm. Two recent line of research are particularly attractive: separation-logic based proofs of (sequential) time complexity [13, 23], and type-based characterisations of parallel complexity [2, 3]. It would be very interesting to combine the ideas of these two lines of work and develop a proof system for parallel time complexity based on separation logic.

Formal proofs. This work is preliminary to a more foundational approach. We aim at proving formally the soundness of the axioms that are here encoded in Viper logic.

Acknowledgements. We would like to thank all the JFLA anonymous reviewers for their comments that greatly improved the present paper.

References

[1] Git repository of viper implementation. https://gitlab.com/lgermerie/viperfutures.

[2] Baillot, P., and Ghyselen, A. Types for complexity of parallel computation in pi-calculus. In *Programming Languages and Systems - 30th European Symposium on Programming, ESOP 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings* (2021), pp. 59–86.

[3] Baillot, P., Ghyselen, A., and Kobayashi, N. Sized types with usages for parallel complexity of pi-calculus processes. In *32nd International Conference on Concurrency Theory, CONCUR 2021, August 24-27, 2021, Virtual Conference* (2021), pp. 34:1–34:22.

[4] Blelloch, G. E., and Reid-Miller, M. Pipelining with futures. *Theory Comput. Syst. 32*, 3 (1999), 213–239.

[5] Bornat, R., Calcagno, C., O'Hearn, P. W., and Parkinson, M. J. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005* (2005), pp. 259–270.

[6] Denis, X. Mastering program verification using possession and prophecies. In *JFLA* (2021).

[7] Din, C. C., and Owe, O. Compositional reasoning about active objects with shared futures. *Formal Aspects Comput. 27*, 3 (2015), 551–572.

[8] Dinsdale-Young, T., Birkedal, L., Gardner, P., Parkinson, M. J., and Yang, H. Views: compositional reasoning for concurrent programs. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013* (2013), pp. 287–300.

[9] Fernandez-Reyes, K., Clarke, D., Henrio, L., Johnsen, E. B., and Wrigstad, T. Godot: All the benefits of implicit and explicit futures. In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom* (2019), pp. 2:1–2:28.

[10] Filliâtre, J., and Paskevich, A. Why3 - where programs meet provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings* (2013), M. Felleisen and P. Gardner, Eds., vol. 7792 of *Lecture Notes in Computer Science*, Springer, pp. 125–128.

[11] Francalanza, A., Rathke, J., and Sassone, V. Permission-based separation logic for message-passing concurrency. *Log. Methods Comput. Sci. 7*, 3 (2011).

[12] Gotsman, A., Berdine, J., Cook, B., Rinetzky, N., and Sagiv, M. Local reasoning for storable locks and threads. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings* (2007), pp. 19–37.

[13] Guéneau, A., Charguéraud, A., and Pottier, F. A fistful of dollars: Formalizing asymptotic complexity claims via deductive program verification. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings* (2018), pp. 533–560.

[14] Haack, C., Huisman, M., and Hurlin, C. Reasoning about java's reentrant locks. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings* (2008), pp. 171–187.

[15] Hinrichsen, J. K., Bengtson, J., and Krebbers, R. Actris: session-type based reasoning in separation logic. *Proc. ACM Program. Lang. 4*, POPL (2020), 6:1–6:30.

[16] Hobor, A., and Gherghina, C. Barriers in concurrent separation logic. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings* (2011), pp. 276–296.

[17] Hobor, A., and Gherghina, C. Barriers in concurrent separation logic: Now with tool support! *Log. Methods Comput. Sci. 8*, 2 (2012).

[18] Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., and Piessens, F. Verifast: A powerful, sound, predictable, fast verifier for C and java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings* (2011), M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., vol. 6617 of *Lecture Notes in Computer Science*, Springer, pp. 41–55.

[19] Jung, R., Krebbers, R., Jourdan, J., Bizjak, A., Birkedal, L., and Dreyer, D. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program. 28* (2018), e20.

[20] Jung, R., Lepigre, R., Parthasarathy, G., Rapoport, M., Timany, A., Dreyer, D., and Jacobs, B. The future is ours: prophecy variables in separation logic. *Proc. ACM Program. Lang. 4*, POPL (2020), 45:1–45:32.

[21] Leino, K. R. M., Müller, P., and Smans, J. Deadlock-free channels and locks. In *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings* (2010), pp. 407–426.

[22] Lozes, É., and Villard, J. Shared contract-obedient channels. *Sci. Comput. Program. 100* (2015), 28–60.

[23] Mével, G., Jourdan, J., and Pottier, F. Time credits and time receipts in iris. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings* (2019), pp. 3–29.

[24] Müller, P., Schwerhoff, M., and Summers, A. J. Viper: A verification infrastructure for permission-based reasoning. In *Dependable Software Systems Engineering*, vol. 50 of *NATO Science for Peace and Security Series - D: Information and Communication Security*. IOS Press, 2017, pp. 104–125.

[25] O'Hearn, P. W., Yang, H., and Reynolds, J. C. Separation and information hiding. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004* (2004), pp. 268–280.

[26] Parkinson, M. J. The next 700 separation logics - (invited paper). In *Verified Software: Theories, Tools, Experiments, Third International Conference, VSTTE 2010, Edinburgh, UK, August 16-19, 2010. Proceedings* (2010), pp. 169–182.

[27] Pottier, F. Hiding local state in direct style: A higher-order anti-frame rule. In *Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA* (2008), pp. 331–340.

[28] Sampaio, G., Santos, J. F., Maksimovic, P., and Gardner, P. A trusted infrastructure for symbolic analysis of event-driven web applications. In *34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference)* (2020), pp. 28:1–28:29.

[29] Santos, J. F., Maksimovic, P., Ayoun, S., and Gardner, P. Gillian, part i: a multi-language platform for symbolic execution. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020* (2020), pp. 927–942.

[30] Schwinghammer, J., Birkedal, L., Pottier, F., Reus, B., Støvring, K., and Yang, H. A step-indexed kripke model of hidden state. *Math. Struct. Comput. Sci. 23*, 1 (2013), 1–54.

[31] Vafeiadis, V., and Parkinson, M. J. A marriage of rely/guarantee and separation logic. In *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings* (2007), pp. 256–271.

[32] Villard, J., Lozes, É., and Calcagno, C. Proving copyless message passing. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings* (2009), pp. 194–209.

A reactive operational semantics for a lambda-calculus with time warps

Adrien Guatto[1], Christine Tasson[2], and Ada Vienot[1]

[1] Université de Paris, CNRS, IRIF, Paris, 75006, France
guatto@irif.fr vienot@irif.fr
[2] Sorbonne Université, LIP6, Paris, 75005, France
christine.tasson@lip6.fr

This work is supported by the Emergence project "ReaLiSe" funded by the city of Paris.

Abstract

Many computer systems are *reactive*: they execute in continuous interaction with their environment. From the perspective of functional programming, reactive systems can be modeled and programmed as stream functions. Several lines of research have exploited this point of view. Works based on Nakano's guarded recursion use type-theoretical modalities to guarantee productivity. Works based on synchronous programming use more specialized methods but guarantee incremental processing of stream fragments.

In this paper, we contribute to a recent synthesis between the two approaches by describing how a language with a family of type-theoretical modalities can be given an incremental semantics in the synchronous style. We prove that this semantics coincides with a more usual big-step semantics.

1 Introduction

Reactive systems can be found in a variety of application domains, ranging from hard real-time and life-critical systems such as airplane control to more mundane areas like video games and graphical user interfaces. Their specificity as computer systems lies in their continuous interaction with an external environment.

At a low level of abstraction, reactive systems can be modeled as state machines. Yet, reasoning about state machines and, more generally, state transition systems requires beautiful but complex notions of bisimilarity. Furthermore, in practice, the dominant approach to the implementation of state machines is via a mix of callbacks and mutable state, which makes compositional reasoning delicate. These difficulties have motivated a lot of works in the imperative world, such as the well-known model-view-controller pattern.

An alternative approach to the programming of reactive systems is to use functional programming with infinite data structures. In this view, a reactive program is seen as a pure function from some infinite data structure describing incoming events to some other infinite data structure describing outgoing events. This idea can be traced back to the seminal work of Kahn [10], in which he observed how interacting parallel processes could be described as well-behaved functions between infinite sequences — or *streams*, for short. Thanks to this approach, all the classical tools from functional programming (e.g., higher-order functions) can be applied to reactive systems.

The idea of reactive programming with pure functions has been fertile, and has developed into several distinct lines of research. To introduce these works, and explain how the present paper fits into them, let us consider the example below, given in functional pseudocode.

```
sum : Stream Nat -> Stream Nat
sum xs = ys where rec ys = map2 (+) xs (0 :: ys)
```

This function takes a stream of integers and returns its running sum, that is, the ith element of the output stream `ys` is the sum of the first i elements of the input stream `xs`. Here `(::)` denotes the stream constructor and `map2 f xs ys` applies the function `f` to `xs` and `ys` pointwise.

Handling stream functions like `sum` in a programming language for reactive programming raises two interconnected questions.

1. What constructions and data types does the language offer, and under what conditions? For example, should one allow arbitrary higher-order functions? Arbitrary recursive definitions? Other infinite data types such as infinite trees or streams of streams?
2. How is the language implemented? A reactive program, being in continuous interaction with its environment, ought to be able to process streams in an incremental fashion. Moreover, this processing should use as little space and time as possible.

Different research lines have answered these questions differently, leading to distinct tradeoffs between expressiveness, safety, and efficiency.

A first answer is that of *synchronous functional programming* languages in the vein of Lustre [5], which favor efficiency and safety over expressiveness. In synchronous functional programming, base types are that of streams of scalars, computed along an infinite sequence of execution *ticks*. In exchange for this restrictive focus, synchronous languages benefit from specialized yet powerful type systems:

- *causality types* classify *productive* recursive definitions, e.g., rejecting a variant of `sum` without the initial 0 on the last argument to `map2`;
- *clock types* classify stream types according to its *clock*: the set of ticks at which a new stream element must appear.

These analyses serve as a foundation to the backends of synchronous compilers, which turn clocked stream functions into state machines realized as memory-frugal imperative code.

While well-suited to hard real-time and life-critical systems, synchronous programming can feel restrictive from the perspective of mainstream functional programming. Indeed, even an expressive higher-order synchronous language such as Lucid Synchrone [6, 16] does not support streams of streams, nor streams of arbitrary functions. This is to be contrasted with the situation in a second approach to reactive programming, *functional reactive programming*.

Functional reactive programming, as originally proposed by Elliott and Hudak [8], consists in realizing infinite data structures by way of laziness in a language such as Haskell. The expressiveness of Haskell makes it simple to write concise and elegant reactive code, making it especially relevant for prototyping. Unfortunately, using the full power of a lazy language and its facilities for general recursion comes at a cost. This cost is the ease with which one can inadvertently write a non productive definition or introduce unbounded memory usage [13].

To be used in applications with performance or safety needs, functional reactive programming has to be tamed. This can be achieved in two ways. One possibility is to implement a restrictive domain-specific language for reactive programming as a Haskell library. From this perspective, Haskell serves as a powerful macro language for the embedded language. This is the approach used in, e.g., *causal commutative arrows* [13].

$$\textsc{GuardedRec}\quad \frac{\Gamma, x : \triangleright A \vdash M : A}{\Gamma \vdash \mathsf{rec}\, x.M : A}$$

Figure 1 Guarded recursion

Another choice is to drop Haskell and develop a standalone domain-specific language. This approach has become increasingly popular since the seminal work of Krishnaswami and Benton [12]. They remarked that a certain unary connective (or *modality*) studied in foundational works on modal logic [15] and denotational

semantics [1, 4] can be used to give a type-theoretical criterion for productive recursion. This criterion consists in Figure 1. The type $\triangleright A$ (pronounced "later A") is a slightly weaker version of A that does not give any information at the beginning of execution. This *guarded* recursion rule, by requiring the body of the definition to have type $\triangleright A \to A$, makes sure that the definition actually extends `x` before using it. In particular, computing the fixed point of the identity function is no longer permitted. This simple idea has led to many developments, including memory-conscious implementations [11, 2].

Interestingly, the introduction of the later modality in functional reactive programming brings it closer to synchronous concepts. Indeed, the type $\triangleright A$ should be thought of as "A but without the data available at the first execution tick." For example, the type $\triangleright\,\mathsf{Stream}\,\mathbb{N}$ is that of streams that start growing at the second tick rather than the first one. Thus, the later modality introduces an idea of global discrete time akin to that found in synchronous programming. Moreover, its effect on types is tantalizingly similar to that of the clocks used in synchronous programming. In light of these similarities, the first author built a unifying calculus, termed λ^*, that bridges synchronous programming and functional reactive programming [9].

In λ^*, the later modality arises as a special case of a more general *warping modality*. The warping modality $p \Rrightarrow -$ acts upon a type as prescribed by the *time warp* denoted p. A time warp is, briefly speaking, a well-behaved map from the discrete time scale to itself. The type $p \Rrightarrow A$ should be thought of as a variant of A suitably stretched or compressed so that, at tick n, one can observe from it what can be observed from A at tick $p(n)$. The calculus can be thought of as connecting two lines of research in the sense that all synchronous clocks are time warps and all modalities used in guarded recursion can be described as time warps. Moreover, many time warps where hitherto unavailable as type connectives.

In addition to a type system, λ^* comes equipped with an execution mechanism expressed as an operational semantics. It can be seen as an evaluator parameterized with a quantity of fuel. It is a theorem of λ^* that a closed term of type $\mathsf{Stream}\,\mathbb{B}$, when evaluated with n units of fuel, converges to a list of n booleans. This list is a prefix of the ideal conceptually-infinite stream computed by the program. While this is satisfactory from a metatheoretical perspective, this process leaves much to be desired in the reactive setting. The evaluation process is not incremental: the evaluation process with $n+1$ units of fuel cannot reuse any of the computations that arose while computing the shorter prefixes.

Contributions. In this paper, we continue to bridge the gap between synchronous functional programming and functional reactive programming by introducing an incremental semantics for λ^*. Taking inspiration from compilation techniques introduced for Lucid Synchrone [7, 16], we introduce a notion of typed program *states* that store results to be reused, and replace prefix values with collections of instantaneous deltas that we call *increments*. We prove by a logical-relation argument that our incremental semantics is sound: glueing together the consecutive increments recovers the prefix computed by the reference semantics.

In Section 2, we introduce the first author's λ^* calculus, and our modifications of its type system needed to type-check program states. We then describe our incremental semantics in depth in Section 3, including its metatheory and the adequacy theorem for our logical relation. Finally, in Section 4 we discuss our results and their relation to previous works.

Notations. We write ω for the first infinite ordinal and $\omega+1$ for its successor ordinal $\omega \cup \{\omega\}$. Additition and subtraction extend to $\omega+1$ by setting $n+\omega=\omega$ and $n-\omega=0$ for all $n \in \omega+1$. The *domain* of a partial map $f : X \rightharpoonup Y$, denoted $\mathrm{dom}(f)$, is defined as $f^{-1}(Y)$. A *finite map* is a partial map of finite domain. Given a finite map f, the notation $f(x)$ implies $x \in \mathrm{dom}(f)$. We write $f, x : y$ or $f[y/x]$ for the finite map sending x to y and $x' \in \mathrm{dom}(f) \setminus \{x\}$ to $f(x')$.

2 The λ^* calculus

In order to save space, we leave aside the treatment of product and sum types in the short version of this paper. A complete treatment of λ^* can be found in the full version of the paper, available at http://hal.archives-ouvertes.fr/hal-03465519.

2.1 Type system

Time warps and terms. In reactive languages, types are implicitly indexed by logical time steps. In λ^*, this indexing can be changed through the use of the warping modality $p \Rrightarrow -$. To explain how this modality works, we must first define what a time warp p exactly is.

Definition 2.1. A *time warp* is a suprema-preserving map $p : \omega + 1 \to \omega + 1$.

Equivalently, a time warp is a monotonic map from $\omega + 1$ to itself such that $p(0) = 0$ and $p(\omega) = \max_{i<\omega} p(i)$. Time warps are denoted p, q, r. The time warps **id** and **lat** respectively correspond to the maps $n \mapsto n$ and $n \mapsto n - 1$.

Time warps appear in the terms of λ^* as well as in its types: they decorate the term constructors $\mathsf{delay}^{q \leqslant p}(M)$ and $M\,\mathsf{by}\,p$. The other terms of the language are those of a simply-typed λ-calculus with streams, constants c belonging to some fixed set C, and recursive definitions.

$$M, N ::= x \mid c \mid \lambda x^A.\, M \mid MN \mid \mathsf{rec}\, x^A.\, M \mid M :: N \mid \mathsf{hd}\, M \mid \mathsf{tl}\, M \mid M\,\mathsf{by}\,p \mid \mathsf{delay}^{q \leqslant p}(M)$$

Types. The types of λ^* are those of the simply-typed λ-calculus with a fixed set of ground types G (whose elements are denoted ν) and streams, as well as the warping modality $p \Rrightarrow (-)$.[1]

$$A, B ::= \nu \mid \mathsf{Str}\, A \mid A \overset{S}{\Rightarrow} B \mid p \Rrightarrow A$$

The label S annotating the arrow type $A \overset{S}{\Rightarrow} B$ is new to the present paper. It is linked to the proposed incremental semantics. We explain the purpose this *state type* S serves in Section 3.

Informally, the values of type $p \Rrightarrow A$ at tick n are those of A at tick $p(n)$. Another possible way to think of $p \Rrightarrow A$ is as a type where time passes "p-times" faster than in type A. For example, a value of type $p \Rrightarrow \mathsf{Str}\,\mathsf{Nat}$ at tick n is a list of $p(n)$ numbers.

We assume that for every ground type ν there exists a set $C_\nu \subseteq C$ such that every constant $c \in C$ belongs to exactly one such C_ν.

Typing contexts. Typing contexts are finite maps from identifiers to types, and are denoted Γ, Δ. We write $p \Rrightarrow \Gamma$ for the typing context sending $x \in \mathrm{dom}(\Gamma)$ to $p \Rightarrow \Gamma(x)$.

Typing judgements. The typing judgment $\Gamma \vdash M : A \mid S$ states that if the free variables of M respect the types prescribed by Γ then M computes a result of type A using a state of type S. Its rules are given in Figure 2. Most of them are those of the simply-typed λ-calculus, thus we focus our explanation on the rules specific to temporal constructs. The state-type aspect is new and we discuss it in the next section.

In λ^*, the type $\mathsf{Str}\, A$ classifies streams that unfold at the rate of one element per step. This is reflected by the rules Head, Tail, and Cons, which state that the head of a stream is available now while its tail is only available at type $\mathbf{lat} \Rrightarrow \mathsf{Str}\, A$. This type describes streams which start to grow strictly after the first tick. Thus, the ith element of a stream of type $\mathsf{Str}\, A$ is only available at tick i.

[1] The warping modality was denoted $p * (-)$ in the original λ^* paper [9].

$$\boxed{\Gamma \vdash M : A \mid S}$$

$$\textsc{Var}\ \frac{}{\Gamma, x{:}A \vdash x : A \mid \mathbb{1}} \qquad \textsc{Scalar}\ \frac{c \in C_\nu}{\Gamma \vdash c : \nu \mid \mathbb{1}} \qquad \textsc{Lam}\ \frac{\Gamma, x{:}A \vdash M : B \mid S}{\Gamma \vdash \lambda x^A.\, M : A \overset{S}{\Rightarrow} B \mid \mathbb{1}}$$

$$\textsc{App}\ \frac{\Gamma \vdash M : A \overset{S''}{\Longrightarrow} B \mid S \qquad \Gamma \vdash N : A \mid S'}{\Gamma \vdash MN : B \mid S \times S' \times S''} \qquad \textsc{Head}\ \frac{\Gamma \vdash M : \mathsf{Str}\, A \mid S}{\Gamma \vdash \mathsf{hd}\, M : A \mid S} \qquad \textsc{Tail}\ \frac{\Gamma \vdash M : \mathsf{Str}\, A \mid S}{\Gamma \vdash \mathsf{tl}\, M : \mathbf{lat} \Rightarrow \mathsf{Str}\, A \mid S}$$

$$\textsc{Cons}\ \frac{\Gamma \vdash M : A \mid S \qquad \Gamma \vdash N : \mathbf{lat} \Rightarrow \mathsf{Str}\, A \mid S'}{\Gamma \vdash M :: N : \mathsf{Str}\, A \mid S \times S'} \qquad \textsc{Rec}\ \frac{\Gamma, x{:}\mathbf{lat} \Rightarrow A \vdash M : A \mid S}{\Gamma \vdash \mathsf{rec}\, x^A.\, M : A \mid S \times \int A} \qquad \textsc{By}\ \frac{\Gamma \vdash M : A \mid S}{p \Rightarrow \Gamma \vdash M\, \mathsf{by}\, p : p \Rightarrow A \mid p \Rightarrow S}$$

$$\textsc{Delay}\ \frac{q \leqslant p \qquad \Gamma \vdash M : p \Rightarrow A \mid S}{\Gamma \vdash \mathsf{delay}^{q \leqslant p}(M) : q \Rightarrow A \mid S \times \int(p \Rightarrow A)}$$

Figure 2 Typing judgement for λ^*

Rule Rec is that of guarded recursive definitions. It mirrors the rule GuardedRec described in the introduction, but expresses the later modality $\triangleright$ as a special application of the warping modality with **lat** as time warp. This rule enforces productivity by forbidding instantaneous dependencies in recursive definitions.

The by construction (Rule By) is used to introduce and eliminate the warp modality. It states that if a program M produces an output at rate A by consuming inputs at rate Γ, then one can always speed M up to produce an output at rate $p \Rightarrow A$, at the cost of consuming inputs at rate $p \Rightarrow \Gamma$, for any p.

If $q(n) \leqslant p(n)$ then the values of type $p \Rightarrow A$ at the nth tick can be *truncated* into values of type $q \Rightarrow A$ at the nth tick. Rule Delay generalizes this reasoning to all ticks: the delay construction turns results of type $p \Rightarrow A$ into results of type $q \Rightarrow A$ as long as $q \leqslant p$, where $\leqslant$ denotes the pointwise ordering. We describe the operational content of truncation in the next section.

2.2 Prefix semantics

The terms of λ^* manipulate ideal, infinite data structures, but they have to execute on a concrete machine that only deals with finite syntax. For this reason, the reference semantics of the language computes finite yet arbitrarily precise approximants, or *prefixes*, of streams and other infinite data structures. The length of these prefixes is controlled by a parameter to the evaluation judgment, which can be thought of alternatively as the current tick or as a quantity of fuel (as in step-indexing [1]). Using this parameter, one may evaluate, e.g., a program computing the stream of natural numbers at tick 1 and obtain the prefix $0 :: \mathbf{stop}$, or evaluate at tick 2 and obtain the prefix $0 :: 1 :: \mathbf{stop}$, and so on. For this reason, we call this evaluation semantics the *prefix semantics*.

Prefix values and environments. Prefix values, denoted V, should be thought of as prefixes of a certain length of an infinite object of a certain type. A prefix of length n is the result of a computation at tick n. Figure 3 describes the typing rules of the judgment $V : A \mathbin{\mathfrak{C}} n$, which states that V is a prefix of length n of an object of type A.

The special value **stop** is a prefix of every value at 0. Prefixes at ω should be infinite but

$$\boxed{V : A \mathbin{@} n}$$

$$\textsc{VStop}\ \frac{}{\mathbf{stop} : A \mathbin{@} 0} \qquad \textsc{VBox}\ \frac{\Gamma \vdash M : A \mid S \qquad E : \Gamma \mathbin{@} \omega}{\mathbf{box}(M)\{E\} : A \mathbin{@} \omega} \qquad \textsc{VScalar}\ \frac{c \in C_\nu}{c : \nu \mathbin{@} n+1}$$

$$\textsc{VWarp}\ \frac{V : A \mathbin{@} p(n+1)}{\mathbf{w}\,(p, V) : p \Rightarrow A \mathbin{@} n+1} \qquad \textsc{VClo}\ \frac{\Gamma,\, x{:}A \vdash M : B \mid S \qquad E : \Gamma \mathbin{@} n+1}{(x.M)\{E\} : A \overset{S}{\Rightarrow} B \mathbin{@} n+1} \qquad \textsc{VStream}\ \frac{V : A \mathbin{@} n+1 \qquad V' : \mathsf{Str}\, A \mathbin{@} n}{V :: V' : \mathsf{Str}\, A \mathbin{@} n+1}$$

$$\boxed{E : \Gamma \mathbin{@} n} \qquad \textsc{VEnv}\ \frac{\mathrm{dom}(\Gamma) = \mathrm{dom}(E) \qquad \forall x \in \mathrm{dom}(\Gamma).\ E(x) : \Gamma(x) \mathbin{@} n}{E : \Gamma \mathbin{@} n}$$

Figure 3 Typing judgement for prefixes

are modeled as suspended computations (*thunks*). All other rules apply for finite positive n. Warped values of type $p \Rightarrow A$ at n are prefixes of length $p(n)$ of A. Functional values are closures. Values $V :: V'$ of type $\mathsf{Str}\, A$ are formed of a head value V at n and of a tail value V' at $n-1$.

Environments, denoted E, are partial maps from identifiers to values. We write $\mathbf{w}\,(p, E)$ for the environment E' sending $x \in \mathrm{dom}(E)$ to $\mathbf{w}\,(p, E(x))$.

$$\boxed{\lfloor V \rfloor_n} \qquad \lfloor \mathbf{w}\,(p, V) \rfloor_{n+1} = \mathbf{w}\left(p, \lfloor V \rfloor_{p(n+1)}\right)$$

$$\lfloor V \rfloor_0 = \mathbf{stop} \qquad \lfloor (x.M)\{E\} \rfloor_{n+1} = (x.M)\{\lfloor E \rfloor_{n+1}\}$$

$$\lfloor V \rfloor_\omega = V \qquad \lfloor V_1 :: V_2 \rfloor_{n+1} = \lfloor V_1 \rfloor_{n+1} :: \lfloor V_2 \rfloor_n$$

$$\lfloor c \rfloor_{n+1} = c \qquad \lfloor \mathbf{box}(M)\{E\} \rfloor_{n+1} = V \text{ if } M\,;\lfloor E \rfloor_{n+1} \Downarrow_{n+1} V$$

Figure 4 Truncation of prefixes

As mentioned before, prefixes can be truncated. Given a prefix V of length m and $n \leq m$, one may obtain a shorter prefix $\lfloor V \rfloor_n$ as defined in Figure 4. This extends to environments pointwise. Since the evaluation judgment is deterministic [9], $\lfloor - \rfloor_n$ defines a partial (because of the **box** case) function rather than a relation.

The special value **stop** is a 0-length prefix of every other value. Truncating at ω does nothing. Truncating a closure means truncating its environment. Truncating a stream prefix of length $n > 0$ involves truncating its tail at length $n-1$; for example $\lfloor 1 :: 2 :: \mathbf{stop} \rfloor_1 = 1 :: \mathbf{stop}$. To truncate a thunk one has to evaluate it.

In addition to being total on well-typed terms, truncation preserves typing [9].

Theorem 2.1. *If $V : A \mathbin{@} n$, then for all $m \leqslant n$, we have $\lfloor V \rfloor_m : A \mathbin{@} m$.*

Evaluation judgement. The judgment given in Figure 5 is mostly that of a big-step semantics for call-by-value λ-calculus with environments and closures, extended to depend on a time step n.

We explain the rules that have a time-specific aspect. Rule PZero states that a program evaluated at time 0 always returns the same initial value named **stop**. Rule POmega freezes both the code and the environment in a thunk. When an actual value is required, the program can then be evaluated as needed to any finite length. Rule PBy formalizes the idea that type $p \Rightarrow A$ is composed of p-long prefixes of type A. Rule PRec is reminiscent of the Kleene fixed point theorem

as, in essence, it computes the value of $\mathsf{rec}\, x.\, M$ at n as the iterated application $(\lambda x.\, M)^n(\mathbf{stop})$. This process is formalized by the *iteration* judgment $M\,;E\,;x\,;V \Uparrow_m^n V'$, whose iterative behavior is captured by the lemma below.

Lemma 2.1. *If* $M\,;E\,;x\,;V \Uparrow_i^k V'$ *then for all* j *such that* $i \leqslant j \leqslant k$, *there is* V'' *such that* $M\,;\lfloor E\rfloor_j\,;x\,;V \Uparrow_i^j V''$ *and* $M\,;E\,;x\,;V'' \Uparrow_j^k V'$.

$$\boxed{M\,;E \Downarrow_n V}$$

$$\textsc{PZero}\ \frac{}{M\,;E \Downarrow_0 \mathbf{stop}} \qquad \textsc{POmega}\ \frac{}{M\,;E \Downarrow_\omega \mathbf{box}(M)\{E\}} \qquad \textsc{PScalar}\ \frac{}{c\,;E \Downarrow_{n+1} c} \qquad \textsc{PVar}\ \frac{x \in \mathrm{dom}(E)}{x\,;E \Downarrow_{n+1} E(x)}$$

$$\textsc{PLam}\ \frac{}{\lambda x^A.\, M\,;E \Downarrow_{n+1} (x.M)\{E\}} \qquad \textsc{PApp}\ \frac{M\,;E \Downarrow_{n+1} (x.P)\{E'\} \qquad N\,;E \Downarrow_{n+1} V \qquad P\,;E'[V/x] \Downarrow_{n+1} V'}{MN\,;E \Downarrow_{n+1} V'}$$

$$\textsc{PHead}\ \frac{M\,;E \Downarrow_{n+1} V :: V'}{\mathsf{hd}\, M\,;E \Downarrow_{n+1} V} \qquad \textsc{PTail}\ \frac{M\,;E \Downarrow_{n+1} V :: V'}{\mathsf{tl}\, M\,;E \Downarrow_{n+1} \mathbf{w}\,(\mathbf{lat}, V')} \qquad \textsc{PCons}\ \frac{M\,;E \Downarrow_{n+1} V \qquad N\,;E \Downarrow_{n+1} \mathbf{w}\,(\mathbf{lat}, V')}{M :: N\,;E \Downarrow_{n+1} V :: V'}$$

$$\textsc{PBy}\ \frac{M\,;E \Downarrow_{p(n+1)} V}{M\ \mathsf{by}\, p\,;\mathbf{w}\,(p,E) \Downarrow_{n+1} \mathbf{w}\,(p,V)} \qquad \textsc{PRec}\ \frac{M\,;E\,;x\,;\mathbf{stop} \Uparrow_0^{n+1} V}{\mathsf{rec}\, x^A.\, M\,;E \Downarrow_{n+1} V} \qquad \textsc{PDelay}\ \frac{M\,;E \Downarrow_{n+1} \mathbf{w}\,(p,V)}{\mathsf{delay}^{q\leqslant p}(M)\,;E \Downarrow_{n+1} \mathbf{w}\left(q, \lfloor V\rfloor_{q(n+1)}\right)}$$

$$\boxed{M\,;E\,;x\,;V \Uparrow_m^n V'}$$

$$\textsc{PIFinish}\ \frac{}{M\,;E\,;x\,;V \Uparrow_n^n V} \qquad \textsc{PIIter}\ \frac{m < n \qquad M\,;E\,;x\,;V' \Uparrow_{m+1}^n V'' \qquad M\,;E[\mathbf{w}\,(\mathbf{lat}, V)/x] \Downarrow_{m+1} V'}{M\,;E\,;x\,;V \Uparrow_m^n V''}$$

Figure 5 Prefix evaluation judgement

As an illustration on how the prefix evaluation works, consider the following program defining the stream of natural numbers. The function $\mathsf{incr} : \mathsf{Str}\,\mathsf{Int} \overset{\mathbb{1}}{\Rightarrow} \mathsf{Str}\,\mathsf{Int}$ adds one to all of the constituents of a stream of integers. For the example to be well-typed, we need to use the function $\mathrm{shift}_A : A \overset{\mathbb{1}}{\Rightarrow} (\mathbf{lat} \Rightarrow A)$, which acts as a causality-preserving type coercion.[2]

$$\mathsf{let}\ \mathrm{incr}' = \mathrm{shift}\ \mathrm{incr}\ \mathsf{in}\ \mathsf{let}\ \mathsf{rec}\ \mathrm{nat} = 0 :: (\mathrm{incr}'\,\mathrm{nat}\ \mathsf{by}\ \mathbf{lat})$$

The table below gives the first prefixes produced by evaluating the program nat.

n	1	2	3
nat	$0 :: \mathbf{stop}$	$0 :: 1 :: \mathbf{stop}$	$0 :: 1 :: 2 :: \mathbf{stop}$

The program nat is recursive. The fixed point rule evaluates the body of the rec block by recomputing the value obtained at the previous tick, which is incremented by calling incr′. At the first step, this yields **stop**. For the recursion to be guarded, the by **lat** wraps it under a special constructor to indicate its provenance: this yields $\mathbf{w}\,(\mathbf{lat}, \mathbf{stop})$. Finally, 0 is concatenated to it, yielding $0 :: \mathbf{stop}$. Next step, the process is repeated: we obtain $0 :: \mathbf{stop}$ again at the first tick. We increment it using incr′, which yields $1 :: \mathbf{stop}$, and wrap it under a constructor, turning it into $\mathbf{w}\,(\mathbf{lat}, 1 :: \mathbf{stop})$. Concatenating 0 to it yields $0 :: 1 :: \mathbf{stop}$, and so on.

[2] For brevity's sake, we do not make explicit how this function works. An explanation can be found in Guatto's original paper[9]. We discuss this choice in more detail in Section 4.1.

Metatheory. We state the correctness theorems of [9] that will be important to the rest of the paper. Combined, they express that the evaluation of well-typed terms defines a total function, and that this function is monotonic with respect to truncation.

Theorem 2.2 (Subject reduction)**.** *If* $\Gamma \vdash M : A \mid S$ *and* $M\,;E \Downarrow_n V$ *with* $E : \Gamma \mathbin{@} n$*, then* $V : A \mathbin{@} n$*.*

Theorem 2.3 (Totality)**.** *If* $\Gamma \vdash M : A \mid S$ *and* $E : \Gamma \mathbin{@} n$*, then there exists* V *such that* $M\,;E \Downarrow_n V$*.*

Theorem 2.4 (Determinism)**.** *If* $M\,;E \Downarrow_n V_1$ *and* $M\,;E \Downarrow_n V_2$*, then* $V_1 = V_2$*.*

Theorem 2.5 (Monotonicity)**.** *If* $m \leqslant n$ *and* $M\,;E \Downarrow_n V$ *then* $M\,;\lfloor E \rfloor_m \Downarrow_m \lfloor V \rfloor_m$*.*

3 Reactive semantics for λ^*

In this section, we present a new semantics for the λ^* calculus, quite different from the prefix one. In this semantics, program execution consists in a sequence of reaction steps, each of which consumes a slice of the inputs, produces a slice of the outputs, and updates some internal state. To define this semantics, we first describe states and their types, as well as *increments*, a new sort of values describing these "slices." We show in Section 3.4 that this semantics computes the same results as the prefix semantics, in the appropriate sense.

3.1 Increments and states

Increments. Increments, denoted δ, are syntactic objects representing the information gained by running the program for one step. For example, each time a program of type $\mathsf{Str}\,\mathsf{Nat}$ does a step, it produces a number. The syntax of increments is defined by the following grammar.

$$\delta, \delta' ::= \mathbf{nil} \mid c \mid (x.M)\{\gamma\} \mid \delta :: \delta' \mid \triangleright(\delta_i)_i$$

The null increment is denoted **nil**, c denotes a constant, $(x.M)\{\gamma\}$ denotes an incremental closure, $\delta :: \delta'$ denotes a temporal sequence of increments and $\triangleright(\delta_i)_{i<n}$ denotes a warped family of increments.

Incremental environments, denoted γ, are finite maps from identifiers to increments. If the context permits no ambiguity, they will also be referred to simply as "environments" for the sake of brevity. By abuse of notation, if $(\gamma_i)_{i<n}$ is a family of environments with a common domain D, we write $\triangleright(\gamma_i)_{i<n}$ for the environment sending $x \in D$ to $\triangleright(\gamma_i(x))_{i<n}$.

Families of increments and typing. Another intuition behind increments is that they represent one-step-wide slices of prefixes. A collection of n increments then represents a prefix of length n. Such collections are represented by possibly infinite families of increments $(\delta_i)_{i<n}$. The empty family is written ε. The concatenation of two families of increments $(\delta_i)_{i<n}$ and $(\delta'_i)_{i<m}$, denoted $(\delta_i)_{i<n} \oplus (\delta'_i)_{i<m}$, is the family $(\delta''_i)_{i<n+m}$ such that δ''_i is either δ_i when $i < n$ or δ'_{i-m} when $m \leqslant i < m + n$.

As families of increments are intended to represent prefixes, they are typed according to the judgement rules given in Figure 6. We explain three interesting rules.

Rule ISCALAR states that scalars do not evolve as time passes. They yield all of their informational content at the very first tick, and they bring no additional information during subsequent ticks. Thus, families of increments of a scalar type comprise an initial scalar value followed by the null increment **nil** forever after.

$$\boxed{(\delta_i)_{i<n} : A \mathbin{@} n}$$

$$\textsc{IStop}\ \frac{}{\varepsilon : A \mathbin{@} 0} \qquad \textsc{IScalar}\ \frac{c \in C_\nu}{(c) \oplus (\mathbf{nil})_{i<n} : \nu \mathbin{@} n+1}$$

$$\textsc{IWarp}\ \frac{(\delta_i)_{i<p(n+1)} : A \mathbin{@} p(n+1)}{\left(\triangleright\, (\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n+1} : p \Rrightarrow A \mathbin{@} n+1} \qquad \textsc{IClo}\ \frac{\Gamma,\, x{:}A \vdash M : B \mid S \qquad (\gamma_i)_{i<n+1} : \Gamma \mathbin{@} n+1}{\left((x.M)\{\gamma_i\}\right)_{i<n+1} : A \overset{S}{\Rightarrow} B \mathbin{@} n+1}$$

$$\textsc{IStream}\ \frac{(\delta_i)_{i<n+1} : A \mathbin{@} n+1 \qquad (\delta'_i)_{i<n+1} : \mathbf{lat} \Rrightarrow \mathsf{Str}\, A \mathbin{@} n+1}{(\delta_i :: \delta'_i)_{i<n+1} : \mathsf{Str}\, A \mathbin{@} n+1} \qquad \textsc{IBox}\ \frac{\forall m \in \omega.\ (\delta_i)_{i<m} : A \mathbin{@} m}{(\delta_i)_{i<\omega} : A \mathbin{@} \omega}$$

$$\boxed{(\gamma_i)_{i<n} : \Gamma \mathbin{@} n} \qquad \textsc{IEnv}\ \frac{\forall i < n.\ \ \mathrm{dom}(\Gamma) = \mathrm{dom}(\gamma_i) \qquad \forall x \in \mathrm{dom}(\Gamma).\ (\gamma_i(x))_{i<n} : \Gamma(x) \mathbin{@} n}{(\gamma_i)_{i<n} : \Gamma \mathbin{@} n}$$

Figure 6 Typing judgement for λ^* increments

Rule IWarp expresses that flattening the n lists under the $\triangleright$ constructor yields a family of size $p(n)$. Recall that a prefix of $p \Rrightarrow A$ at time n is a prefix of A at time $p(n)$. Thus, while one step from time $n-1$ to n is performed on the outside of the program, $\partial p(n) = p(n) - p(n-1)$ steps are performed under the by during this time. Since increments represent the informational content of the program during one step, an increment of $p \Rrightarrow A$ at time n is a family of increments of size $\partial p(n)$, encompassing steps from $p(n-1)$ to $p(n)-1$.

Finally, rule IBox is the incremental counterpart to rule VBox. The main difference is that we no longer represent infinite prefixes as thunks (as in the prefix semantics) but as infinite families of increments. For this reason, truncation in the incremental semantics consists in restricting a (possibly infinite) family of increments to one of its initial segments. This is simpler than in the prefix semantics, where truncating a thunk actually consists in evaluating its body. However, this means that a program may return an infinite family of increments in finite time, e.g., as the result of M by p at the ith tick for p such that $p(i) = \omega$. We discuss the consequences of this choice in Section 4.

We now prove a counterpart to Theorem 2.1 by induction over typing derivations. A well-typed family of increments can be truncated and preserve its typing, albeit at a shorter length.

Theorem 3.1 (Type safety of increment truncation)**.** *If* $(\delta_i)_{i<n} : A \mathbin{@} n$ *and* $m \leqslant n$ *then* $(\delta_i)_{i<m} : A \mathbin{@} m$.

Notice that the typing judgement is only applicable on family of increments, and not increments alone. As an illustration, let $x{:}A \vdash M : B \mid S$ and $x{:}A \vdash N : B \mid S$ be two different well-typed terms. Now, consider the family of increments $\left((x.M)\{\}, (x.N)\{\}\right)$ with M and N unrelated terms of the same type. It maps to no prefix of type $A \overset{S}{\Rightarrow} B$, since the code inside the two closures differs. If we could somehow type increments on their own, there would be families of increments that are well-typed but do not correspond to a prefix. This is undesirable, as we want well-typed families of increments to map one-to-one to prefixes.

States and their types. States, denoted s, are syntactic objects. They are classified by a special category of types, *state types*, denoted S, according to the rules given in Figure 7.

$$s, s' ::= \mathbf{stop} \mid \mathbf{done} \mid \mathbf{unit} \mid \mathbf{w}\,(p, s) \mid (s, s') \mid \mathbf{inc}\,((\delta_i)_i)$$
$$S, S' ::= \mathbb{1} \mid p \Rrightarrow S \mid S \times S' \mid \textstyle\int A$$

These rules are similar to the ones used for typing prefixes (cf. Figure 3), except for SDone and SIncr. Rule SDone reflects the fact that a program that has performed ω reactions no longer needs to store any relevant state. We will develop this point shortly. Rule SIncr introduces the type constructor $\int$, which embeds families of increments into states. As such, $\int A$ denotes a "buffer" able to store values of type A — this allows us to store previously computed values into our states for further computations. Notice that it is the only interesting base state type, as the only other one is $\mathbb{1}$, and denotes a stateless computation.

$$\boxed{s : S \,\mathfrak{G}\, n}$$

$$\textsc{SStop}\ \frac{}{\mathbf{stop} : S \,\mathfrak{G}\, 0} \qquad \textsc{SDone}\ \frac{}{\mathbf{done} : S \,\mathfrak{G}\, \omega} \qquad \textsc{SUnit}\ \frac{}{\mathbf{unit} : \mathbb{1} \,\mathfrak{G}\, n+1}$$

$$\textsc{SWarp}\ \frac{s : S \,\mathfrak{G}\, p(n+1)}{\mathbf{w}\,(p, s) : p \Rrightarrow S \,\mathfrak{G}\, n+1} \qquad \textsc{SPair}\ \frac{s_1 : S_1 \,\mathfrak{G}\, n+1 \qquad s_2 : S_2 \,\mathfrak{G}\, n+1}{(s_1, s_2) : S_1 \times S_2 \,\mathfrak{G}\, n+1}$$

$$\textsc{SIncr}\ \frac{(\delta_i)_{i<n+1} : A \ @\ n+1}{\mathbf{inc}\,((\delta_i)_{i<n+1}) : \textstyle\int A \,\mathfrak{G}\, n+1}$$

Figure 7 Typing judgement for states

We can now explain the state-type component S of the typing judgment $\Gamma \vdash M : A \mid S$ that we deliberately ignored in Section 2.1. This state type describes the state that our incremental semantics associates to M. It is handled in the same way as effect annotations in type-and-effect disciplines [14]. In particular, App expresses that an application requires three substates: one for the function, one for the argument, and one for the body of the function. Most other rules follow simpler patterns, with the states from subterms becoming substates of the parent term. Only the rules dealing with the warping modality require further component.

Rule By asserts that the state of $p\,\mathsf{by}\,M$ is just the state of M at rhythm p. This is because running a term under a by is akin to running it under an independant local clock p — thus, its state also grows at this rhythm.

We have seen how the prefix semantics mimicks Kleene's iteration sequence, approximating the fixed point of M at n by $(\lambda x.\, M)^n(\mathbf{stop})$, so to speak. In the incremental semantics, at the $(n+1)$th tick, rather than recompute $n+1$ applications of M we apply M once to the previously-computed result. In order to be able to do so, we include a copy of the output of M in the state of $\mathsf{rec}\,x.\, M$, as expressed by Rec. This behavior is a key ingredient of the incremental semantics.

Rule Delay specifies a nontrivial state type due to our wish to incrementalize the computations. To understand the specified state type, imagine that we receive incremental results from some term M at rythm p, and wish to produce incremental results at rythm q. If q is slow enough with respect to p, we may have to output *now* increments received *previously*. Hence, we choose to store the output of M in the state of $\mathsf{delay}^{q \leqslant p}(M)$. (We discuss how much buffering is needed in Section 4.)

3.2 Evaluation judgements

Discrete derivatives. We are now interested in how data changes from one tick to the next. Thus we define the *discrete derivative* of a time warp p, denoted ∂p, as $n \mapsto p(n) - p(n-1)$.

$$\boxed{M \parallel s\,;\gamma \Longrightarrow s'\,;\delta}$$

$$\textsc{EScalar}\ \frac{}{c \parallel \mathbf{unit}\,;\gamma \Longrightarrow \mathbf{unit}\,;\mathbf{nil}} \qquad \textsc{EScalarI}\ \frac{c \in C_\nu}{c \parallel \mathbf{stop}\,;\gamma \Longrightarrow \mathbf{unit}\,;c}$$

$$\textsc{EVar}\ \frac{x \in \mathrm{dom}(\gamma)}{x \parallel \mathbf{unit}\,;\gamma \Longrightarrow \mathbf{unit}\,;\gamma(x)} \qquad \textsc{ELam}\ \frac{}{\lambda x^A.\,M \parallel \mathbf{unit}\,;\gamma \Longrightarrow \mathbf{unit}\,;(x.M)\{\gamma\}}$$

$$\textsc{EApp}\ \frac{M \parallel s_M\,;\gamma \Longrightarrow s'_M\,;(x.P)\{\gamma'\} \qquad N \parallel s_N\,;\gamma \Longrightarrow s'_N\,;\delta \qquad P \parallel s_P\,;\gamma'[\delta/x] \Longrightarrow s'_P\,;\delta'}{MN \parallel (s_M, s_N, s_P)\,;\gamma \Longrightarrow (s'_M, s'_N, s'_P)\,;\delta'}$$

$$\textsc{EHead}\ \frac{M \parallel s\,;\gamma \Longrightarrow s'\,;\delta :: \delta'}{\mathsf{hd}\, M \parallel s\,;\gamma \Longrightarrow s'\,;\delta} \qquad \textsc{ETail}\ \frac{M \parallel s\,;\gamma \Longrightarrow s'\,;\delta :: \delta'}{\mathsf{tl}\, M \parallel s\,;\gamma \Longrightarrow s'\,;\delta'} \qquad \textsc{ECons}\ \frac{M \parallel s_M\,;\gamma \Longrightarrow s'_M\,;\delta \qquad N \parallel s_N\,;\gamma \Longrightarrow s'_N\,;\delta'}{M :: N \parallel (s_M, s_N)\,;\gamma \Longrightarrow (s'_M, s'_N)\,;\delta :: \delta'}$$

$$\textsc{EBy}\ \frac{M \parallel s\,;(\gamma_i)_{i<\partial p(n)} \overset{\partial p(n)}{\Longrightarrow} s'\,;(\delta_i)_{i<\partial p(n)}}{M\ \mathsf{by}\ p \parallel \mathbf{w}\,(p,s)\,;\triangleright(\gamma_i)_{i<\partial p(n)} \Longrightarrow \mathbf{w}\,(p,s')\,;\triangleright(\delta_i)_{i<\partial p(n)}}$$

$$\textsc{ERecI}\ \frac{M \parallel \mathbf{stop}\,;\gamma[\triangleright\varepsilon/x] \Longrightarrow s'\,;\delta}{\mathsf{rec}\, x^A.\,M \parallel \mathbf{stop}\,;\gamma \Longrightarrow (s', \mathbf{inc}\,(\delta))\,;\delta} \qquad \textsc{ERec}\ \frac{M \parallel s\,;\gamma[\triangleright(\delta_{n-1})/x] \Longrightarrow s'\,;\delta_n}{\mathsf{rec}\, x^A.\,M \parallel \left(s, \mathbf{inc}\,((\delta_i)_{i<n})\right)\,;\gamma \Longrightarrow \left(s', \mathbf{inc}\,((\delta_i)_{i<n+1})\right)\,;\delta_n}$$

$$\textsc{EDelay}\ \frac{M \parallel s\,;\gamma \Longrightarrow s'\,;\triangleright(\delta_i)_{p(n)\leqslant i<p(n+1)}}{\mathsf{delay}^{q\leqslant p}(M) \parallel \left(s, \mathbf{inc}\left(\left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}\right)\right)\,;\gamma \Longrightarrow \left(s', \mathbf{inc}\left(\left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n+1}\right)\right)\,;\triangleright(\delta_i)_{q(n)\leqslant i<q(n+1)}}$$

Figure 8 Single-step evaluation judgement

Single-step evaluation judgement. The main idea behind the incremental semantics is to have a relation that advances a computation by one tick. Such a relation can be pictured by viewing the well-typed term $\Gamma \vdash M : A \mid S$ as an automaton (see Figure 9). It is given a state s of type S, and an incremental environment γ representing a slice of a prefix environment of type Γ. The automaton produces an increment δ representing a slice of the output value of type A, and a new state s' that is ready to be used to advance the program another step. To this end, we introduce the *single-step evaluation judgement*, denoted $M \parallel s\,;\gamma \Longrightarrow s'\,;\delta$. The rules composing this judgement are given in Figure 8.

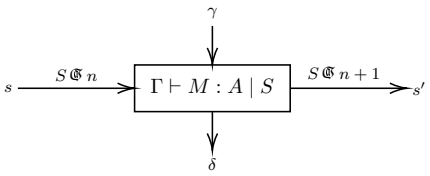


Figure 9 Terms as block diagrams

All rules — except two — that handle cases where the initial state is equal to **stop** have been omitted here for shortness sake.[3] The reason for singling out EScalarI and ERecI is that unlike the other rules, they are not just structural coercions. EScalarI expresses that once a scalar has yielded its full informational content at the first step, it becomes inert and returns the vacuous increment, **nil**. ERecI is in charge of initializing the recursion. It does so by binding $\triangleright\varepsilon$ in the environment, which corresponds to the first increment of a family of type $\mathbf{lat} \Rrightarrow A$.

[3] They can be found in the full version of the paper, available at http://hal.archives-ouvertes.fr/hal-03465519.

We detail three remarkable rules of the judgement.

Rule EBy makes use of the many-step evaluation judgement in its premise, which will formally be introduced in the next paragraph. It corresponds to repeated application of the single-step judgement. When the program M by p makes a step from time $n-1$ to time n, the subterm M makes a step from time $p(n-1)$ to $p(n)$, thus producing $\partial p(n)$ increments. The $\partial p(n)$-family of increments output by the subterm M is then turned into a single increment by the $\triangleright$ constructor. The input and output states are p-shifted in time, and protected under the constructor $\mathbf{w}\,(p,-)$.

Rule ERec is the most surprising one, due to the fact that it only needs to use the increment produced at the previous tick to produce the one for the current step. While it receives by way of its state a family of increments intended to represent the entierety of the previously-computed prefix, it only uses the very last one to produce a result. Apart from this particular observation, the rule behaves similarly to its prefix counterpart PRec, as the premise of ERec behaves similarly to the iteration judgement (cf. Figure 5): a value of type A and length $n-1$ is inserted under a constructor for a value of type $\mathbf{lat} \Rrightarrow A$, and bound to the environment. Once this increment has been produced, it is appended to the input state's increments.

Rule EDelay states that at time n, the subterm M of type $p \Rrightarrow A$ produces a warped increment family encompassing ticks from $p(n-1)$ to $p(n)-1$. To produce an increment of $q \Rrightarrow A$ at time n, we need a warped increment family encompassing ticks from $q(n-1)$ to $q(n)-1$. We buffer the increments produced by the subterm M by appending them to the input state. This yields a family of increments encompassing times 0 to times $p(n)-1$, which becomes a new output. From this family, we extract increments ranging from time $q(n-1)$ to time $q(n)-1$. This is always possible since $q \leqslant p$. By warping this subfamily extracted from the output state, we produce our final result (see Figure 10).

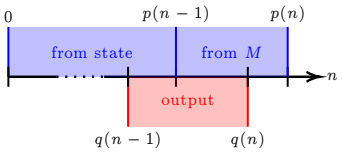


Figure 10 EDelay timings

Recall the program nat given in Section 2.2. The table below gives the first increments produced by evaluating it using the single-step reduction.

n	1	2	3
nat	$0 :: \triangleright[\,]$	$\mathbf{nil} :: \triangleright[1 :: \triangleright[\,]]$	$\mathbf{nil} :: \triangleright[\mathbf{nil} :: \triangleright[2 :: \triangleright[\,]]]$

The fixed point rule evaluates the body under rec. At the first tick, the warped empty increment $\triangleright[\,]$ is bound in the context. Applying incr′ on it leaves it unchanged. Notice that at the very first tick, the program '0' returns its value as an increment. Thus, when '$0::$' is called, it produces the increment $0 :: \triangleright[\,]$. The result is then stored in the state before being returned by the whole program. At the next tick, we take this state, and unwrap rec again. This time, the previous increment is $0 :: \triangleright[\,]$, and applying incr′ on it yields $1 :: \triangleright[\,]$. Then, we call '$0::$' on it. However, since we are not at the first tick, the program '0' is inert and produces the increment **nil**. Thus, when evaluated at tick 1, the program returns the increment $\mathbf{nil} :: \triangleright[1 :: \triangleright[\,]]$. The same reasoning applies for subsequent evaluations.

Many-step evaluation judgement. We define a relation $\overset{n}{\Rightarrow}$, the *many-step evaluation judgement*, whose rules are given in Figure 11. The statement $M \parallel s\,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s'\,;(\delta_i)_{i<n}$ expresses that repeated applications of single-step reduction starting from state s leads to state s', with step $i \in [0,n)$ consuming γ_i and producing δ_i.

Rule EOmega deserves special attention. Being able to run a program for ω steps means being able to run it for m steps, for every finite m. The infinite family of increments returned as output is defined elementwise by all the finite runs of the program. However, the output

state of all the finite runs is thrown away, and an infinite run always has output state **done**. Intuitively speaking, the final state of a program after ω steps is irrelevant since it can never do any further step. Since the rules in Figure 8 do not contain any rule that takes **done** as input state, ESTEP can never be applied, making **done** final — which is the intended behavior.

Any two many-step reductions originating from the same term, inputs, and initial state yield the same outputs for both state and produced increments. In other words, the automaton described by M is deterministic.

Theorem 3.2 (Determinism). *If* $M \Vert s; (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s'_1 ; (\delta^1_i)_{i<n}$ *and* $M \parallel s ; (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s'_2 ; (\delta^2_i)_{i<n}$, *then* $s'_1 = s'_2$ *and* $\delta^1_i = \delta^2_i$ *for all* i.

Since the many-step evaluation judgement depends on the single-step evaluation judgement and vice-versa due to rules ESTEP and EBY respectively, the proof has to proceed by mutual induction on the judgements using a lemma that proves determinism in the single-step case.

Lemma 3.1 (Determinism, single-step). *If we have* $M \parallel s ; \gamma \Longrightarrow s'_1 ; \delta_1$ *and* $M \parallel s ; \gamma \Longrightarrow s'_2 ; \delta_2$ *then* $s'_1 = s'_2$ *and* $\delta_1 = \delta_2$.

$$\boxed{M \parallel s ; (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s' ; (\delta_i)_{i<n}}$$

EZERO

$$\frac{}{M \parallel s ; \varepsilon \overset{0}{\Longrightarrow} s ; \varepsilon}$$

ESTEP

$$\frac{M \parallel s ; (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s' ; (\delta_i)_{i<n} \qquad M \parallel s' ; \gamma_n \Longrightarrow s'' ; \delta_n}{M \parallel s ; (\gamma_i)_{i<n+1} \overset{n+1}{\Longrightarrow} s'' ; (\delta_i)_{i<n+1}}$$

EOMEGA

$$\frac{\forall m \leq \omega, \quad M \parallel s ; (\gamma_i)_{i<m} \overset{m}{\Longrightarrow} s'_m ; (\delta_i)_{i<m}}{M \parallel s ; (\gamma_i)_{i<\omega} \overset{\omega}{\Longrightarrow} \mathbf{done} ; (\delta_i)_{i<\omega}}$$

Figure 11 Many-step evaluation judgement

3.3 Metatheory

We now prove metatheoretical properties of the type system in relation with the many-step evaluation judgment. These theorems prove that our incremental semantics is as well-behaved as the prefix semantics: it is a type-respecting total function on well-typed terms. Since the type system of increments applies to families of increments indexed by a downward-closed subset of $\omega + 1$, the theorems only apply with **stop** as the initial state.

Many-step reduction is type-safe. Running a closed term of type A with initial state **stop** for n steps produces a family of increments $(\delta_i)_{i<n}$ of type A at n. This result, generalized to open terms, is proved by induction over typing derivations.

Theorem 3.3 (Type safety). *Let* $\Gamma \vdash M : A \mid S$, *and* $n \leqslant \omega$. *For all* $(\gamma_i)_{i<n} : \Gamma \mathbin{@} n$ *and* $(\delta_i)_{i<n}$ *such that* $M \parallel \mathbf{stop} ; (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s ; (\delta_i)_{i<n}$, *we have* $(\delta_i)_{i<n} : A \mathbin{@} n$ *and* $s : S \mathfrak{G} n$.

Many-step reduction is a total function. For any well-typed closed term M, there exists a unique family of increments $(\delta_i)_{i<n}$ such that running M for n steps produces $(\delta_i)_{i<n}$. This is due to Theorem A.2 and to the theorem below, proved again by induction over typing derivations.

Theorem 3.4 (Totality). *Let* $\Gamma \vdash M : A \mid S$, *and* $n \leqslant \omega$. *For all* $(\gamma_i)_{i<n} : \Gamma \mathbin{@} n$, *there exists* s *and* $(\delta_i)_{i \leq n}$ *such that* $M \parallel \mathbf{stop} ; (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s ; (\delta_i)_{i<n}$.

$$
\begin{array}{rll}
\varepsilon \models_0^A & \mathbf{stop} & \\
(\delta_i)_{i<\omega} \models_\omega^A & \mathbf{box}(M)\{E\} & \overset{\triangle}{\iff} \forall n < \omega.\ (\delta_i)_{i<n} \models_n^A \lfloor \mathbf{box}(M)\{E\} \rfloor_n \\
(c) \oplus (\mathbf{nil})_{i<n-1} \models_n^\nu & c & \\
\left(\triangleright (\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n} \models_n^{p \Rightarrow A} & \mathbf{w}\,(p, V) & \overset{\triangle}{\iff} (\delta_i)_{i<p(n)} \models_{p(n)}^A V \\
\left(\delta_i :: \triangleright (\delta'_j)_{i-1\leqslant j<i}\right)_{i<n} \models_n^{\mathsf{Str}\,A} & V :: V' & \overset{\triangle}{\iff} (\delta_i)_{i<n} \models_n^A V \text{ and } (\delta'_i)_{i<n-1} \models_{n-1}^{\mathsf{Str}\,A} V' \\
\left((x.M)\{\gamma_i\}\right)_{i<n} \models_n^{A \overset{S}{\Rightarrow} B} & (x.N)\{E\} & \overset{\triangle}{\iff} \forall m \leqslant n, \forall (\delta_i)_{i<m} \models_m^A V, \exists s, (\delta'_i)_{i<m}, V', \\
 & & \begin{cases} M \parallel \mathbf{stop}\,;\, \left(\gamma_i[\delta_i/x]\right)_{i<m} \overset{m}{\Longrightarrow} s\,;\, (\delta'_i)_{i<m} \\ N\,;\, \lfloor E \rfloor_m [V/x] \Downarrow_m V' \\ (\delta'_i)_{i<m} \models_m^B V' \end{cases}
\end{array}
$$

Figure 12 Logical relation for values

3.4 Correctness

While the many-step evaluation judgments define a total function, it remains to be proved that it is correct, in the sense that it computes the same results as the prefix semantics. In particular, we wish to prove the following result.

Theorem 3.5. *If* $\vdash M : \mathsf{Nat} \mid S$ *and* $M\,;\varnothing \Downarrow_1 c$ *then there is* s *such that* $M \parallel \mathbf{stop}\,;\emptyset \Longrightarrow s\,;c$.

This theorem is more general than it looks. In the presence of by in the language, the term M may contain subterms which have to run for more than one step. In particular, the result c could be the ith element of some stream of numbers, for i fixed but arbitrary. For this reason, to prove this theorem, we must generalize it to many-step reduction, in addition to open terms of arbitrary types. We do so by way of the logical relation defined in Figure 12.

The logical relation specifies how families of increments assemble into prefix values. Given a family of increments $(\delta_i)_{i<n} : A \mathbin{@} n$ and a prefix value $V : A \mathbin{@} n$, we say that $(\delta_i)_i$ is a *coherent slicing* of V when $\Delta \models_n^A V$ holds. This relation is defined by well-founded induction on the lexicographical product $<_{\mathsf{ty}} \times_{\mathsf{lex}} <_{\omega+1}$ between the strict subtree ordering on types $<_{\mathsf{ty}}$ and the canonical strict ordering $<_{\omega+1}$ on $\omega + 1$.

The case of scalars, pairs, sums, and streams are unremarkable. Thunks representing infinite prefixes are coherently sliced into infinite families of prefixes in such a way that truncation at a finite length always produces coherent slicings. A warped increment family is a coherent slicing of a warped value if their unfoldings are coherent slicings of each other. Finally, functions follow the usual pattern of logical relations, adapted to our setting: a (prefix) closure is coherently sliced by a family of incremental closures when, applied to coherent arguments, they return coherent results.

The logical relation over values extends to incremental and prefix environments.

Definition 3.1 (Logical relation for environments)**.** A family $(\gamma_i)_i : \Gamma \mathbin{@} n$ is a *coherent slicing* of $E : \Gamma \mathbin{@} n$, denoted $(\gamma_i)_i \models_n^\Gamma E$, when, for all $x \in \mathrm{dom}(\Gamma)$, we have $(\gamma_i(x))_i \models_n^{\Gamma(x)} E(x)$.

We extend the logical relation to terms. Intuitively, two closed terms M and N are logically related when the increments produced by n reactions of M coherently slice the prefix of length n computed from N. The definition below generalizes this to open terms.

Definition 3.2 (Logical relation for terms)**.** The terms M and N are *logically related* at Γ and A, denoted $M \models^{\Gamma\vdash A} N$, when, for all n, E, $(\gamma_i)_{i<n}$, V, $(\delta_i)_{i<n}$ and s such that $(\gamma_i)_{i<n} \models_n^\Gamma E$, if $N\,;E \Downarrow_n V$ and $M \parallel \mathbf{stop}\,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s\,;(\delta_i)_{i<n}$, then $(\delta_i)_{i<n} \models_n^A V$.

Adequacy. The adequacy lemma for our logical relation immediately implies Theorem 3.5 by the definition of AScalar and the fact that the single-step reduction is a special case of the many-step one.

Lemma 3.2 (Adequacy)**.** *If* $\Gamma \vdash M : A \mid S$*, then* $M \models^{\Gamma\vdash A} M$.

The proof is done by induction over the typing hypothesis. Inversion lemmas are needed to invert whole chains of many-step reductions, as it is defined in term of single-step reductions. Such lemmas are not needed for inverting the prefix reduction. The induction hypothesis is then to be applied on subterms, which in turn proves that they produce coherent slicings. Application of the rules of Figure 12 yields the desired results.

To prove the adequacy lemma, we need a result connecting truncation in the original prefix semantics with our incremental semantics in the following sense.

Lemma 3.3 (Coherence of truncation)**.** *If* $(\delta_i)_{i<n} \models_n^A V$ *and* $m \leqslant n$*, then* $(\delta_i)_{i<m} \models_m^A \lfloor V \rfloor_m$.

The key case is the one of Rec. It is dealt with by an inner induction on the number of steps, using Lemma A.9 to connect the intermediate families of increments with the values computed by the prefix semantics.

4 Perspectives

4.1 Limitations

Increments storage in states. Because our increment typing judgement only classifies whole families of increments, states have to store entire families of increments as well to be well-typed. As a consequence, a naive implementation of our semantics of delays and fixed points would keep all past increments in memory, resulting in unbounded space usage. This is very unsatisfactory since the elimination of such spurious "space leaks" is a core concern in reactive programming. Fortunately, inspection of the single-step judgement shows that the rules that use state to store intermediate computations only rely on a subset of the increments stored in this state. For example, the semantics of fix actually uses only the last increment from its state, and thus only requires a buffer of size one.

Infinitary syntax. As mentioned in Section 3, in the original semantics, prefixes are always finite since prefixes of type $A \mathbin{@} \omega$ are thunks. In contrast, families of increments can be actually infinite. This choice makes for simple semantics but cannot be used in a non-lazy implementation. We believe that using thunks in the incremental semantics would be technically heavier but would not raise any conceptual issue.

Structural coercions of the warping modality. The original presentation of λ^* includes primitive constructs to mediate between isomorphic types such as $p \Rrightarrow (A \times B)$ and $(p \Rrightarrow A) \times (p \Rrightarrow B)$. We have omitted their unremarkable treatment from this paper for lack of space.

Recursive states. Our state types do not include recursive types such as streams. This restriction makes it impossible to capture most higher-order recursive functions, such as the general map function over streams. This function should have type $(A \overset{S}{\Rightarrow} B) \overset{\mathbb{1}}{\Rightarrow} \mathsf{Str}\,A \overset{\mathsf{Str}\,S}{\Longrightarrow} \mathsf{Str}\,B$, reflecting how map f xs creates a new copy of the state of f for each element of xs.

4.2 Related work

The implementation of reactive programs is a wide area that has been explored in distinct directions by distinct lines of research. We briefly discuss two of them.

Synchronous programming. Synchronous languages restrict stream elements to be of scalar types, eschewing types such as streams of streams. This restriction allows for efficient if specialized implementation techniques. Our incremental semantics is directly inspired from their "single-loop" state-passing transform [17, 6, 3], but handles streams of arbitrary element type, such as streams of streams, as well as higher-order functions.

We owe a particularly large debt to Lucid Synchrone [16], whose compiler introduced the idea of type-checking program states in a higher-order reactive language more than twenty years ago. In contrast with the present work, in Lucid Synchrone state types are almost[4] invisible at the source level and only appear in the generated OCaml code, relying on the OCaml compiler for this part of type-checking. Our choice to include state types in the source type system simplifies metatheoretical proofs and would facilitate an implementation of separate compilation. However, the absence of recursive state types forces us to reject many higher-order programs (see Section 4.2), some of which are accepted by Lucid Synchrone.

Functional reactive programming. Several works in functional reactive programming use techniques similar to that of synchronous programming. For example, causal commutative arrows [13] can be compiled to simple state-passing code but are purely first-order.

Several recent proposals such as that of Krishnaswami [11] or Bahr et al. [2] exploit a modal type discipline to reject certain recursion patterns and rule out implicit memory leaks in an expressive higher-order language. In their operational semantics, program state is untyped and dynamically allocated in a global store, while in ours it is typed and thus controlled by term structure. An in-depth comparison of the two styles remains for future work.

4.3 Conclusion

We have presented a new operational semantics for λ^*, a λ-calculus for productive reactive programming proposed in previous work. This semantics, in contrast with the existing one, is incremental. It can be seen as an adaptation of classic synchronous-language compilation techniques to a setting featuring rich temporal types such as streams of streams or streams of stream functions. Our incremental semantics enjoys the same desirable metatheoretical properties: type safety, totality, and determinism. Using a logical relation to specify how prefixes can be sliced into families of incremental values, we have proved that it is fully consistent with respect to the existing semantics.

In future work, we plan to refine the space usage of the new semantics, extend it beyond streams to arbitrary guarded recursive types, and capture its incremental character in a denotational model by adapting the existing interpretation of λ^* in the topos of trees. We would also like to adapt it into an implementation.

References

[1] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A Very Modal Model of a Modern, Major, General Type System. In *Principles of Programming Languages*

[4] Lucid Synchrone distinguishes between functions of unit state type and functions of non-unit state types at the source level, grouping all of the former functions into a single type.

(POPL'07). ACM, 2007.

[2] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The Clocks Are Ticking: No More Delays! Reduction Semantics for Type Theory with Guarded Recursion. In *Logic in Computer Science (LICS'17)*. Springer, 2017.

[3] Dariusz Biernacki, Jean-Louis Colaço, Gregoire Hamon, and Marc Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Languages, Compilers and Tools for Embedded Systems (LCTES'08)*, 2008.

[4] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012.

[5] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Principles of Programming Languages (POPL'87)*, 1987.

[6] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *International Conference on Functional Programming (ICFP'96)*. ACM, 1996.

[7] Paul Caspi and Marc Pouzet. A co-iterative characterization of synchronous stream functions. *Electronic Notes in Theoretical Computer Science*, 11:1–21, 1998.

[8] Conal Elliott and Paul Hudak. Functional Reactive Animation. In *International Conference on Functional Programming (ICFP'97)*. ACM, 1997.

[9] Adrien Guatto. A Generalized Modality for Recursion. In *Logic in Computer Science (LICS'18)*, 2018.

[10] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing Congress (IFIP'74)*. IFIP, 1974.

[11] Neelakantan R Krishnaswami. Higher-Order Functional Reactive Programming without Spacetime Leaks. In *International Conference on Functional Programming (ICFP'13)*. ACM, 2013.

[12] Neelakantan R. Krishnaswami and Nick Benton. Ultrametric Semantics of Reactive Programs. In *Logic in Computer Science (LICS'11)*. IEEE, 2011.

[13] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows. *Journal of Functional Programming*, 2011.

[14] John Lucassen and David Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL'88)*. ACM, 1988.

[15] Hiroshi Nakano. A Modality for Recursion. In *Logic in Computer Science (LICS'00)*. IEEE, 2000.

[16] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006.

[17] Pascal Raymond. *Compilation efficace d'un langage déclaratif synchrone : Le générateur de code Lustre-V3*. PhD thesis, Institut National Polytechnique de Grenoble, 1991.

A Interesting proofs

A.1 Inversion and construction lemmas

A.1.1 Inversion lemmas

Lemma A.1 (Inversion lemma for by)**.** *Let $0 < n \leqslant \omega$. Suppose that:*

$$M \text{ by } p \parallel \mathbf{stop} \,;\, \left(\triangleright(\gamma_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n} \xRightarrow{n} s \,;\, (\delta_i)_{i<n}$$

Then, we have:

$$\exists s'.\ \exists(\delta'_i)_{i<p(n)}.\ \begin{cases} M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<p(n)} \xRightarrow{p(n)} s' \,;\, (\delta'_i)_{i<p(n)} \\ \delta_i = \triangleright(\delta'_j)_{p(i)<j\leqslant p(i+1)} \text{ for all } i < n \\ s = \begin{cases} \mathbf{w}\,(p, s') & \text{if } n < \omega \\ \mathbf{done} & \text{otherwise} \end{cases} \end{cases}$$

Proof. By induction on n.

- Case $n = 1$. By hypothesis and inversion of EStep we have:

  $$M \text{ by } p \parallel \mathbf{stop} \,;\, \triangleright(\gamma_i)_{i<p(1)} \Longrightarrow s \,;\, \delta_0$$

  Then, by inversion of EByI, we obtain what we wanted:

  $$M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<p(1)} \xRightarrow{p(1)} s' \,;\, (\delta'_i)_{i<p(1)} \qquad \delta_0 = \triangleright(\delta'_i)_{i<p(1)} \qquad s = \mathbf{w}\,(p, s')$$

- Case $1 < n < \omega$. By hypothesis and inversion of EStep we have:

  $$M \text{ by } p \parallel \mathbf{stop} \,;\, \left(\triangleright(\gamma_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n-1} \xRightarrow{n-1} s_I \,;\, (\delta_i)_{i<n-1} \tag{a}$$

  $$M \text{ by } p \parallel s_I \,;\, \triangleright(\gamma_j)_{p(n-1)\leqslant j<p(n)} \Longrightarrow s \,;\, \delta_n \tag{b}$$

  Applying the induction hypothesis to (a) yields:

  $$M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<p(n-1)} \xRightarrow{p(n-1)} s'_I \,;\, (\delta'_i)_{i<p(n-1)} \tag{c}$$

  $$\delta_i = \triangleright(\delta'_j)_{p(i)<j\leqslant p(i+1)} \quad \forall i < n-1 \tag{d}$$

  $$s_I = \mathbf{w}\,(p, s'_I) \tag{e}$$

  Now, we distinguish two cases.

  - $p(n-1) = \omega$. In this case, $s'_I = \mathbf{done}$, $p(n) = \omega$ and $\partial p(n) = 0$. We have by EZero that :

    $$M \parallel s'_I \,;\, \triangleright\varepsilon \xRightarrow{\partial p(n)} s'_I \,;\, \triangleright\varepsilon$$

    Thus, $\delta_{n-1} = \triangleright\varepsilon$. Substituting $p(n-1) = p(n) = \omega$ in the above equations yields what we wanted:

    $$M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<p(n)} \xRightarrow{p(n)} s' \,;\, (\delta'_i)_{i<p(n)}$$

    $$\delta_i = \triangleright(\delta'_j)_{p(i)<j\leqslant p(i+1)} \quad \forall i < n$$

    $$s = \mathbf{w}\,(p, s') = \mathbf{w}\,(p, \mathbf{done})$$

– $p(n-1) < \omega$. In this case, $s_I = \mathbf{w}\,(p, s'_I)$. By substituting (e) in (b) and then inverting EBy, we obtain:

$$M \parallel s'_I \,;(\gamma_i)_{p(n-1)\leqslant i<p(n)} \xRightarrow{\partial p(n)} s' \,;(\delta'_i)_{p(n-1)\leqslant i<p(n)} \tag{f}$$

$$\delta_{n-1} = \triangleright(\delta'_j)_{p(n-1)<j\leqslant p(n)} \tag{g}$$

$$s = \begin{cases} \mathbf{w}\,(p, s') & \text{if } \partial p(n) < \omega \\ \mathbf{done} & \text{otherwise} \end{cases} \tag{h}$$

Notice that if $\partial p(n) = \omega$, then $p(n) = \omega$

Since $p(n) = p(n-1) + \partial p(n)$, combining (c) and (f) yields what we wanted:

$$M \parallel \mathbf{stop} \,;(\gamma_i)_{i<p(n)} \xRightarrow{p(n)} s' \,;(\delta'_i)_{i<p(n)}$$

$$\delta_i = \triangleright(\delta'_j)_{p(i)<j\leqslant p(i+1)} \quad \forall i < n$$

$$s = \begin{cases} \mathbf{w}\,(p, s') & \text{if } p(n) < \omega \\ \mathbf{done} & \text{otherwise} \end{cases}$$

- Case $n = \omega$. By hypothesis and inversion of EOmega we have:

$$\forall m < \omega. \quad M \text{ by } p \parallel \mathbf{stop} \,;\left(\triangleright(\gamma_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<m} \xRightarrow{m} s_m \,;(\delta_i)_{i<m}$$

The induction hypothesis applies, and gives :

$$\forall m < \omega. \quad \begin{cases} M \parallel \mathbf{stop} \,;(\gamma_i)_{i<p(m)} \xRightarrow{p(m)} s'_{p(m)} \,;(\delta'_i)_{i<p(m)} \\ \delta_i = \triangleright(\delta'_j)_{p(i)<j\leqslant p(i+1)} \quad \forall i < m \end{cases} \tag{i}$$

By inversion of EOmega on the hypothesis we have that $s = \mathbf{done}$. By (i), we obtain that $\forall i < \omega.\ \delta_i = \triangleright(\delta'_j)_{p(i)<j\leqslant p(i+1)}$.

We conclude in different ways depending on the finiteness of $p(\omega)$.

– $p(\omega) = \omega$. Let $k < \omega$. Since p is monotonic due to being a time warp, and $p(\omega) = \omega$, there exists a M such that $k \leqslant p(M)$. By $p(M) - k$ inversions of EStep (or EOmega if $p(M) = \omega$), we obtain that :

$$\exists s''_k. \quad M \parallel \mathbf{stop} \,;(\gamma_i)_{i<k} \xRightarrow{k} s''_k \,;(\delta'_i)_{i<k}$$

Thus, we have that :

$$\forall k < \omega. \quad M \parallel \mathbf{stop} \,;(\gamma_i)_{i<k} \xRightarrow{k} s''_k \,;(\delta'_i)_{i<k}$$

Immediately applying EOmega and substituting $p(\omega) = \omega$, we obtain :

$$M \parallel \mathbf{stop} \,;(\gamma_i)_{i<p(\omega)} \xRightarrow{p(\omega)} \mathbf{done} \,;(\delta'_i)_{i<p(\omega)}$$

Which concludes our proof.

– $p(\omega) < \omega$. In this case, p is ultimately constant: there is some finite M such that $p(M) = p(\omega)$. Thus, by instantiating (i) with $m := M$, we obtain :

$$M \parallel \mathbf{stop} \,;(\gamma_i)_{i<p(M)} \xRightarrow{p(M)} s'_M \,;(\delta'_i)_{i<p(M)}$$

By substituting $p(M)$ for $p(\omega)$ in the above, we prove that:

$$M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<p(\omega)} \xRightarrow{p(\omega)} s'_M \,;\, (\delta'_i)_{i<p(\omega)}$$

Which concludes our proof.

□

Lemma A.2 (Inversion lemma for rec)**.** *Let $0 < n \leqslant \omega$. Suppose that:*

$$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n} \xRightarrow{n} s \,;\, (\delta_i)_{i<n}$$

Then, we have:

$$\exists s'.\ \begin{cases} M \parallel \mathbf{stop} \,;\, \left(\gamma_0\left[\triangleright\varepsilon/_x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/_x\right]\right)_{1\leqslant i<n} \xRightarrow{n} s' \,;\, (\delta_i)_{i<n} \\ s = \begin{cases} \left(s', \mathbf{inc}\left((\delta_i)_{i<n}\right)\right) & \text{if } n < \omega \\ \mathbf{done} & \text{otherwise} \end{cases} \end{cases}$$

Proof. By induction on n.

- Case $n = 1$. By hypothesis and inversion of EStep we have:

  $$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, \gamma_0 \Longrightarrow s \,;\, \delta_0$$

  Then, by inversion of EFixI, we obtain:

  $$M \parallel \mathbf{stop} \,;\, \left(\gamma_0\left[\triangleright\varepsilon/_x\right]\right) \Longrightarrow s' \,;\, \delta_0 \qquad\qquad s = \left(s', \mathbf{inc}\,(\delta)\right)$$

  Applying EStep yields what we wanted.

- Case $1 < n < \omega$. By hypothesis and inversion of EStep we have:

  $$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n-1} \xRightarrow{n-1} s_I \,;\, (\delta_i)_{i<n-1} \tag{a}$$
  $$\mathsf{rec}\, x^A.\, M \parallel s_I \,;\, \gamma_{n-1} \Longrightarrow s \,;\, \delta_{n-1} \tag{b}$$

  Applying the induction hypothesis to (a) yields :

  $$M \parallel \mathbf{stop} \,;\, \left(\gamma_0\left[\triangleright\varepsilon/_x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/_x\right]\right)_{1\leqslant i<n-1} \xRightarrow{n-1} s'_I \,;\, (\delta_i)_{i<n-1} \tag{c}$$
  $$s_I = \left(s'_I, \mathbf{inc}\left((\delta_i)_{i<n-1}\right)\right) \tag{d}$$

  By substituting (d) in (b) and then inverting ERec, we obtain:

  $$M \parallel s'_I \,;\, \left(\gamma_{n-1}\left[\triangleright(\delta_{n-2})/_x\right]\right) \Longrightarrow s' \,;\, \delta_{n-1} \tag{e}$$
  $$s = \left(s', \mathbf{inc}\left((\delta_i)_{i<n}\right)\right) \tag{f}$$

  Together with (f), applying EStep to (c) and (e) yields what we wanted:

  $$M \parallel \mathbf{stop} \,;\, \left(\gamma_0\left[\triangleright\varepsilon/_x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/_x\right]\right)_{1\leqslant i<n} \xRightarrow{n} s' \,;\, (\delta_i)_{i<n}$$
  $$s = \left(s', \mathbf{inc}\left((\delta_i)_{i<n}\right)\right)$$

- Case $n = \omega$. By hypothesis and inversion of EOmega we have $s = \mathbf{done}$ and:

  $$\forall m < \omega.\ \ \mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<m} \xRightarrow{m} s'_m \,;\, (\delta_i)_{i<m}$$

The induction hypothesis applies, and gives :

$$\forall m < \omega.\ M \parallel \mathbf{stop}\,;\left(\gamma_0\left[\triangleright\varepsilon/_x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/_x\right]\right)_{1\leqslant i<m} \overset{m}{\Longrightarrow} s'_m\,;(\delta_i)_{i<m}$$

Applying EOmega to the above equation gives what we wanted.

□

Lemma A.3 (Inversion lemma for delay)**.** *Let $0 < n \leqslant \omega$. Suppose that:*

$$\mathsf{delay}^{q\leqslant p}(M) \parallel \mathbf{stop}\,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s\,;(\delta_i)_{i<n}$$

Then, we have:

$$\exists s'.\ \exists(\delta'_i)_{i<p(n)}.\ \begin{cases} M \parallel \mathbf{stop}\,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s'\,;\left(\triangleright(\delta'_i)_{p(i)<j\leqslant p(i+1)}\right)_{i<n} \\ \delta_i = \triangleright(\delta'_j)_{q(i)<j\leqslant q(i+1)} \text{ for all } i<n \\ s = \begin{cases} \left(s', \mathbf{inc}\left(\left(\triangleright(\delta'_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}\right)\right) & \text{if } n<\omega \\ \mathbf{done} & \text{otherwise} \end{cases} \end{cases}$$

Proof. By induction on n.

- Case $n = 1$. By hypothesis and inversion of EStep we have:

  $$\mathsf{delay}^{q\leqslant p}(M) \parallel \mathbf{stop}\,;\gamma_0 \Longrightarrow s\,;\delta_0$$

  Then, by inversion of EDelayI and immediate application of EStep, we obtain what we wanted:

  $$M \parallel \mathbf{stop}\,;(\gamma_0) \overset{1}{\Longrightarrow} s'\,;\left(\triangleright(\delta'_i)_{i<q(1)}\right) \qquad \delta_0 = \triangleright(\delta'_i)_{i<q(1)} \qquad s = \left(s', \mathbf{inc}\left(\triangleright(\delta'_i)_{i<p(1)}\right)\right)$$

- Case $1 < n < \omega$. By hypothesis and inversion of EStep we have:

  $$\mathsf{delay}^{q\leqslant p}(M) \parallel \mathbf{stop}\,;(\gamma_i)_{i<n-1} \overset{n-1}{\Longrightarrow} s_I\,;(\delta_i)_{i<n-1} \tag{a}$$
  $$\mathsf{delay}^{q\leqslant p}(M) \parallel s_I\,;\gamma_n \Longrightarrow s\,;\delta_n \tag{b}$$

  Applying the induction hypothesis to (a) yields:

  $$M \parallel \mathbf{stop}\,;(\gamma_i)_{i<n-1} \overset{n-1}{\Longrightarrow} s'_I\,;\left(\triangleright(\delta'_i)_{p(i)<j\leqslant p(i+1)}\right)_{i<n-1} \tag{c}$$
  $$\delta_i = \triangleright(\delta'_j)_{q(i)<j\leqslant q(i+1)} \text{ for all } i<n-1 \tag{d}$$
  $$s_I = \left(s'_I, \mathbf{inc}\left(\left(\triangleright(\delta'_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n-1}\right)\right) \tag{e}$$

  Now, by substitution of (e) in (b) and immediate inversion of EDelay, we get:

  $$M \parallel s'_I\,;\gamma_n \Longrightarrow s'\,;\triangleright(\delta'_i)_{p(n-1)<j\leqslant p(n)} \tag{f}$$
  $$\delta_n = \triangleright(\delta'_j)_{q(n-1)<j\leqslant q(1)} \tag{g}$$
  $$s = \left(s', \mathbf{inc}\left(\left(\triangleright(\delta'_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}\right)\right) \tag{h}$$

Applying EStep on (c) and (f), merging (d) with (g), and (h) yield what we wanted:

$$M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s' \,;\, \left(\triangleright(\delta'_i)_{p(i)<j\leqslant p(i+1)}\right)_{i<n}$$

$$\delta_i = \triangleright(\delta'_j)_{q(i)<j\leqslant q(i+1)} \text{ for all } i < n$$

$$s = \left(s', \mathbf{inc}\left(\left(\triangleright(\delta'_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}\right)\right)$$

- Case $n = \omega$. By hypothesis and inversion of EOmega we have:

$$\forall m < \omega. \quad \mathsf{delay}^{q\leqslant p}(M) \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<m} \overset{m}{\Longrightarrow} s_m \,;\, (\delta_i)_{i<m}$$

The induction hypothesis applies, and gives :

$$\forall m < \omega. \quad \begin{cases} M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<m} \overset{m}{\Longrightarrow} s'_m \,;\, \left(\triangleright(\delta'_i)_{p(i)<j\leqslant p(i+1)}\right)_{i<m} \\ \delta_i = \triangleright(\delta'_j)_{q(i)<j\leqslant q(i+1)} \text{ for all } i < m \end{cases} \tag{i}$$

By inversion of EOmega on the hypothesis we have that $s = \mathbf{done}$. By (i), we obtain that $\forall i < \omega.\ \delta_i = \triangleright(\delta'_j)_{q(i)<j\leqslant q(i+1)}$.

Applying EOmega on (i) gives:

$$M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<\omega} \overset{\omega}{\Longrightarrow} \mathbf{done} \,;\, \left(\triangleright(\delta'_i)_{p(i)<j\leqslant p(i+1)}\right)_{i<\omega}$$

Which concludes our proof.

□

A.1.2 Construction lemmas

Lemma A.4 (Construction lemma for variables)**.** *Let $0 < n \leqslant \omega$. Let $(\gamma_i)_{i<n}$, such that:*

$$\forall i < n.\ x \in \mathrm{dom}(\gamma_i(x))$$

Then, we have:

$$x \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s \,;\, (\gamma_i(x))_{i<n} \text{ – with } s = \begin{cases} \mathbf{unit} \text{ if } n < \omega \\ \mathbf{done} \text{ elsewise} \end{cases}$$

Proof. By induction on n.

- Case $n = 1$. By hypothesis, since $x \in \mathrm{dom}(\gamma_0)$, then EVarI applies and yields:

$$x \parallel \mathbf{stop} \,;\, \gamma_0 \Longrightarrow \mathbf{unit} \,;\, \gamma_0(x)$$

Immediate application of EStep yields what we wanted.

- Case $1 < n < \omega$. The induction hypothesis gives:

$$x \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n-1} \overset{n-1}{\Longrightarrow} \mathbf{unit} \,;\, (\gamma_i(x))_{i<n-1}$$

By hypothesis, since $x \in \mathrm{dom}(\gamma_{n-1})$, then EVar applies and yields:

$$x \parallel \mathbf{unit} \,;\, \gamma_{n-1} \Longrightarrow \mathbf{unit} \,;\, \gamma_{n-1}(x)$$

Immediate application of EStep yields what we wanted.

- Case $n = \omega$. Since $x \in \mathrm{dom}(\gamma_m)$ for all $m < \omega$, the induction hypothesis applies and gives:
$$\forall m < \omega. \quad x \parallel \mathbf{stop} ; (\gamma_i)_{i<m} \overset{m}{\Longrightarrow} \mathbf{unit} ; (\gamma_i(x))_{i<m}$$
Immediate application of EOmega yields what we wanted.

□

Lemma A.5 (Construction lemma for by)**.** *Let $0 < n \leqslant \omega$. Suppose that:*
$$M \parallel \mathbf{stop} ; (\gamma_i)_{i<p(n)} \overset{p(n)}{\Longrightarrow} s' ; (\delta_i)_{i<p(n)}$$
Then, we have:
$$M \text{ by } p \parallel \mathbf{stop} ; \left(\triangleright(\gamma_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n} \overset{n}{\Longrightarrow} s ; \left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}$$
With s such that:
$$s = \begin{cases} \mathbf{w}\,(p, s') & \text{if } n < \omega \\ \mathbf{done} & \text{elsewise} \end{cases}$$

Proof. By induction on n.

- Case $n = 1$. By hypothesis we have:
$$M \parallel \mathbf{stop} ; (\gamma_i)_{i<p(1)} \overset{p(1)}{\Longrightarrow} s' ; (\delta_i)_{i<p(1)}$$
Notice that since p is a time warp, $p(0) = 0$. We apply EByI:
$$M \text{ by } p \parallel \mathbf{stop} ; \triangleright(\delta_j)_{p(0)\leqslant j<p(1)} \Longrightarrow \mathbf{w}\,(p, s') ; \triangleright(\gamma_j)_{p(0)\leqslant j<p(1)}$$
Applying EStep yields what we wanted.

- Case $1 < n < \omega$. By hypothesis and inversion of EStep we have:
$$M \parallel \mathbf{stop} ; (\gamma_i)_{i<p(n-1)} \overset{p(n-1)}{\Longrightarrow} s'_I ; (\delta_i)_{i<p(n-1)} \tag{a}$$
$$M \parallel s'_I ; (\gamma_i)_{p(n-1)\leqslant i<p(n)} \overset{\partial p(n)}{\Longrightarrow} s' ; (\delta_i)_{p(n-1)\leqslant i<p(n)} \tag{b}$$
Applying the induction hypothesis to (a), and EBy to (b) yields :
$$M \text{ by } p \parallel \mathbf{stop} ; \left(\triangleright(\gamma_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n-1} \overset{n-1}{\Longrightarrow} \mathbf{w}\,(p, s'_I) ; \left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n-1}$$
$$M \text{ by } p \parallel \mathbf{w}\,(p, s'_I) ; \triangleright(\gamma_j)_{p(n-1)\leqslant j<p(n)} \Longrightarrow \mathbf{w}\,(p, s') ; \triangleright(\delta_j)_{p(n-1)\leqslant j<p(n)}$$
Immediate application of EStep yields what we wanted:
$$M \text{ by } p \parallel \mathbf{stop} ; \left(\triangleright(\gamma_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n} \overset{n}{\Longrightarrow} \mathbf{w}\,(p, s') ; \left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}$$

- Case $n = \omega$. By hypothesis we have:
$$M \parallel \mathbf{stop} ; (\gamma_i)_{i<p(\omega)} \overset{p(\omega)}{\Longrightarrow} s' ; (\delta_i)_{i<p(\omega)} \tag{c}$$
We distinguish to cases, depending on the value of $p(\omega)$.

– Case $p(\omega) < \omega$. p is ultimately constant – this means that there is a finite N such that $p(N) = p(\omega)$. Thus, (c) rewrites to:

$$M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<p(N)} \xRightarrow{p(N)} s' \,;\, (\delta_i)_{i<p(N)}$$

By application of the induction hypothesis, we obtain:

$$M \text{ by } p \parallel \mathbf{stop} \,;\, \left(\triangleright(\gamma_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<N} \xRightarrow{N} \mathbf{w}\,(p,s') \,;\, \left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<N} \quad \text{(d)}$$

By EZero, we always have $M \parallel s' \,;\, \varepsilon \xRightarrow{0} s' \,;\, \varepsilon$. Thus, by EBy:

$$M \text{ by } p \parallel \mathbf{w}\,(p,s') \,;\, \triangleright\varepsilon \Longrightarrow \mathbf{w}\,(p,s') \,;\, \triangleright\varepsilon$$

Notice that since p is ultimately constant at N, then for all $m \geqslant N$, we have:

$$(\gamma_i)_{p(m)\leqslant i<p(m+1)} = (\delta_i)_{p(m)\leqslant i<p(m+1)} = \varepsilon$$

Thus, EStep applies any number of times to (d) and yields:

$$\forall m < \omega.\ M \text{ by } p \parallel \mathbf{stop} \,;\, \left(\triangleright(\gamma_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<m} \xRightarrow{m} \mathbf{w}\,(p,s') \,;\, \left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<m}$$

Applying EOmega proves what we wanted.

– Case $p(\omega) = \omega$. Immediate inversion of EOmega in (c) yields:

$$\forall m < \omega.\ M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<m} \xRightarrow{m} s'_m \,;\, (\delta_i)_{i<m}$$

This especially gives:

$$\forall m < \omega.\ M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<p(m)} \xRightarrow{p(m)} s'_{p(m)} \,;\, (\delta_i)_{i<p(m)}$$

Applying the induction hypothesis gives:

$$\forall m < \omega.\ M \text{ by } p \parallel \mathbf{stop} \,;\, \left(\triangleright(\gamma_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<m} \xRightarrow{m} \mathbf{w}\left(p, s'_{p(m)}\right) \,;\, \left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<m}$$

Applying EOmega proves what we wanted.

□

Lemma A.6 (Construction lemma for rec)**.** *Let $0 < n \leqslant \omega$. Suppose that:*

$$M \parallel \mathbf{stop} \,;\, \left(\gamma_0\left[\triangleright\varepsilon/x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/x\right]\right)_{1\leqslant i<n} \xRightarrow{n} s' \,;\, (\delta_i)_{i<n}$$

Then, we have:

$$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n} \xRightarrow{n} s \,;\, (\delta_i)_{i<n} \text{ – with } s = \begin{cases} (s', \mathbf{inc}\,((\delta_i)_{i<n})) & \text{if } n < \omega \\ \mathbf{done} & \text{elsewise} \end{cases}$$

Proof. By induction on n.

- Case $n = 1$. By hypothesis and inversion of EStep we have:

$$M \parallel \mathbf{stop} \,;\, \gamma_0\left[\triangleright\varepsilon/x\right] \Longrightarrow s' \,;\, \delta_0$$

Then, by application of EFixI, we obtain:

$$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, \gamma_0 \Longrightarrow (s', \mathbf{inc}\,(\delta_0)) \,;\, \delta_0$$

Applying EStep yields what we wanted.

- Case $1 < n < \omega$. By hypothesis and inversion of EStep we have:

$$M \parallel \mathbf{stop} \,;\, \left(\gamma_0\left[\triangleright\varepsilon/x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/x\right]\right)_{1\leqslant i<n-1} \overset{n-1}{\Longrightarrow} s'_I \,;\, (\delta_i)_{i<n-1} \tag{a}$$

$$M \parallel s_I \,;\, \left(\gamma_{n-1}\left[\triangleright(\delta_{n-2})/x\right]\right) \Longrightarrow s' \,;\, \delta_{n-1} \tag{b}$$

  Applying the induction hypothesis to (a), and ERec to (b) yields :

$$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n-1} \overset{n-1}{\Longrightarrow} (s'_I, \mathbf{inc}\left((\delta_i)_{i<n-1}\right)) \,;\, (\delta_i)_{i<n-1}$$

$$\mathsf{rec}\, x^A.\, M \parallel (s'_I, \mathbf{inc}\left((\delta_i)_{i<n-1}\right)) \,;\, \gamma_{n-1} \Longrightarrow (s', \mathbf{inc}\left((\delta_i)_{i<n}\right)) \,;\, \delta_{n-1}$$

  Immediate application of EStep yields what we wanted:

$$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} (s', \mathbf{inc}\left((\delta_i)_{i<n}\right)) \,;\, (\delta_i)_{i<n}$$

- Case $n = \omega$. By hypothesis and inversion of EOmega we have:

$$\forall m < \omega. \quad M \parallel \mathbf{stop} \,;\, \left(\gamma_0\left[\triangleright\varepsilon/x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/x\right]\right)_{1\leqslant i<m} \overset{m}{\Longrightarrow} s'_m \,;\, (\delta_i)_{i<m}$$

  Applying the induction hypothesis gives:

$$\forall m < \omega. \quad \mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<m} \overset{m}{\Longrightarrow} \left(s'_m, \mathbf{inc}\left((\delta_i)_{i<m}\right)\right) \,;\, (\delta_i)_{i<m}$$

  Immediate application of EOmega yields what we wanted.

□

A.2 Metatheory

A.2.1 Preliminaries

We first prove some preliminary theorems.

Theorem A.1 (Type safety of increment truncation)**.** *If* $(\delta_i)_{i<n} : A \mathbin{@} n$ *and* $m \leqslant n$ *then* $(\delta_i)_{i<m} : A \mathbin{@} m$.

Proof. Let $m < n$, and $(\delta_i)_{i<n} : A \mathbin{@} n$. In the case where $m = 0$, then $(\delta_i)_{i<m} = \varepsilon$. Consequently, we have $(\delta_i)_{i<m} : A \mathbin{@} m$ by rule IStop, which proves what we wanted. In all the following, we thus suppose that $m > 0$. We proceed by induction over the typing judgement.

- *Rule* IStop. We have $n = 0$, and $(\delta_i)_{i<n} = \varepsilon$. Since $m > 0$ by the above and $m \leqslant n$, this case never happens (we have already demonstrated that the theorem always holds when $m = 0$).

- *Rule* IScalar. We have $n > 0$, $(\delta_i)_{i<n} = (c) \oplus (\mathbf{nil})_{i<n}$ and $A = \nu$. Since $m > 0$, rule IScalar gives that $(c) \oplus (\mathbf{nil})_{i<m}$. This proves what we wanted.

- *Rule* IProd. We have $n > 0$, $(\delta_i)_{i<n} = \left((\delta_i^1, \delta_i^2)\right)_{i<n}$ and $A = A_1 \times A_2$. Inversion of rule IProd gives that $(\delta_i^1)_{i<n} : A_1 \mathbin{@} n$ and $(\delta_i^2)_{i<n} : A_2 \mathbin{@} n$. The induction hypothesis gives us that $(\delta_i^1)_{i<m} : A_1 \mathbin{@} m$ and $(\delta_i^2)_{i<m} : A_2 \mathbin{@} m$. Immediate application of rule IProd allows us to conclude that $\left((\delta_i^1, \delta_i^2)\right)_{i<m} : A_1 \times A_2 \mathbin{@} m$. This proves what we wanted.

- *Rule* IInj. We have $n > 0$, $(\delta_i)_{i<n} = (\mathbf{inj}_j(\delta'_i))_{i<n}$ and $A = A_1 + A_2$. Inversion of rule IInj gives that $(\delta'_i)_{i<n} : A_j \ @ \ n$. The induction hypothesis gives us that $(\delta'_i)_{i<m} : A_j \ @ \ m$. Immediate application of rule IInj allows us to conclude that $(\mathbf{inj}_j(\delta'_i))_{i<m} : A_1 + A_2 \ @ \ m$. This proves what we wanted.

- *Rule* IWarp. We have $n > 0$, $(\delta_i)_{i<n} = \left(\triangleright(\delta'_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}$ and $A = p \Rrightarrow A'$. Inversion of rule IWarp gives that $(\delta'_i)_{i<p(n)} : A' \ @ \ p(n)$. Since $m \leqslant n \implies p(m) \leqslant p(n)$, the induction hypothesis gives us that $(\delta'_i)_{i<p(m)} : A' \ @ \ p(m)$. Immediate application of rule IWarp allows us to conclude that $\left(\triangleright(\delta'_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<m} : p \Rrightarrow A' \ @ \ m$. This proves what we wanted.

- *Rule* IClo. We have $n > 0$, $(\delta_i)_{i<n} = \left((x.M)\{\gamma_i\}\right)_{i<n}$ and $A = A \overset{S}{\Rightarrow} B$. Inversion of rule IClo gives that $\Gamma, x{:}A \vdash M : B \mid S$ and $(\gamma_i)_{i<n} : \Gamma \ @ \ n$. The induction hypothesis gives us that $(\gamma_i)_{i<m} : \Gamma \ @ \ m$. Immediate application of rule IClo allows us to conclude that $\left((x.M)\{\gamma_i\}\right)_{i<m} : A \overset{S}{\Rightarrow} B \ @ \ m$. This proves what we wanted.

- *Rule* IStream. We have $n > 0$, $(\delta_i)_{i<n} = (\delta_i^1 :: \delta_i^2)_{i<n}$ and $A = \mathsf{Str}\, A'$. Inversion of rule IStream gives that $(\delta_i^1)_{i<n} : A' \ @ \ n$ and $(\delta_i^2)_{i<n} : \mathbf{lat} \Rrightarrow \mathsf{Str}\, A' \ @ \ n$. The induction hypothesis gives us that $(\delta_i^1)_{i<m} : A' \ @ \ m$ and $(\delta_i^2)_{i<m} : \mathbf{lat} \Rrightarrow \mathsf{Str}\, A' \ @ \ m$. Immediate application of rule IStream allows us to conclude that $(\delta_i^1 :: \delta_i^2)_{i<m} : \mathsf{Str}\, A' \ @ \ m$. This proves what we wanted.

- *Rule* IBox. We have $n = \omega$. If $m = \omega$, we are done. Suppose $m < n$. Inversion of rule IBox gives that $\forall k < \omega.\ (\delta_i)_{i<k} : A \ @ \ k$. Since $m < \omega$, this especially gives us that $(\delta_i)_{i<m} : A \ @ \ m$. This proves what we wanted.

- *Rule* IEnv. We have $(\gamma_i)_{i<n} : \Gamma \ @ \ n$. Inversion IEnv gives that $\mathrm{dom}(\Gamma) = \mathrm{dom}\,(\gamma_i)$ for all $i < n$, and $(\gamma_i(x))_{i<n} : \Gamma(x) \ @ \ n$ for all $x \in \mathrm{dom}(\Gamma)$. Since $m \leqslant n$, we also have $\mathrm{dom}(\Gamma) = \mathrm{dom}\,(\gamma_i)$ for all $i < m$. Furtherore, the induction hypothesis applies, and gives $(\gamma_i(x))_{i<m} : \Gamma(x) \ @ \ m$ for all $x \in \mathrm{dom}(\Gamma)$. Immediate application of rule IEnv allows us to conclude that $(\gamma_i)_{i<m} : \Gamma \ @ \ m$. This proves what we wanted.

□

A.2.2 Determinism

In the following, Theorem A.2 and Lemma A.7 are proved by mutual induction.

Theorem A.2 (Determinism)**.** *If* $M \,\|\, s; (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s'_1; (\delta_i^1)_{i<n}$ *and* $M \,\|\, s; (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s'_2; (\delta_i^2)_{i<n}$*, then* $s'_1 = s'_2$ *and* $\delta_i^1 = \delta_i^2$ *for all* i*.*

Proof. Let us denote:

$$(\pi_1) \equiv M \parallel s \,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s'_1 \,;(\delta_i^1)_{i<n}$$
$$(\pi_2) \equiv M \parallel s \,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s'_2 \,;(\delta_i^2)_{i<n}$$

We proceed by structural induction over (π_1).

- Rule EZero. We have $n = 0$, $(\delta_i^1)_{i<0} = \varepsilon$ and $s'_1 = s$. Thus, substitution in (π_2) and immediate inversion yields that rule EZero applies and $(\delta_i^2)_{i<0} = \varepsilon$ and $s'_2 = s$. This gives

the following, which concludes the proof:

$$(\delta_i^1)_{i<0} = (\delta_i^2)_{i<0} = \varepsilon$$
$$s'1 = s_2' = s$$

- Rule ESTEP. We have $1 \leqslant n < \omega$, and:

$$M \parallel s\,;(\gamma_i)_{i<n-1} \stackrel{n-1}{\Longrightarrow} s_1''\,;(\delta_i^1)_{i<n-1} \tag{a}$$
$$M \parallel s_1''\,;\gamma_{n-1} \Longrightarrow s_1'\,;\delta_{n-1}^1 \tag{b}$$

Substitution of n in (π_2) and immediate inversion yields that rule ESTEP applies, and:

$$M \parallel s\,;(\gamma_i)_{i<n-1} \stackrel{n-1}{\Longrightarrow} s_2''\,;(\delta_i^2)_{i<n-1} \tag{c}$$
$$M \parallel s_2''\,;\gamma_{n-1} \Longrightarrow s_2'\,;\delta_{n-1}^2 \tag{d}$$

Application of the induction hypothesis over eq. (a) and eq. (c) gives:

$$\delta_i^1 = \delta_i^2 \text{ forall } i < n-1$$
$$s_1'' = s_2''$$

Application of the mutual induction hypothesis yields that Lemma A.7 applies on eq. (b) and eq. (d):

$$\delta_{n-1}^1 = \delta_{n-1}^2$$
$$s_1' = s_2'$$

This gives the following, which concludes the proof:

$$\delta_i^1 = \delta_i^2 \text{ forall } i < n$$
$$s_1' = s_2'$$

- Rule EOMEGA. We have $n = \omega$, and:

$$\forall m < \omega.\ M \parallel s\,;(\gamma_i)_{i<m} \stackrel{m}{\Longrightarrow} s_{1,m}''\,;(\delta_i^1)_{i<m} \tag{e}$$
$$s_1' = \mathbf{done} \tag{f}$$

Substitution of n in (π_2) and immediate inversion yields that rule ESTEP applies, and:

$$\forall m < \omega.\ M \parallel s\,;(\gamma_i)_{i<m} \stackrel{m}{\Longrightarrow} s_{2,m}''\,;(\delta_i^2)_{i<m} \tag{g}$$
$$s_2' = \mathbf{done} \tag{h}$$

Application of the induction hypothesis over eq. (e) and eq. (g) gives:

$$\delta_m^1 = \delta_m^2 \text{ forall } m < \omega$$
$$s_{1,m}'' = s_{2,m}''$$

The above especially gives $(\delta_i^1)_{i<\omega} = (\delta_i^2)_{i<\omega}$. Since $s_1' = s_2' = \mathbf{done}$, this concludes the proof.

□

Lemma A.7 (Determinism, single-step)**.** *If we have* $M \parallel s\,;\gamma \Longrightarrow s'_1\,;\delta_1$ *and* $M \parallel s\,;\gamma \Longrightarrow s'_2\,;\delta_2$ *then* $s'_1 = s'_2$ *and* $\delta_1 = \delta_2$*.*

Proof. Let us denote:

$$(\pi_1) \equiv M \parallel s\,;\gamma \Longrightarrow s'_1\,;\delta_1$$
$$(\pi_2) \equiv M \parallel s\,;\gamma \Longrightarrow s'_2\,;\delta_2$$

We proceed by structural induction over (π_1). For most cases, immediate inversion of the rule gives us values for M, s and γ. Then, by substitution, we notice that inversion of (pi_2) gives the exact same rule. Application of the induction hypothesis yields the desired result. The only cases which requires a slightly different argument are those of rules EBy and EByI. Since their premises make use of the many-step judgement, we must use Theorem A.2 by using the mutual induction hypothesis. Due to the large number of rules and the many uninteresting cases, we will give a detailed proof only for those two cases, and rule EPair (as a more general example).

- Rule EPair. We have:

$$\frac{N \parallel s_N\,;\gamma \Longrightarrow s'_{1,N}\,;\delta_{1,N} \qquad P \parallel s_P\,;\gamma \Longrightarrow s'_{1,P}\,;\delta_{1,P}}{\underbrace{(N,P)}_{=M} \parallel \underbrace{(s_N,s_P)}_{=s}\,;\gamma \Longrightarrow \underbrace{(s'_{1,N},s'_{1,P})}_{=s'_1}\,;\underbrace{(\delta_{1,N},\delta_{1,P})}_{=\delta_1}} \tag{a}$$

  By immediate substitution of M, s and γ in (π_2), we obtain that its last deduction rule is also EPair. This gives:

$$\frac{N \parallel s_N\,;\gamma \Longrightarrow s'_{2,N}\,;\delta_{2,N} \qquad P \parallel s_P\,;\gamma \Longrightarrow s'_{2,P}\,;\delta_{2,P}}{\underbrace{(N,P)}_{=M} \parallel \underbrace{(s_N,s_P)}_{=s}\,;\gamma \Longrightarrow \underbrace{(s'_{2,N},s'_{2,P})}_{=s'_2}\,;\underbrace{(\delta_{2,N},\delta_{2,P})}_{=\delta_2}} \tag{b}$$

  Inversion of eq. (a) and eq. (b) yields:

$$N \parallel s_N\,;\gamma \Longrightarrow s'_{1,N}\,;\delta_{1,N} \quad P \parallel s_P\,;\gamma \Longrightarrow s'_{1,P}\,;\delta_{1,P}$$
$$N \parallel s_N\,;\gamma \Longrightarrow s'_{2,N}\,;\delta_{2,N} \quad P \parallel s_P\,;\gamma \Longrightarrow s'_{2,P}\,;\delta_{2,P}$$

  By the induction hypothesis, we have $s'_{1,N} = s'_{2,N}$, $s'_{1,P} = s'_{2,P}$, as well as $\delta_{1,N} = \delta_{2,N}$ and $\delta_{1,P} = \delta_{2,P}$. Thus, we have the following, which concludes the proof:

$$s'_1 = (s'_{1,N}, s'_{1,P}) = (s'_{2,N}, s'_{2,P}) = s'_2$$
$$\delta_1 = (\delta_{1,N}, \delta_{1,P}) = (\delta_{2,N}, \delta_{2,P}) = \delta_2$$

- Rule EByI. We have:

$$\frac{N \parallel \mathbf{stop}\,;(\gamma_i)_{i<p(1)} \overset{p(1)}{\Longrightarrow} s''_1\,;(\delta^1_i)_{i<p(1)}}{\underbrace{N \text{ by } p}_{=M} \parallel \underbrace{\mathbf{stop}}_{=s}\,;\underbrace{\triangleright(\gamma_i)_{i<p(1)}}_{=\gamma} \Longrightarrow \underbrace{\mathbf{w}\,(p,s''_1)}_{=s'_1}\,;\underbrace{\triangleright(\delta^1_i)_{i<p(1)}}_{=\delta_1}} \tag{c}$$

  By immediate substitution of M, s and γ in (π_2), we obtain that its last deduction rule is

also EByI. This gives:

$$\frac{N \parallel \mathbf{stop} \,;(\gamma_i)_{i<p(1)} \xrightarrow{p(1)} s_2'' \,;(\delta_i^2)_{i<p(1)}}{\underbrace{N \text{ by } p}_{=M} \parallel \underbrace{\mathbf{stop}}_{=s} \,;\underbrace{\triangleright(\gamma_i)_{i<p(1)}}_{=\gamma} \Longrightarrow \underbrace{\mathbf{w}\,(p, s_2'')}_{=s_2'} \,;\underbrace{\triangleright(\delta_i^2)_{i<p(1)}}_{=\delta_2}} \tag{d}$$

Inversion of eq. (c) and eq. (d) yields:

$$N \parallel \mathbf{stop} \,;(\gamma_i)_{i<p(1)} \xrightarrow{p(1)} s_1'' \,;(\delta_i^1)_{i<p(1)}$$
$$N \parallel \mathbf{stop} \,;(\gamma_i)_{i<p(1)} \xrightarrow{p(1)} s_2'' \,;(\delta_i^2)_{i<p(1)}$$

By the mutual induction hypothesis, Theorem A.2 applies and gives $s_1'' = s_2''$, and $\delta_i^1 = \delta_i^2$ for all $i < p(1)$. Thus, we have the following, which concludes the proof:

$$s_1' = \mathbf{w}\,(p, s_1'') = \mathbf{w}\,(p, s_2'') = s_2'$$
$$\delta_1 = \triangleright(\delta_i^1)_{i<p(1)} = \triangleright(\delta_i^2)_{i<p(1)} = \delta_2$$

- Rule EBy. We have:

$$\frac{N \parallel s_N \,;(\gamma_i)_{i<\partial p(n)} \xrightarrow{\partial p(n)} s_1'' \,;(\delta_i^1)_{i<\partial p(n)}}{\underbrace{N \text{ by } p}_{=M} \parallel \underbrace{\mathbf{w}\,(p, s_N)}_{=s} \,;\underbrace{\triangleright(\gamma_i)_{i<\partial p(n)}}_{=\gamma} \Longrightarrow \underbrace{\mathbf{w}\,(p, s_1'')}_{=s_1'} \,;\underbrace{\triangleright(\delta_i^1)_{i<\partial p(n)}}_{=\delta_1}} \tag{e}$$

By immediate substitution of M, s and γ in (π_2), we obtain that its last deduction rule is also EBy. This gives:

$$\frac{N \parallel s_N \,;(\gamma_i)_{i<\partial p(n)} \xrightarrow{\partial p(n)} s_2'' \,;(\delta_i^2)_{i<\partial p(n)}}{\underbrace{N \text{ by } p}_{=M} \parallel \underbrace{\mathbf{w}\,(p, s_N)}_{=s} \,;\underbrace{\triangleright(\gamma_i)_{i<\partial p(n)}}_{=\gamma} \Longrightarrow \underbrace{\mathbf{w}\,(p, s_2'')}_{=s_2'} \,;\underbrace{\triangleright(\delta_i^2)_{i<\partial p(n)}}_{=\delta_2}} \tag{f}$$

Inversion of eq. (e) and eq. (f) yields:

$$N \parallel s_N \,;(\gamma_i)_{i<\partial p(n)} \xrightarrow{\partial p(n)} s_1'' \,;(\delta_i^1)_{i<\partial p(n)}$$
$$N \parallel s_N \,;(\gamma_i)_{i<\partial p(n)} \xrightarrow{\partial p(n)} s_2'' \,;(\delta_i^2)_{i<\partial p(n)}$$

By the mutual induction hypothesis, Theorem A.2 applies and gives $s_1'' = s_2''$, and $\delta_i^1 = \delta_i^2$ for all $i < p(1)$. Thus, we have the following, which concludes the proof:

$$s_1' = \mathbf{w}\,(p, s_1'') = \mathbf{w}\,(p, s_2'') = s_2'$$
$$\delta_1 = \triangleright(\delta_i^1)_{i<p(1)} = \triangleright(\delta_i^2)_{i<p(1)} = \delta_2$$

□

A.2.3 Type safety

In order to simplify our proof, we deal with the base case $n = 0$ in a separate lemma.

Lemma A.8 (Type safety at $n = 0$)**.** *Let* $\Gamma \vdash M : A \mid S$.

$$\textit{If we have } \begin{cases} (\gamma_i)_{i<0} : \Gamma \ @ \ 0 \\ M \parallel \mathbf{stop} \,;(\gamma_i)_{i<0} \overset{0}{\Longrightarrow} s \,;(\delta_i)_{i<0} \end{cases} \textit{— then } \begin{cases} (\delta_i)_{i<0} : A \ @ \ 0 \\ s : S \,\mathfrak{G}\, 0 \end{cases}$$

Proof. Since $n = 0$, $(\gamma_i)_i = \varepsilon$. By EZero, we have $M \parallel \mathbf{stop} \,;\varepsilon \overset{0}{\Longrightarrow} \mathbf{stop} \,;\varepsilon$. Thus, $(\delta_i)_i = \varepsilon$ and $s = \mathbf{stop}$. Rules IStop and SStop yield $(\delta_i)_i = \varepsilon : A \ @ \ 0$ and $s = \mathbf{stop} : S \,\mathfrak{G}\, 0$ – which is what we wanted. □

Let us restate Theorem A.3:

Theorem A.3 (Type safety)**.** *Let* $\Gamma \vdash M : A \mid S$*, and* $n \leqslant \omega$*. For all* $(\gamma_i)_{i<n} : \Gamma \ @ \ n$ *and* $(\delta_i)_{i<n}$ *such that* $M \parallel \mathbf{stop} \,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s \,;(\delta_i)_{i<n}$*, we have* $(\delta_i)_{i<n} : A \ @ \ n$ *and* $s : S \,\mathfrak{G}\, n$.

Proof. Let M be a well-typed term such that $(\pi) : \Gamma \vdash M : A \mid S$. By induction over (π).

Almost all cases can be dealt with by inverting the typing derivation, obtaining the shape of the term, observing that subject reduction holds for subterms due to the induction hypothesis, and using these hypotheses to conclude that the property holds for the term. Thus, we will focus on three nontrivial examples – By, Rec, and Delay.

- Case Rec. We have $\Gamma \vdash \mathsf{rec}\, x^A.\, M : A \mid S \times \int A$. By inversion of Rec, we obtain:

$$\Gamma,\, x : \mathbf{lat} \Rrightarrow A \vdash M : A \mid S \tag{a}$$

  Now, we proceed by induction over $n \in \omega + 1$.

  - $\underline{n = 0}$. True by Lemma A.8.
  - $\underline{1 \leqslant n < \omega}$. Suppose that we have:

$$(\gamma_i)_{i<n} : \Gamma \ @ \ n \tag{b}$$

$$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s \,;(\delta_i)_{i<n} \tag{c}$$

    By inversion of EStep in (c), we also have:

$$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;(\gamma_i)_{i<n-1} \overset{n-1}{\Longrightarrow} s_I \,;(\delta_i)_{i<n-1} \tag{d}$$

    By Theorem A.1, we have $(\gamma_i)_{i<n-1} : \Gamma \ @ \ n-1$. Thus, the induction hypothesis over n applies to (d), and immediate application of IWarp yields:

$$\begin{gathered} (\delta_i)_{i<n-1} : A \ @ \ n-1 \\ (\triangleright\varepsilon) \oplus \left(\triangleright(\delta_{i-1})\right)_{1\leqslant i<n} : \mathbf{lat} \Rrightarrow A \ @ \ n \end{gathered} \tag{e}$$

    Applying IEnv to (b) and (e) gives:

$$\left(\gamma_0\left[\triangleright\varepsilon/x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/x\right]\right)_{1\leqslant i<n} : \mathbf{lat} \Rrightarrow A \ @ \ n \tag{f}$$

    Finally, we apply Lemma A.2 to (d) – this yields:

$$\exists s'.\ \begin{cases} M \parallel \mathbf{stop} \,;\left(\gamma_0\left[\triangleright\varepsilon/x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/x\right]\right)_{1\leqslant i<n} \overset{n}{\Longrightarrow} s' \,;(\delta_i)_{i<n} \\ s = \left(s', \mathbf{inc}\left((\delta_i)_{i<n}\right)\right) \end{cases} \tag{g}$$

Since, by (a), and (f):

$$\Gamma,\, x{:}\mathbf{lat} \Rrightarrow A \vdash M : A \mid S$$
$$\left(\gamma_0\left[\triangleright\varepsilon/x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/x\right]\right)_{1\leqslant i<n} : \mathbf{lat} \Rrightarrow A \mathbin{@} n$$

Then, the induction hypothesis applies to (g) and yields:

$$(\delta_i)_{i<n} : A \mathbin{@} n$$
$$s' : S \mathbin{\mathfrak{G}} n$$

Immediate application of SPair and SIncr gives us:

$$(\delta_i)_{i<n} : A \mathbin{@} n$$
$$s = (s', \mathbf{inc}\,((\delta_i)_{i<n})) : S \times \textstyle\int A \mathbin{\mathfrak{G}} n$$

Which is what we wanted.

– $\underline{n = \omega}$. Suppose that we have:

$$(\gamma_i)_{i<\omega} : \Gamma \mathbin{@} \omega \tag{h}$$
$$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop}\,;(\gamma_i)_{i<\omega} \overset{\omega}{\Longrightarrow} s\,;(\delta_i)_{i<\omega} \tag{i}$$

By inversion of EOmega in (i), we have:

$$s = \mathbf{done} \tag{j}$$
$$\forall m. \quad \mathsf{rec}\, x^A.\, M \parallel \mathbf{stop}\,;(\gamma_i)_{i<m} \overset{m}{\Longrightarrow} s_m\,;(\delta_i)_{i<m} \tag{k}$$

By SDone, we obtain :

$$s = \mathbf{done} : S \times \textstyle\int A \mathbin{\mathfrak{G}} \omega \tag{l}$$

By applying Theorem A.1 on (h), then $(\gamma_i)_{i<m} : \Gamma \mathbin{@} m$ for all m. Consequently, the induction hypothesis applies on (k) and yields:

$$\forall m.\ (\delta_i)_{i<m} : A \mathbin{@} m$$

Which by immediate application of IBox yields:

$$(\delta_i)_{i<\omega} : A \mathbin{@} \omega \tag{m}$$

In the end, (l) and (m) prove what we wanted.

- Case By. We have $p \Rrightarrow \Gamma \vdash M \,\mathsf{by}\, p : p \Rrightarrow A \mid p \Rrightarrow S$. By inversion of By, we obtain:

$$\Gamma \vdash M : A \mid S \tag{a}$$

If $n = 0$, Lemma A.8 proves what we want. Thus, suppose that $n > 0$.

Let:

$$(\gamma_i)_{i<n} : p \Rrightarrow \Gamma \mathbin{@} n \tag{b}$$
$$M \,\mathsf{by}\, p \parallel \mathbf{stop}\,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s\,;(\delta_i)_{i<n} \tag{c}$$

Inversion of IEnv and IWarp in (b) gives:

$$\exists (\gamma'_i)_{i<p(n)}.\ \begin{cases} (\gamma'_i)_{i<p(n)} : \Gamma\ @\ p(n) \\ \gamma_i = \triangleright(\gamma'_j)_{p(i)\leqslant j<p(i+1)} \text{ for all } i<n \end{cases} \tag{d}$$

Substitution of (d) in (c) and immediate application of Lemma A.1 yields:

$$\exists s'.\ \exists (\delta'_i)_{i<p(n)}.\ \begin{cases} M \parallel \mathbf{stop}\,;(\gamma'_i)_{i<p(n)} \overset{p(n)}{\Longrightarrow} s'\,;(\delta'_i)_{i<p(n)} \\ \delta_i = \triangleright(\delta'_j)_{p(i)<j\leqslant p(i+1)} \text{ for all } i<n \\ s = \begin{cases} \mathbf{w}\,(p,s') & \text{if } n<\omega \\ \mathbf{done} & \text{elsewise} \end{cases} \end{cases} \tag{e}$$

Since (d) gives that $(\gamma'_i)_{i<p(n)} : \Gamma\ @\ p(n)$, and (a) gives that M is well-typed – the induction hypothesis applies on (e) and gives:

$$(\delta'_i)_{i<p(n)} : A\ @\ p(n)$$
$$s' : S\ \mathfrak{G}\ p(n)$$

Immediate application of IWarp, and SWarp if $n<\omega$ or SDone if $n=\omega$ yield:

$$\left(\triangleright(\delta'_j)_{p(i)<j\leqslant p(i+1)}\right)_{i<n} : p \Rrightarrow A\ @\ n$$
$$\begin{cases} \mathbf{w}\,(p,s') : p \Rrightarrow S\ \mathfrak{G}\ n & \text{if } n<\omega \\ \mathbf{done} : p \Rrightarrow S\ \mathfrak{G}\ \omega & \text{elsewise} \end{cases}$$

Substitution of both equalities in (e) yield what we wanted.

- Case Delay. We have $\Gamma \vdash \mathsf{delay}^{q\leqslant p}(M) : q \Rrightarrow A \mid S \times \int(p \Rrightarrow A)$. By inversion of Delay, we obtain:

$$\Gamma \vdash M : p \Rrightarrow A \mid S \text{ and } q \leqslant p \tag{a}$$

If $n=0$, Lemma A.8 proves what we want. Thus, suppose that $n>0$. Let:

$$(\gamma_i)_{i<n} : \Gamma\ @\ n \tag{b}$$
$$\mathsf{delay}^{q\leqslant p}(M) \parallel \mathbf{stop}\,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s\,;(\delta_i)_{i<n} \tag{c}$$

Application of Lemma A.3 yields:

$$\exists s'.\ \exists (\delta'_i)_{i<p(n)}.\ \begin{cases} M \parallel \mathbf{stop}\,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s'\,;\left(\triangleright(\delta'_i)_{p(i)<j\leqslant p(i+1)}\right)_{i<n} \\ \delta_i = \triangleright(\delta'_j)_{q(i)<j\leqslant q(i+1)} \text{ for all } i<n \\ s = \begin{cases} \left(s', \mathbf{inc}\left(\left(\triangleright(\delta'_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}\right)\right) & \text{if } n<\omega \\ \mathbf{done} & \text{otherwise} \end{cases} \end{cases} \tag{d}$$

Since (b) gives that $(\gamma_i)_{i<n} : \Gamma\ @\ n$, and (a) gives that M is well-typed – the induction hypothesis applies on (d) and gives:

$$\left(\triangleright(\delta'_i)_{p(i)<j\leqslant p(i+1)}\right)_{i<n} : p \Rrightarrow A\ @\ n \tag{e}$$
$$s' : S\ \mathfrak{G}\ n \tag{f}$$

Inversion of IWarp in (e) gives that $(\delta'_i)_{i<p(n)} : A \mathbin{@} p(n)$. Since the typing of increments is monotonic (cf. Theorem A.1), and $q(n) \leqslant p(n)$ (for we have $q \leqslant p$ by (a)) – we have that $(\delta'_i)_{i<q(n)} : A \mathbin{@} q(n)$.

Immediate application of IWarp proves that:

$$\left(\triangleright(\delta'_i)_{q(i)<j\leqslant q(i+1)}\right)_{i<n} : q \Rrightarrow A \mathbin{@} n \tag{g}$$

If $n = \omega$, then $s = \mathbf{done}$, and SDone immediately gives that $s : S \mathfrak{G} \omega$. Otherwise, if $n < \omega$, application of SIncr on (e) followed immediate application of SPair with (f) give:

$$\left(s', \mathbf{inc}\left(\left(\triangleright(\delta'_i)_{p(i)<j\leqslant p(i+1)}\right)_{i<n}\right)\right) : S \times \textstyle\int(p \Rrightarrow A) \mathfrak{G} n \tag{h}$$

Substitution of s and δ_i in (h) and (g) yield what we wanted.

□

A.2.4 Totality

Theorem A.4 (Totality)**.** *Let $\Gamma \vdash M : A \mid S$, and $n \leqslant \omega$. For all $(\gamma_i)_{i<n} : \Gamma \mathbin{@} n$, there exists s and $(\delta_i)_{i\leq n}$ such that $M \parallel \mathbf{stop} ; (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s ; (\delta_i)_{i<n}$.*

Proof. Let M be a well-typed term such that $(\pi) : \Gamma \vdash M : A \mid S$. By induction over (π).

- Var. Inversion of this rule gives $M = x$ and $x \in \mathrm{dom}(\Gamma)$ – (π) rewrites to :

$$\underbrace{\Gamma', x{:}A}_{\Gamma} \vdash x : A \mid \mathbb{1} \tag{a}$$

We proceed by case over n.

  - $\underline{n = 0}$. By immediate application of EZero.
  - $\underline{n > 0}$. Suppose we have $(\gamma_i)_{i<n} : \Gamma \mathbin{@} n$. Inversion of IEnv yields that for all $i < n$, we have $\mathrm{dom}(\gamma_i) = \mathrm{dom}(\Gamma)$.
    Since (a) gives $x \in \mathrm{dom}(\Gamma)$, we have that for all $i < n$, $x \in \mathrm{dom}(\gamma_i)$. Thus, Lemma A.4 applies, which proves what we wanted.

- Rec. We have $\Gamma \vdash \mathsf{rec}\, x^A.\, M : A \mid S \times \int A$. Immediate inversion of Rec gives:

$$\Gamma, x{:}\mathbf{lat} \Rrightarrow A \vdash M : A \mid S \tag{a}$$

By induction over n.

  - $\underline{n = 0}$. By immediate application of EZero.
  - $\underline{1 \leqslant n < \omega}$. Suppose we have $(\gamma_i)_{i<n} : \Gamma \mathbin{@} n$. Since increment typing is monotonic (cf. Theorem A.1), it also gives $(\gamma_i)_{i<n-1} : \Gamma \mathbin{@} n-1$. Thus, the induction hypothesis over n applies and gives:

$$\exists s_I.\ \exists(\delta_i)_{i<n-1}.\ \mathsf{rec}\, x^A.\, N \parallel \mathbf{stop} ; (\gamma_i)_{i<n-1} \overset{n-1}{\Longrightarrow} s_I ; (\delta_i)_{i<n-1} \tag{b}$$

By applying subject reduction (cf. Theorem A.3) on (b), then immediate application of IWarp and IEnv, we respectively obtain:

$$(\delta_i)_{i<n-1} : A \mathbin{@} n-1$$
$$(\triangleright\varepsilon) \oplus (\triangleright(\delta_{i-1}))_{1\leqslant i<n} : \mathbf{lat} \Rrightarrow \Gamma \mathbin{@} n$$
$$\left(\gamma_0\left[\triangleright\varepsilon/x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/x\right]\right)_{1\leqslant i<n} : \Gamma, x{:}\mathbf{lat} \Rrightarrow A \mathbin{@} n$$

This allows us to apply on the subterm N the induction hypothesis over (π):

$$N \parallel \mathbf{stop} \,;\, \left(\gamma_0\left[\triangleright\varepsilon/x\right]\right) \oplus \left(\gamma_i\left[\triangleright(\delta_{i-1})/x\right]\right)_{1\leqslant i<n} \overset{n}{\Longrightarrow} s' \,;\, (\delta_i)_{i<n}$$

Immediate application of Lemma A.6 proves what we wanted:

$$\mathsf{rec}\, x^A.\, N \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} \left(s', \mathbf{inc}\left((\delta_i)_{i<n}\right)\right) \,;\, (\delta_i)_{i<n} \tag{c}$$

– $\underline{n = \omega}$. Suppose we have $(\gamma_i)_{i<\omega} : \Gamma \mathbin{@} \omega$. Inversion of IBox and IEnv gives that for all $m < \omega$, we have $(\gamma_i)_{i<m} : \Gamma \mathbin{@} m$. Thus, the induction hypothesis over n applies and gives:

$$\forall m < \omega. \quad \exists s'_m. \quad \mathsf{rec}\, x^A.\, M \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<m} \overset{m}{\Longrightarrow} s'_m \,;\, (\delta_i)_{i<m}$$

Immediate application of EOmega yields what we wanted.

- $\boxed{\text{By}}$. We have $p \Rrightarrow \Gamma \vdash M \mathsf{\ by\ } p : p \Rrightarrow A \mid p \Rrightarrow S$. Immediate inversion of By gives us:

$$\Gamma \vdash M : A \mid S \tag{a}$$

We proceed by case over n.

– $\underline{n = 0}$. By immediate application of EZero.

– $\underline{n > 0}$. Suppose we have $(\gamma_i)_{i<n} : p \Rrightarrow \Gamma \mathbin{@} n$. Inversion of IWarp and IEnv gives:

$$\exists (\gamma'_i)_{i<p(n)}. \quad \begin{cases} (\gamma'_i)_{i<p(n)} : A \mathbin{@} p(n) \\ \gamma_i = \triangleright\left(\gamma'_j\right)_{p(i)\leqslant j<p(i+1)} \text{ for all } i < n \end{cases} \tag{b}$$

Thus, the induction hypothesis applies on (a) and yields:

$$M \parallel \mathbf{stop} \,;\, (\gamma'_i)_{i<p(n)} \overset{p(n)}{\Longrightarrow} s' \,;\, (\delta_i)_{i<p(n)}$$

Applying Lemma A.5 proves what we wanted:

$$M \mathsf{\ by\ } p \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} \mathbf{w}\,(p, s') \,;\, \left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}$$

□

A.3 Correctness

A.3.1 Preliminaries

Recall Lemma A.9, presented in Section 3.3.

Lemma A.9. *If* $M\,;E\,;x\,;V \Uparrow_i^k V'$ *then for all* j *such that* $i \leqslant j \leqslant k$, *there is* V'' *such that* $M\,;\lfloor E\rfloor_j\,;x\,;V \Uparrow_i^j V''$ *and* $M\,;E\,;x\,;V'' \Uparrow_j^k V'$.

Proof. Let's proceed by induction over $k-i$.

- Base case. We have $k-i=0$, which in turn gives gives $i=j=k$.
  We have $N\,;E\,;x\,;V \Uparrow_i^k V'$. By PIFinish, this gives $V=V'$. Set $V''=V$. Since $i=j$, PIFinish yields $N\,;E\,;x\,;V \Uparrow_i^j V''$. Since $j=k$, PIFinish yields $N\,;E\,;x\,;V \Uparrow_j^k V'$. This concludes our proof for $k=0$

- Inductive case. Set $k+1-i=n$. Suppose the theorem is true for every value of $k-i\leqslant n$. If $i=j$, the theorem is trivially true, since rule PIFinish gives $V''=V$. Thus, in all the following, we will suppose that $j>i$.
  Suppose that $N\,;E\,;x\,;V \Uparrow_i^{k+1} V'$. By inversion of rule PIIter, we obtain:
  $$\begin{cases} N\,;\lfloor E\rfloor_{i+1}\left[\mathbf{w}\,(\mathbf{lat},V)\,/_x\right] \Downarrow_{i+1} V_0 \\ N\,;E\,;x\,;V_0 \Uparrow_{i+1}^{k+1} V' \end{cases}$$
  Since $i+1\leqslant j$ and $(k+1)-(i+1)=n$, applying the induction hypothesis on $N;E;x;V_0 \Uparrow_{i+1}^{k+1} V'$ is allowed, and yields:
  $$\begin{cases} N\,;E\,;x\,;V_0 \Uparrow_{i+1}^{j} V'' \\ N\,;E\,;x\,;V'' \Uparrow_j^{k+1} V' \end{cases}$$
  Notice that $i+1\leqslant j$ gives that $i<j$. Now, consider the following proof tree:
  $$\frac{i<j \qquad N\,;\lfloor E\rfloor_{i+1}\left[\mathbf{w}\,(\mathbf{lat},V)\,/_x\right] \Downarrow_{i+1} V_0 \qquad N\,;E\,;x\,;V_0 \Uparrow_{i+1}^{j} V''}{N\,;E\,;x\,;V \Uparrow_i^j V''}$$
  Since all hypotheses hold, this proves that $N\,;E\,;x\,;V \Uparrow_i^j V''$.
  Since we also had $N\,;E\,;x\,;V'' \Uparrow_j^{k+1} V'$, this concludes the proof. □

This lemma immediately implies the following result:

Corollary A.4.1. *If we have* $M\,;E\,;x\,;\mathbf{stop} \Uparrow_0^{n+1} V$, *then there exists a value* V' *such that* $M\,;\lfloor E\rfloor_n\,;x\,;\mathbf{stop} \Uparrow_0^n V'$ *and* $M\,;E\left[\mathbf{w}\,(\mathbf{lat},V')\,/_x\right] \Downarrow_{n+1} V$.

Proof. By Lemma A.9, with $i=0$, $j=n$, $k=n+1$. Then, notice that the iteration judgement over $n\leqslant n+1$ is equivalent to performing just one step of the reduction $\Rightarrow$. □

We then give the proof of Lemma A.11, which we will need in order to prove the full adequacy lemma. In order to do so, we need the following lemma, which handles the case of truncating a coherent slicing to a size of zero.

Lemma A.10 (Coherence of truncation at $n=0$)**.** *If* $(\delta_i)_{i<n} \models_n^A V$, *then* $(\delta_i)_{i<0} \models_0^A \lfloor V\rfloor_0$.

Proof. Since $\lfloor V\rfloor_0=\mathbf{stop}$ and $(\delta_i)_{i<0}=\varepsilon$, the proof immediately follows from the definition of the logical relation. □

We now prove the general case.

Lemma A.11 (Coherence of truncation)**.** *If* $(\delta_i)_{i<n} \models^A_n V$ *and* $m \leqslant n$*, then* $(\delta_i)_{i<m} \models^A_m \lfloor V \rfloor_m$.

Proof. We proceed by induction over the pair (A, n) using the well-founded lexicographical product order $<_{\mathsf{ty}} \times_{\mathsf{lex}} <_{\omega+1}$ between the strict subtree ordering on types $<_{\mathsf{ty}}$ and the canonical strict ordering $<_{\omega+1}$ on $\omega + 1$.

- Case $(A, 0)$. Let V, and $(\delta_i)_{i<n}$ such that $(\delta_i)_{i<n} \models^A_n V$. Since $n = 0$, we have $V = \mathbf{stop}$ and $(\delta_i)_i = \varepsilon$. Let $m \leqslant n$. This means that $m = 0$, which in turn implies that $\lfloor V \rfloor_m = \mathbf{stop}$. Thus, $(\delta_i)_{i<m} \models^A_m V$.

- Inductive case (A, n) with $1 \leqslant n < \omega$. The induction hypothesis gives us that the property holds for each (A', n') such that $A' <_{\mathsf{ty}} A \vee (A' = A \wedge n' < n)$.

  Let V, and $(\delta_i)_{i<n}$ such that $(\delta_i)_{i<n} \models^A_n V$. We now proceed by case over A:

  - $\boxed{A = \nu}$. By definition of the logical relation in Figure 12, we have $V = c$, $\delta_0 = c \in S_\nu$, and $\delta_i = \mathbf{nil}$ for all $i > 0$. Let $m \leqslant n$. If $m = 0$, Lemma A.10 proves what we wanted. Thus, suppose that $1 \leqslant m < \omega$. By definition, we have that:

    $$(c) \oplus (\mathbf{nil})_{1 \leqslant i < m} \models^\nu_m c$$

    Since $\lfloor V \rfloor_m = \lfloor c \rfloor_m = c$, we obtained what we wanted.

  - $\boxed{A = A_1 \times A_2}$. By definition of the logical relation in Figure 12, we have:

    $$\left.\begin{array}{r} \forall i < n.\ \delta_i = (\delta_i^1, \delta_i^2) \\ V = (V_1, V_2) \end{array}\right\} \text{ st. } \begin{cases} (\delta_i^1)_{i<n} \models^{A_1}_n V_1 \\ (\delta_i^2)_{i<n} \models^{A_2}_n V_2 \end{cases}$$

    Let $m \leqslant n$. If $m = 0$, Lemma A.10 proves what we wanted. Thus, suppose that $1 \leqslant m < \omega$. Since A_1 and A_2 are structural subtypes of A, the induction hypothesis applies and gives:

    $$(\delta_i^1)_{i<m} \models^{A_1}_m \lfloor V_1 \rfloor_m \qquad (\delta_i^2)_{i<m} \models^{A_2}_m \lfloor V_2 \rfloor_m$$

    By definition of the logical relation, this gives:

    $$(\delta_i^1, \delta_i^2)_{i<m} \models^{A_1 \times A_2}_m (\lfloor V_1 \rfloor_m, \lfloor V_2 \rfloor_m)$$

    Substitution of $\lfloor V \rfloor_m = (\lfloor V_1 \rfloor_m, \lfloor V_2 \rfloor_m)$ and $\delta_i = (\delta_i^1, \delta_i^2)$ in the above proves what we wanted.

  - $\boxed{A = A_1 + A_2}$. By definition of the logical relation in Figure 12, we have:

    $$\left.\begin{array}{r} \forall i < n.\ \delta_i = \mathbf{inj}_k(\delta_i') \\ V = \mathbf{inj}_k(V') \end{array}\right\} \text{ st. } (\delta_i')_{i<n} \models^{A_k}_n V'$$

    Let $m \leqslant n$. If $m = 0$, Lemma A.10 proves what we wanted. Thus, suppose that $1 \leqslant m < \omega$. Since A_1 and A_2 are structural subtypes of A, the induction hypothesis applies and gives:

    $$(\delta_i')_{i<m} \models^{A_k}_m \lfloor V' \rfloor_m$$

By definition of the logical relation, this gives:

$$(\mathbf{inj}_k(\delta'_i))_{i<m} \models_m^{A_1+A_2} \lfloor V' \rfloor_m$$

Substitution of $\lfloor V \rfloor_m = \mathbf{inj}_k(\lfloor V' \rfloor_m)$ and $\delta_i = \mathbf{inj}_k(\delta'_i)$ in the above proves what we wanted.

– $\boxed{A = p \Rrightarrow A'}$ By definition of the logical relation in Figure 12, we have:

$$\left.\begin{array}{r} \forall i < n.\ \delta_i = \triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)} \\ V = \mathbf{w}\,(p, V') \end{array}\right\} \text{ st. } (\delta'_i)_{i<p(n)} \models_{p(n)}^{A'} V'$$

Let $m \leqslant n$. Notice that $p(m) \leqslant p(n)$. If $m = 0$, Lemma A.10 proves what we wanted. Thus, suppose that $1 \leqslant m < \omega$. Since A_1 and A_2 are structural subtypes of A, the induction hypothesis applies and gives:

$$(\delta'_i)_{i<p(m)} \models_{p(m)}^{A'} \lfloor V' \rfloor_{p(m)}$$

By definition of the logical relation, this gives:

$$\left(\triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<m} \models_m^{p \Rrightarrow A'} \mathbf{w}\left(p, \lfloor V' \rfloor_{p(m)}\right)$$

Substitution of $\lfloor V \rfloor_m = \mathbf{w}\left(p, \lfloor V \rfloor_{p(m)}\right)$ and $\delta_i = \triangleright(\delta_j)_{p(i)\leqslant j<p(i+1)}$ in the above proves what we wanted.

– $\boxed{A = \mathsf{Str}\,A'}$. By definition of the logical relation in Figure 12, we have:

$$\left.\begin{array}{r} \forall i < n.\ \delta_i = \delta'_i :: \triangleright(\delta''_j)_{i-1\leqslant j<i} \\ V = V' :: V'' \end{array}\right\} \text{ st. } \begin{cases} (\delta'_i)_{i<n} \models_n^{A'} V' \\ (\delta''_i)_{i<n-1} \models_{n-1}^{\mathsf{Str}\,A'} V'' \end{cases}$$

Let $m \leqslant n$. If $m = 0$, Lemma A.10 proves what we wanted. Thus, suppose that $1 \leqslant m < \omega$. Since $(A', n) <_{\mathsf{lex}} (\mathsf{Str}\,A', n)$ and $(\mathsf{Str}\,A', n-1) <_{\mathsf{lex}} (\mathsf{Str}\,A', n)$, the induction hypothesis applies and gives:

$$(\delta'_i)_{i<m} \models_m^{A'} \lfloor V' \rfloor_m \qquad (\delta''_i)_{i<m-1} \models_{m-1}^{\mathsf{Str}\,A'} \lfloor V'' \rfloor_{m-1}$$

By definition of the logical relation, this gives:

$$\left(\delta'_i :: \triangleright(\delta''_j)_{i-1\leqslant j<i}\right)_{i<m} \models_m^{\mathsf{Str}} \lfloor V' \rfloor_m :: \lfloor V'' \rfloor_{m-1}$$

Substitution of $\lfloor V \rfloor_m = \lfloor V' \rfloor_m :: \lfloor V'' \rfloor_{m-1}$ and $\delta_i = \delta'_i :: \triangleright(\delta''_j)_{i-1\leqslant j<i}$ in the above proves what we wanted.

– $\boxed{A = A_1 \overset{S}{\Rightarrow} A_2}$. By definition of the logical relation in Figure 12, we have:

$$\forall i < n.\ \delta_i = (x.M)\{\gamma_i\}$$
$$V = (x.N)\{E\}$$

Such that for all $k \leqslant n$:

$$\forall (\delta_i')_{i<k} \models_k^{A_1} V', \exists s, (\delta_i'')_{i<k}, V'', \begin{cases} M \parallel \mathbf{stop} \,;\, \left(\gamma_i\left[\delta_i'/x\right]\right)_{i<k} \overset{k}{\Longrightarrow} s \,;\, (\delta_i'')_{i<k} \\ N \,;\, \lfloor E \rfloor_k \left[V'/x\right] \Downarrow_k V'' \\ (\delta_i'')_{i<k} \models_k^{A_2} V'' \end{cases} \tag{a}$$

Let $m \leqslant n$. If $m = 0$, Lemma A.10 proves what we wanted. Thus, suppose that $1 \leqslant m < \omega$. Since $m \leqslant n$, if the property (a) holds for all $k \leqslant n$, it also holds for all $k \leqslant m$. Due to the functoriality of truncation, we have that $\lfloor \lfloor E \rfloor_m \rfloor_k = \lfloor E \rfloor_{\min(k,m)}$. However, when $k \leqslant m$, this rewrites as $\lfloor \lfloor E \rfloor_m \rfloor_k = \lfloor E \rfloor_k$. All of these arguments thus yield that, for all $k \leqslant m$:

$$\forall (\delta_i')_{i<k} \models_k^{A_1} V', \exists s, (\delta_i'')_{i<k}, V'', \begin{cases} M \parallel \mathbf{stop} \,;\, \left(\gamma_i\left[\delta_i'/x\right]\right)_{i<k} \overset{k}{\Longrightarrow} s \,;\, (\delta_i'')_{i<k} \\ N \,;\, \lfloor \lfloor E \rfloor_m \rfloor_k \left[V'/x\right] \Downarrow_k V'' \\ (\delta_i'')_{i<k} \models_k^{A_2} V'' \end{cases} \tag{b}$$

This is precisely the right hand side of the definition of the logical relation on functions. Thus, applying said relation to the above yields:

$$((x.M)\{\gamma_i\})_{i<m} \models_m^{A \overset{S}{\Rightarrow} B} (x.N)\{\lfloor E \rfloor_m\}$$

Immediate substitution of $\lfloor V \rfloor_m = (x.M)\{\lfloor E \rfloor_m\}$ and $\delta_i = (x.M)\{\gamma_i\}$ in the above proves what we wanted.

- Inductive case (A, ω). The induction hypothesis gives us that the property holds for each (A', n') such that $A' <_{\mathsf{ty}} A \vee (A' = A \wedge n' < \omega)$.

  Let V, and $(\delta_i)_{i<\omega}$ such that $(\delta_i)_{i<\omega} \models_\omega^A V$. By definition of the logical relation, we have that:

  $$V = \mathbf{box}(M)\{E\} \text{ st. for all } m < \omega,\ (\delta_i)_{i<m} \models_m^A \lfloor \mathbf{box}(M)\{E\} \rfloor_m$$

  Since $m < \omega$ implies that $m \leqslant n$, the above equation proves what we wanted.

□

A.3.2 Adequacy lemma

Lemma A.12 (Adequacy)**.** *If* $\Gamma \vdash M : A \mid S$*, then* $M \models^{\Gamma \vdash A} M$*.*

Proof. Let M be a well-typed term such that $(\pi) : \Gamma \vdash M : A \mid S$. By induction over (π).

- VAR. Immediate inversion of (π) yields:

  $$\underbrace{\Gamma', x : A}_{\Gamma} \vdash x : A \mid S \tag{a}$$

  Let $n \leqslant \omega$ — let $(\gamma_i)_{i<n} : \Gamma \ @\ n$, and $E : \Gamma \ @\ n$. Suppose that:

  $$x \parallel \mathbf{stop} \,;\, (\gamma_i)_{i<n} \overset{n}{\Longrightarrow} - \,;\, (\delta_i)_{i<n} \tag{b}$$

  $$x \,;\, E \Downarrow_n V \tag{c}$$

  $$(\gamma_i)_{i<n} \models_n^\Gamma E \tag{d}$$

Notice that if $n = 0$, then by PZero and EZero, we have $V = \mathbf{stop}$ and $(\delta_i)_{i<n} = \varepsilon$. Therefore, by definition of the logical relation, we automatically have $(\delta_i)_{i<n} \models^A_0 V$. Consequently, we will suppose that $n > 0$ in all the following.

By hypothesis, we have $(\gamma_i)_{i<n} : \Gamma \ @ \ n$, and $E : \Gamma \ @ \ n$. Inversion of IEnv and VEnv yield:

$$\begin{aligned} &\mathrm{dom}(E) = \mathrm{dom}(\Gamma) \\ &\mathrm{dom}(E) = \mathrm{dom}(\gamma_i), \text{ for all } i < n \end{aligned}$$

By (a), we have that $x \in \mathrm{dom}(\Gamma)$. Thus, we have $x \in \mathrm{dom}(\Gamma)$ and $\forall i < n.\ x \in \mathrm{dom}(\gamma_i)$. From this, Lemma A.4 to (b) and inversion of PVar in (c) respectively yield:

$$\forall i < n.\ \delta_i = \gamma_i(x) \qquad\qquad V = E(x) \tag{e}$$

Since x is in the domain of both environments, unfolding Definition 3.1 in (d) yields:

$$(\gamma_i(x))_{i<n} \models^A_n E(x)$$

Immediate substitution of (e) concludes our proof.

- By. We have $(\pi) : p \Rrightarrow \Gamma \vdash M \text{ by } p : p \Rrightarrow A \mid p \Rrightarrow S$. Immediate inversion of (π) yields:

$$\Gamma \vdash M : A \mid S \tag{a}$$

Let $n \leqslant \omega$ — let $(\gamma_i)_{i<n} : p \Rrightarrow \Gamma \ @ \ n$, and $E : p \Rrightarrow \Gamma \ @ \ n$. Thus, there exists $(\gamma'_i)_{i<p(n)}$ and E' such that :

$$\begin{cases} (\gamma'_i)_{i<p(n)} : \Gamma \ @ \ p(n) \\ \forall i < n.\ \gamma_i = \triangleright(\gamma'_j)_{p(i)\leqslant j<p(i+1)} \end{cases} \qquad \begin{cases} E' : \Gamma \ @ \ p(n) \\ E = \mathbf{w}\,(p, E') \end{cases} \tag{b}$$

Now, suppose that:

$$M \text{ by } p \parallel \mathbf{stop} \,;\, \underbrace{\left(\triangleright(\gamma'_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n}}_{=\gamma_i} \overset{n}{\Longrightarrow} - \,;\, (\delta_i)_{i<n} \tag{c}$$

$$M \text{ by } p \,;\, \underbrace{\mathbf{w}\,(p, E')}_{=E} \Downarrow_n V \tag{d}$$

$$\left(\triangleright(\gamma'_j)_{p(i)\leqslant j<p(i+1)}\right)_{i<n} \models^{p \Rrightarrow \Gamma}_n \mathbf{w}\,(p, E') \tag{e}$$

Notice that if $n = 0$, then by PZero and EZero, we have $V = \mathbf{stop}$ and $(\delta_i)_{i<n} = \varepsilon$. Therefore, by definition of the logical relation, we have $(\delta_i)_{i<n} \models^A_0 V$. We will thus suppose that $n > 0$ in all the following.

By application of Lemma A.1 to (c), inversion of PWarp in (d), and unfolding the definition of the logical relation in (e), we obtain:

$$M \parallel \mathbf{stop} \,;\, (\gamma'_i)_{i<p(n)} \overset{p(n)}{\Longrightarrow} - \,;\, (\delta'_i)_{i<p(n)} \text{ st. } \delta_i = \triangleright(\delta'_j)_{p(i)\leqslant j<p(i+1)} \text{ for all } i < n \tag{f}$$

$$M \,;\, E' \Downarrow_{p(n)} V' \text{ st. } V = \mathbf{w}\,(p, V') \tag{g}$$

$$(\gamma'_i)_{i<p(n)} \models^{\Gamma}_{p(n)} E' \tag{h}$$

The induction hypothesis gives:

$$M \models^{\Gamma \vdash A} M$$

Its immediate application to (f), (g), and (h) yields:

$$(\delta'_i)_{i<p(n)} \models^{A}_{p(n)} V'$$

By definition of the logical relation, this gives:

$$\Big((\triangleright(\delta'_j)_{p(i)\leqslant j<p(i+1)}) \Big)_{i<n} \models^{p \Rrightarrow A}_{n} \mathbf{w}\,(p, V')$$

By substition of the equalities given in (f) and (g), we obtain the following, which concludes our proof:

$$(\delta_i)_{i<n} \models^{p \Rrightarrow A}_{n} V$$

- Rec. We have $(\pi) : \Gamma \vdash \mathsf{rec}\, x^A.\, M : A \mid S \times \int A$. Immediate inversion of (π) yields:

$$\Gamma,\, x : \mathbf{lat} \Rrightarrow A \vdash M : A \mid S \tag{a}$$

We want to prove that $\mathsf{rec}\, x^A.\, M \models^{\Gamma \vdash A} \mathsf{rec}\, x^A.\, M$, whose definition is that for all $n \leqslant \omega$,

$$\left.\begin{array}{c} \forall E, V. \\ \forall (\gamma_i)_{i<n}, (\delta_i)_{i<n}. \\ \forall s. \end{array}\quad \begin{array}{r} \mathsf{rec}\, x^A.\, M \parallel \mathbf{stop}\,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s\,;(\delta_i)_{i<n} \\ \mathsf{rec}\, x^A.\, M\,;E \Downarrow_n V \\ (\gamma_i)_{i<n} \models^{\Gamma}_{n} E \end{array}\right\} \Longrightarrow (\delta_i)_n \models^{A}_{n} V$$

We prove the property by induction over n.

- Case $n = 0$. By PZero and EZero, we have $V = \mathbf{stop}$ and $(\delta_i)_{i<n} = \varepsilon$. Therefore, by definition of the logical relation, we have $(\delta_i)_{i<n} \models^{A}_{0} V$.
- Case $1 \leqslant n < \omega$. Let $(\gamma_i)_{i<n} : \Gamma \mathbin{@} n$, and $E : \Gamma \mathbin{@} n$. Now, suppose that:

$$\mathsf{rec}\, x^A.\, M \parallel \mathbf{stop}\,;(\gamma_i)_{i<n} \overset{n}{\Longrightarrow} s\,;(\delta_i)_{i<n} \tag{b}$$

$$\mathsf{rec}\, x^A.\, M\,;E \Downarrow_n V \tag{c}$$

$$(\gamma_i)_{i<n} \models^{\Gamma}_{n} E \tag{d}$$

By application of Lemma A.2 to (b), we obtain that there exists s' such that:

$$\begin{gathered} M \parallel \mathbf{stop}\,;\big(\gamma_0[\triangleright\varepsilon/x]\big) \oplus \big(\gamma_i[\triangleright(\delta_{i-1})/x]\big)_{1\leqslant i<n} \overset{n}{\Longrightarrow} s'\,;(\delta_i)_{i<n} \\ s = \big(s', \mathbf{inc}\,((\delta_i)_{i<n})\big) \end{gathered} \tag{e}$$

By inversion of EStep in (e), we split the many step reduction and obtain that there exists s'_I such that:

$$M \parallel \mathbf{stop}\,;\big(\gamma_0[\triangleright\varepsilon/x]\big) \oplus \big(\gamma_i[\triangleright(\delta_{i-1})/x]\big)_{1\leqslant i<n-1} \overset{n-1}{\Longrightarrow} s'_I\,;(\delta_i)_{i<n-1} \tag{f}$$

$$M \parallel s'_I\,;\gamma_{n-1}[\triangleright(\delta_{n-2})/x] \Longrightarrow s'\,;\delta_{n-1} \tag{g}$$

Inversion of PRec in (c) gives us that $M\,;E\,;x\,;\mathbf{stop} \Uparrow^{n}_{0} V$. Immediate application

of Corollary A.4.1 gives us that there exists a value V_I such that:

$$M\,;\lfloor E\rfloor_{n-1}\,;x\,;\mathbf{stop}\Uparrow_0^{n-1}V_I \tag{h}$$

$$M\,;E\left[\mathbf{w}\,(\mathbf{lat},V_I)\,/x\right]\Downarrow_n V \tag{i}$$

We turn back (f) and (h) into reductions over a rec by respectively applying to them Lemma A.6 and PRec:

$$\mathsf{rec}\,x^A.\,M\parallel\mathbf{stop}\,;(\gamma_i)_{i<n-1}\overset{n-1}{\Longrightarrow}\left(s'_I,\mathbf{inc}\left((\delta_i)_{i<n-1}\right)\right);(\delta_i)_{i<n-1} \tag{j}$$

$$\mathsf{rec}\,x^A.\,M\,;\lfloor E\rfloor_{n-1}\Downarrow_{n-1}V_I \tag{k}$$

Since Lemma A.11 gives us that the logical relation preserves truncation, (d) gives us that $(\gamma_{i<n-1})\models^{\Gamma}_{n-1}\lfloor E\rfloor_{n-1}$. Thus, since (j) and (k) also hold, we can apply the induction hypothesis given by our induction over n, and we obtain:

$$(\delta_i)_{i<n-1}\models^A_{n-1}V_I$$

By definition of the logical relation, this entails:

$$(\triangleright\varepsilon)\oplus(\triangleright\delta_{i-1})_{1\leqslant i<n}\models^{\mathbf{lat}\Rightarrow A}_n\mathbf{w}\,(\mathbf{lat},V_I)$$

Combined with (d), this allows us to conclude:

$$\left(\gamma_0\left[\triangleright\varepsilon/x\right]\right)\oplus\left(\gamma_i\left[\triangleright(\delta_{i-1})/x\right]\right)_{1\leqslant i<n}\models^{\Gamma,\,x:\mathbf{lat}\Rightarrow A}_n E\left[\mathbf{w}\,(\mathbf{lat},V_I)\,/x\right] \tag{l}$$

The induction hypothesis given by our induction over (π) gives that:

$$M\models^{\Gamma,\,x:\mathbf{lat}\Rightarrow A\vdash A}M$$

Thus, since we have that the reductions (e) and (i) hold, and that their environments are coherent slicings by (l), we obtain that:

$$(\delta_i)_{i<n}\models^A_n V$$

Which is what we wanted.

– Case $n=\omega$. Let $(\gamma_i)_{i<\omega}:\Gamma\ @\ \omega$, and $E:\Gamma\ @\ \omega$. Now, suppose that:

$$\mathsf{rec}\,x^A.\,M\parallel\mathbf{stop}\,;(\gamma_i)_{i<\omega}\overset{\omega}{\Longrightarrow}s\,;(\delta_i)_{i<\omega} \tag{m}$$

$$\mathsf{rec}\,x^A.\,M\,;E\Downarrow_\omega V \tag{n}$$

$$(\gamma_i)_{i<\omega}\models^{\Gamma}_{\omega}E \tag{o}$$

By inversion of POmega in (n), we obtain:

$$V=\mathbf{box}(\mathsf{rec}\,x^A.\,M)\{E\} \tag{p}$$

Since $\mathsf{rec}\,x^A.\,M$ is well typed, by totality (Theorem 2.3) and determinism (Theorem 2.4) of the prefix reduction, we have:

$$\forall m<\omega.\ \ \mathsf{rec}\,x^A.\,M\,;\lfloor E\rfloor_m\Downarrow_m\lfloor\mathbf{box}(\mathsf{rec}\,x^A.\,M)\{E\}\rfloor_m \tag{q}$$

Then, by inversion of EOmega in (m) and application of Lemma A.11 in (o) we

obtain:

$$\forall m < \omega.\ \mathsf{rec}\, x^A.\, M \parallel \mathbf{stop}\,;(\gamma_i)_{i<m} \overset{m}{\Longrightarrow} s_m\,;(\delta_i)_{i<m} \tag{r}$$

$$\forall m < \omega.\ (\gamma_i)_{i<m} \models^{\Gamma}_m \lfloor E \rfloor_m \tag{s}$$

The induction hypothesis obtained by our induction over n applied on the three above equations yields:

$$\forall m < \omega.\ (\delta_i)_{i<m} \models^{A}_m \lfloor \mathbf{box}(\mathsf{rec}\, x^A.\, M)\{E\} \rfloor_m$$

By the definition of the logical relation given in Figure 12, this gives what we wanted:

$$(\delta_i)_{i<\omega} \models^{A}_{\omega} \mathbf{box}(\mathsf{rec}\, x^A.\, M)\{E\}$$

□

Articles courts

Connecter l'écosystème OCaml à Software Heritage via opam

Léo Andrès[1,2], Raja Boujbel[1], Louis Gesbert[1], and Dario Pinto[1]

[1] OCamlPro SAS, 21 rue de Châtillon, 75014 Paris, France
leo.andres@ocamlpro.com, raja.boujbel@ocamlpro.com, louis.gesbert@ocamlpro.com, dario.pinto@ocamlpro.com
[2] Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190 Gif-sur-Yvette, France

Résumé

Software Heritage est un projet initié par Inria ayant pour but d'archiver l'ensemble des logiciels libres disponibles sur internet. Dans cet article nous présentons Software Heritage et décrivons nos travaux en lien avec l'écosystème OCaml, opam et Software Heritage. Ces travaux comprennent notamment l'ajout à Software Heritage de modules permettant l'archivage des paquets présents sur opam, le développement d'une bibliothèque OCaml permettent de travailler avec les identifiants Software Heritage, l'ajout à opam de la possibilité de récupérer sur Software Heritage des paquets qui ne sont plus disponibles et enfin la correction du dépôt opam officiel afin de retrouver les paquets déjà manquants. Aujourd'hui, 3516 paquets opam sont déjà archivés sur Software Heritage.

1 Software Heritage

Il arrive qu'un site web ne soit plus accessible. On peut alors le retrouver au moyen de la Wayback Machine, un site web fourni par l'Internet Archive et qui permet l'accès à des sauvegardes d'instantanés de pages webs effectuées à plusieurs dates. Malheureusement, archiver l'intégralité du web est une tâche ardue et il arrive qu'on ne puisse pas retrouver un contenu disparu par ce moyen.

Lorsque les logiciels libres disparaissent d'Internet, cela peut avoir de lourdes conséquences. Un exemple bien connu est celui d'un paquet d'une dizaine de lignes de code appelé leftpad retiré de npm par son auteur, cassant au passage des milliers d'autres paquets dont React. Mais cela arrive de bien d'autres manières :

— une personne supprime un dépôt de GitHub ;
— un chercheur part à la retraite et son site web institutionnel n'est pas maintenu ;
— une forge hébergeant de nombreux logiciels ferme (par exemple Bitbucket ne permettant plus de gérer des dépôts Mercurial, Inria fermant sa forge, rachat et fermeture de Gitorious en l'espace de trois semaines, fermeture de GoogleCode) ;
— une entreprise fait faillite et son instance GitLab n'est plus accessible.

Quelles conséquences à cela ? Vous ne pouvez plus jouer à ce petit jeu vidéo écrit par un étudiant de master qui vous amusait tant ; les résultats d'un article de recherche ne sont plus reproductibles ; des milliers de bibliothèques logicielles ne sont plus disponibles ; une autre entreprise ne peut plus maintenir son logiciel phare car elle dépendait de nombreux logiciels sur cette instance GitLab.

Il existe une solution à cela : Software Heritage. Dans la suite de l'article, vous sera présenté Software Heritage et le travail effectué par OCamlPro en lien avec l'écosystème OCaml, opam

et Software Heritage. Ce travail a été rendu possible grâce à une bourse de l'Alfred P. Sloan Foundation. Software Heritage [CZ17] est une initiative à but non lucratif, dont la mission est, selon leurs propres termes, la suivante :

> We are building the universal software archive. We collect and preserve software in source code form, because software embodies our technical and scientific knowledge and humanity cannot afford the risk of losing it. Software is a precious part of our cultural heritage. We curate and make accessible all the software we collect, because only by sharing it we can guarantee its preservation in the very long term.

À l'origine du projet, il y a Inria. Le projet Software Heritage est dirigé par Roberto Di Cosmo et Stefano Zacchiroli, deux personnes réputées pour leur implication dans le monde du logiciel libre, d'OCaml ou encore du projet Debian.

Aujourd'hui, Software Heritage gère, entre autres, l'archivage de Bitbucket, cgit, CRAN, Debian, Gitea, GitHub, GitLab, Guix, GNU, Heptapod, Launchpad, NixOS, npm, PyPI et SourceForge. Au moment de l'écriture de cet article, sont archivés environ :

— 11 milliards de fichiers sources,

— 9 milliards de dossiers,

— 2 milliards de commits,

— 165 millions de projets,

— 19 millions de versions publiées de logiciels [1].

Archiver autant de logiciels demande des ressources importantes. Software Heritage dispose de nombreux parrains lui permettant de mener à bien sa mission et garantissant sa pérennité. On peut notamment citer Huawei, Microsoft, Intel, le CNRS, la Société Générale, le MESRI, Open Invention Network, Sorbonne Université, Université de Paris, Adacore, CAST Software, DINSIC, GitHub, Google, Università di Pisa, VMware, Dans, FOSSID, University of Bologna, Nokia Bell Labs et UQAM.

2 Gestion d'opam dans Software Heritage

Une partie du logiciel libre que l'on souhaite archiver dans Software Heritage est écrit dans le langage OCaml. À l'instar de nombreux langages, OCaml dispose d'un gestionnaire de paquets, Opam, qui permet d'installer facilement des bibliothèques et des exécutables à partir de leurs sources. Opam étant le gestionnaire de paquets de référence dans la communauté OCaml, il constitue de fait un point d'entrée idéal pour récupérer des sources OCaml. Nous présentons dans cette section un travail qui automatise l'archivage des paquets opam par Software Heritage.

Depuis quelques jours, tout le dépôt opam officiel est archivé ! Vous pouvez vérifier que votre paquet opam favori est bien archivé. Il est prévu d'archiver également des dépôts alternatifs. Ils peuvent être signalés sur discuss.ocaml.org. Il est notamment prévu que soit archivé le dépôt opam Inria hébergeant des paquets Coq. En effet, opam étant agnostique du langage utilisé par les paquets, notre travail permet d'héberger n'importe quel paquet pour n'importe quel langage, pourvu qu'un fichier opam ait été écrit.

Pour ajouter une nouvelle source de logiciels (*software origin*) à Software Heritage, il est nécessaire d'écrire deux éléments : un *lister* et un *loader*. C'est ce qui a été fait pour opam. Software Heritage étant écrit en Python, ces deux éléments doivent prendre la forme d'un

1. C'est en réalité plus mais pour des raisons techniques beaucoup de versions ne sont actuellement considérées que comme des *commits*.

module Python. Pour faciliter leur implémentation et leur maintenance, nous avons fait le choix de la simplicité : le paquet Debian `opam` a été ajouté comme dépendance à Software Heritage - dont toute l'architecture tourne sur des serveurs Debian - et nous nous servons du binaire `opam` dès que possible.

Le *lister* a pour but de lister tous les paquets disponibles. Pour chaque paquet, notre module produit une URL de la forme :

```
1 url = f"opam+{self.url}/packages/{page}"
```

où `self.url` est l'adresse du dépôt opam archivé et où `page` est le nom du paquet à archiver. Un simple appel à `opam list --all` est suffisant pour générer cette liste. Le code complet est disponible sur le dépôt swh-lister.

Le *loader* doit récupérer, à partir d'une URL générée par le *lister*, l'ensemble des versions disponibles d'un paquet et associer à chacune d'elles un ensemble de métadonnées et une URL vers l'archive contenant le code source. Ces informations sont récupérées en faisant des appels à `opam show` en précisant à chaque fois le champ que l'on souhaite récupérer (`url.src`, `authors`...). L'archive sera ensuite insérée automatiquement par un autre module de Software Heritage dans le graphe orienté acyclique de Merkle [PSZ19] servant à stocker tous les objets archivés. Le code complet est disponible dans le dépôt swh-loader-core.

Une fois ces deux modules implémentés et un dépôt opam choisi, Software Heritage va automatiquement et régulièrement récupérer les paquets présents sur ce dépôt. Au moment de tester le déploiement, beaucoup d'erreurs ont été rencontrées. La plupart venaient de Software Heritage, par exemple le format `.tbz` n'était pas géré. Mais beaucoup venaient du fait que des archives de paquets opam n'existaient plus, menant à une erreur 404. La plupart étaient en fait toujours disponibles dans le cache d'opam, certaines avaient été déplacées, etc. Quoi qu'il en soit, environ 80 demandes de hissage (*pull requests*) ont été ouvertes sur le dépôt opam pour corriger ces URLs.

3 La bibliothèque *swhid*

Pour permettre aux développeurs OCaml d'accéder à Software Heritage dans leurs projets, une bibliothèque OCaml *swhid* a été écrite, permettant de manipuler les identifiants persistants (*persistent identifiers*) de Software Heritage. Un *identifiant persistant* est un identifiant vers un objet archivé dans Software Heritage. En voilà un exemple :

```
swh:1:dir:4431f2b743ef6dffc837e710bdd7e2169f0c5fc9
```

Le préfixe `swh` indique tout simplement qu'il s'agit d'un *identifiant persistant* de Software Heritage. Le `1` est le numéro de version du schéma des *identifiants persistants* utilisé. Le mot-clé `dir` indique le type d'objet représenté par l'identifiant et la chaîne finale de 40 caractères est un hachage de l'objet. L'identifiant peut contenir des informations optionnelles appelées qualificatifs (*qualifiers*), par exemple :

```
swh:1:dir:4431f2b743ef6dffc837e710bdd7e2169f0c5fc9
;origin=deb://Debian/packages/freedink
;visit=swh:1:snp:f6df60118578aa9ee91672dc3958908a2bf61fdb
;anchor=swh:1:rev:db21f0afdb54c16b265754ca599869fda0ca4bfc
```

Elles ne sont pas utiles pas dans ce cadre, une explication détaillée est disponible sur la documentation des qualificatifs.

Software Heritage peut archiver cinq types d'objets différents. Voici leur description traduite depuis la documentation de Software Heritage :

— `cnt` pour `content` (aussi appelés `blobs`) : le contenu brut d'un fichier source sous forme d'une séquence d'octets, sans nom de fichier ou autre métadonnée. Le contenu des fichiers est souvent récurrent, par exemple d'une version d'un logiciel à la suivante, ou dans différents répertoires d'un même projet ou même dans des projets complètement disjoints.

— `dir` pour `directory` : une liste d'entrées nommées d'un même répertoire, chacune pointant vers d'autres artéfacts qui sont généralement des objets `cnt` ou bien des sous-répertoires. Ces entrées sont souvent associées à des métadonnées comprenant un nom et des bits de permission.

— `rev` pour `revision` (aussi appelés `commits`) : le développement logiciel au sein d'un projet spécifique est essentiellement une série de copies à partir du répertoire racine, indexées dans le temps, et qui contient la totalité du code source du projet. Le logiciel évolue quand un développeur modifie le contenu d'un ou plusieurs fichiers dans ce répertoire et enregistre ses modifications. Chaque copie enregistrée de la racine s'appelle une révision. Elle pointe vers un répertoire complètement déterminé et est équipée d'un lot arbitraire de métadonnées. Certaines sont ajoutées manuellement par le développeur (les messages de commit), d'autres sont générées par l'outil de gestion de versions (date de modification, commits précédents, etc.).

— `rel` pour `release` (aussi appelés `tags`) : toutes les révisions ne sont pas égales entres elles, et certaines sont choisies par les développeurs comme étant des jalons aussi appelées publications. Chaque *release* pointe vers le commit le plus récent dans l'historique du projet qui correspond à la-dite sortie et peut comporter des métadonnées arbitraires : nom et numéro de version, message d'annonce de sortie, des signatures cryptographiques, etc.

— `snp` pour `snapshot` : quelle que soit l'origine d'un logiciel, celle-ci offrira de multiples pointeurs vers les versions courantes du développement d'un projet. Dans le cas des systèmes de gestion de version, cela se manifeste au travers de branches (master, development, branches dites «de fonctionnalités» font évoluer un logiciel dans une direction précise) ; pour les gestionnaires de paquets aussi ce fait est visible, notamment dans l'existence de *versions (suites)* qui témoignent de la maturité individuelle des paquets qu'ils distribuent (Debian Stable, Debian Testing, ...). Le snapshot de l'origine d'un logiciel donné enregistre toutes les entrées qui s'y trouvent et ce vers quoi ces entrées pointaient à un moment donné. Par exemple, un objet snapshot peut tout aussi bien suivre le commit vers lequel la branche master pointait à un moment donné, que la sortie la plus récente d'un paquet donné dans la version stable d'une distribution.

Deux autres types d'objets spéciaux sont disponibles mais ils ne sont pas pertinents ici : `origins` et `visits`. Ils ne sont donc pas gérés dans la bibliothèque. Une explication détaillée est donnée dans la documentation des artéfacts logiciels (*software artifacts*).

3.1 Analyser syntaxiquement, valider et afficher des SWHIDs

Notre bibliothèque est capable d'analyser syntaxiquement, de valider et d'afficher des *identifiants persistants* :

```
let id = "swh:1:cnt:80131a360f0ae3d4d643f9e222591db8d4aa744c"

let () =
  match Swhid.Parse.from_string id with
  | Error e -> Format.eprintf "error: %s@." e
  | Ok id -> Format.printf "the id is: %a@." Swhid.Pp.identifier id
```

S'il y en a, les `qualificatifs` sont analysés syntaxiquement mais pas validés.

3.2 Calculer des SWHIDs

Notre bibliothèque est capable de calculer des SWHIDs pour les cinq types d'objets présentés précédemment :

```
(* some file for which we'd like to compute a SWHID *)
let content =
{|(executable
(name hello)
(modules hello))
|}

let swhid = Swhid.Compute.content_identifier content

let () =
  match swhid with
  | None -> Format.eprintf "invalid ID :S@."
  | Some swhid -> Format.printf "ID is: `%a`.@." Swhid.Pp.identifier swhid
```

À l'exécution on obtient :

```
ID is: `swh:1:cnt:f5a5bc805b67f7510d2e2eb07500f47ced8af8ca`.
```

On peut maintenant communiquer cet identifiant en sachant que le jour où notre fichier sera archivé, il aura cet identifiant. Les autres types d'objet sont plus compliqués à calculer. On laisse le lecteur curieux se référer à la documentation du module Swhid.Compute.

3.3 Télécharger des SWHIDs

Pour télécharger un objet à partir de son identifiant, une IPA (Interface de Programmation Applicative) est disponible sur Software Heritage. Il n'est généralement pas possible de récupérer directement un objet en utilisant cette API. Il faut effectuer une requête afin d'obtenir une URL où, après un petit temps de préparation, on pourra récupérer l'objet voulu. Cette procédure est assez simple pour `cnt` ou `dir` par exemple, mais plus compliquéé pour `release`, une *release* pouvant pointer vers différents types d'objets. Il faut effectuer une première requête pour obtenir l'objet vers lequel elle pointe et recommencer récursivement. De même, un `snapshot` étant un ensemble d'objets, on obtiendra un ensemble d'URLs où télécharger ces objets. Notre bibliothèque contient un module `Download` contenant une fonction pour chaque type d'objet :

```ocaml
(* an identifier we want to download (a file from FreeDink) *)
let id = "swh:1:cnt:80131a360f0ae3d4d643f9e222591db8d4aa744c"

let url =
  (* we parse the string to get a Swhid.Lang.identifier *)
  match Swhid.Parse.from_string id with
  | Error _e as e -> e
  | Ok id -> (
    (* we ask SWH for an URL from which the object can be downloaded *)
    Swhid.Download.content id

let () =
  match url with
  | Error e ->
    (* we didn't get an URL *)
    Format.eprintf
      "Can't get a download URL: %s@." e;
    exit 1
  | Ok url ->
    (* we got a valid URL ! :D *)
    Format.printf "The file can be downloaded at url `%s`.@." url
```

Et l'on obtient bien :

```
The file can be downloaded at url `https://archive.softwareheritage.org/api/1/con
↪  tent/sha1_git:80131a360f0ae3d4d643f9e222591db8d4aa744c/raw/`.
```

Une fonction `any` est aussi fournie, qui permet de récupérer une liste d'URLs pour un identifiant quelconque. La liste ne contiendra qu'un élément dans tous les cas, sauf pour `snapshot` où elle pourra en contenir plusieurs. Plus d'informations sont disponibles dans la documentation du module Swhid.Download.

4 Gestion dans opam

La dernière partie de ce travail, qui est toujours en cours, consiste à ajouter à opam la capacité de récupérer automatiquement des archives depuis Software Heritage, dans le cas où l'archive d'un paquet aurait disparu et ne serait pas non plus disponible dans le cache d'opam. Les étapes identifiées pour y parvenir sont les suivantes :

— gestion dans opam d'un champ `swhid` optionnel pour les fichiers `opam` ;

— téléchargement depuis Software Heritage de l'archive correspondant à l'identifiant contenu dans ce champ en cas de nécessité ;

— modification du dépôt opam officiel pour ajouter les SWHIDs de toutes les versions de chaque paquet.

Plusieurs solutions ont été envisagées pour la gestion du champ *swhid* dans les fichiers *opam*. L'approche initiale consistait à ajouter un nouveau type de somme de contrôle, mais cela impactait trop de parties du code. Une solution plus simple a ensuite été envisagée : ajouter

un champ *swhid* au sein du champ *url*. Elle fonctionnait correctement mais posait un problème (présent aussi dans la solution initiale) : cela n'est pas rétro-compatible et aurait empêché la mise à jour du dépôt opam officiel avec les SWHIDs avant que la majorité des clients n'aient été mis à jour. La solution retenue consiste à ajouter plutôt une *fausse* adresse au champ *mirrors* :

```
url {
  src: "..."
  mirrors: [ ... "https://swhid.opam.ocaml.org/swh:1:rev:15e2f26f2ae0f0197ce95b997b4fe024c3491f9e"
  ]
}
```

Les anciens clients échoueraient au moment de télécharger l'objet mais ils pourraient toujours utiliser le fichier opam. Les nouveaux clients pourraient quant à eux récupérer le SWHID et récupérer l'archive. De plus, il est prévu de gérer dès maintenant une adresse de la forme `swh:...` [2] afin de pouvoir remplacer les fausses adresses dans le futur. Ainsi, nous utiliserons directement le SWHID qui est une URL valide et dont le schéma est déjà enregistré à l'IANA, la société chargée notamment de gérer la zone racine des noms de domaines.

On ne devrait avoir à gérer que des objets de type *rel*, cependant du fait d'un bogue connu de Software Heritage, certaines versions ne sont considérées que comme des objets de type *rev*. La gestion des deux types d'objet est donc prévu.

Les parties de téléchargement et de mise à jour du dépôt opam sont relativement faciles. Le téléchargement étant déjà implémenté dans la bibliothèque *swhid*, il ne restera plus qu'à l'adapter dans opam. La mise à jour du dépôt opam sera quant à elle générée : on télécharge l'ensemble des archives, on calcule leur identifiant et on modifie le fichier opam à l'endroit idoine.

5 Conclusion

Les logiciels libres sont des objets de valeur. Dans cet article, nous avons décrit la façon dont Software Heritage les préserve et plus spécifiquement comment tous les logiciels publiés sur opam soient eux aussi correctement archivés. Des outils OCaml ont été développés, permettant d'interagir avec l'infrastructure Software Heritage au moyen d'une bibliothèque et en ajoutant à opam un mécanisme pour récupérer sur Software Heritage des logiciels disparus. Grâce à ce travail, c'est déjà plus de 3500 paquets qui ont été archivés et les auteurs de paquets opam ont maintenant la garantie que leur travail restera accessible à jamais via opam et Software Heritage. De même, les développeurs de logiciels OCaml savent désormais que les paquets dont ils dépendent ne risquent plus de disparaître.

Remerciements. Merci à Nicolas Dandrimont, Antoine R. Dumont, Antoine Lambert et Valentin Lorentz pour leur aide sur Software Heritage ; ainsi qu'à Roberto Di Cosmo et Jean-Christophe Filliâtre pour leur relecture.

2. C'est-à-dire `swh:1:...` et non pas `swh://...`.

Références

[CZ17] Roberto Di Cosmo et Stefano Zacchiroli. « Software Heritage : Why and How to Preserve Software Source Code ». In : *iPRES 2017 : 14th International Conference on Digital Preservation*. Kyoto, Japan, 25 sept. 2017. URL : https://hal.archives-ouvertes.fr/hal-01590958. published (cf. p. 2).

[PSZ19] Antoine Pietri, Diomidis Spinellis et Stefano Zacchiroli. « The Software Heritage Graph Dataset : Public software development under one roof ». In : *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. IEEE Press, 27 mai 2019, p. 138-142. DOI : 10.1109/MSR.2019.00030. URL : https://upsilon.cc/~zack/research/publications/msr-2019-swh.pdf. published (cf. p. 3).

Alt-Ergo-Fuzz: A fuzzer for the Alt-Ergo SMT solver

Hichem Rami Ait El Hara, Guillaume Bury, and Steven de Oliveira

OCamlPro, Paris, France

Abstract

Alt-Ergo is an open source Satisfiability Modulo Theories (SMT) solver programmed in OCaml. It was designed for program verification and it's used as a back end by other software verification tools such as Frama-C, SPARK, Why3, Atelier-B and Caveat, the reliability of which depends on the soundness of Alt-Ergo's answers and the absence of bugs in it.

Fuzzing is an efficient technique to test programs and find bugs. It works by quickly and automatically generating input data with which to test the software. American Fuzzy Lop (AFL) is one of the most well-known and most used fuzzers in both academia and the industry. It has managed to find many bugs in various programs thanks to its grey box fuzzing technique that uses genetic algorithms and program instrumentation to generate test data that maximizes code and execution path coverage in the targeted software.

In this paper we present Alt-Ergo-Fuzz, a fuzzer for Alt-Ergo that we developed with the aim of finding faults and unsoundness bugs to solve and improve its reliability. By using AFL as a back end, the Crowbar OCaml library for test case generation and the CVC5 SMT solver as a reference solver of which the answers will be used to determine whether or not Alt-Ergo's answers are correct, we managed to develop Alt-Ergo-Fuzz, which even as a work in progress and in only twenty days of testing managed to find four never found before bugs in Alt-Ergo.

1 Introduction

Computer systems are entrusted with various tasks, such as transportation, financial operations and industrial production. The roles played by software are more important and critical than ever before which means that the presence of bugs in such software could lead to serious consequences. To guarantee the absence of bugs in programs, various tools using formal methods were developed in the last decades, such as model checkers, static analysers and deductive verification platforms. These tools tend to generate a large amount of logical formulas whose validity needs to be verified. SMT solvers [2], thanks to their expressivity and efficiency, are well-adapted for such tasks. Alt-Ergo [6] is one of these SMT solvers, it was designed for program verification and it is used to determine the validity of logical formulas. Our purpose is to test it efficiently and to detect the eventual bugs it might contain. After seeing how well fuzzers managed to find bugs in other SMT solvers, mainly CVC4 [3] and Z3 [10], we decided to develop Alt-Ergo-Fuzz[1], which is the first fuzzer that was developed for Alt-Ergo.

Related work Other fuzzers were developed to test the CVC4 and Z3 SMT solvers. One of them is OPFuzz [12], it works by mutating existing SMT formulas. The mutations are done by replacing operations or function calls in the SMT formulas by other ones while preserving the well-typedness of the formulas. Another one is YinYang [13], a fuzzer that uses semantic fusion to build test oracles, then verify that the SMT solver decides the right satisfiability for the formula. The semantic fusion in question is done by taking two SMT formulas that have the same satisfiability and fusing them to produce a new formula while preserving the satisfiability,

[1] Alt-Ergo-Fuzz's sources are available at: https://github.com/hra687261/alt-ergo-fuzz

and the tested SMT solver is expected to answer with that satisfiability, if it doesn't then that is likely caused by a bug. When it comes to fuzzing OCaml programs, significant progress has been made thanks to the Crowbar [7] and Monolith [11] libraries, that make it possible to fuzz OCaml programs in a quite straightforward way.

Content of the paper In this paper, we will start by giving some necessary preliminaries about the involved tools in Alt-Ergo-Fuzz. Then we detail how it was implemented. After that we present the results of its experimentation and we talk about its limitations and the envisaged additions to it before we conclude.

2 Preliminaries

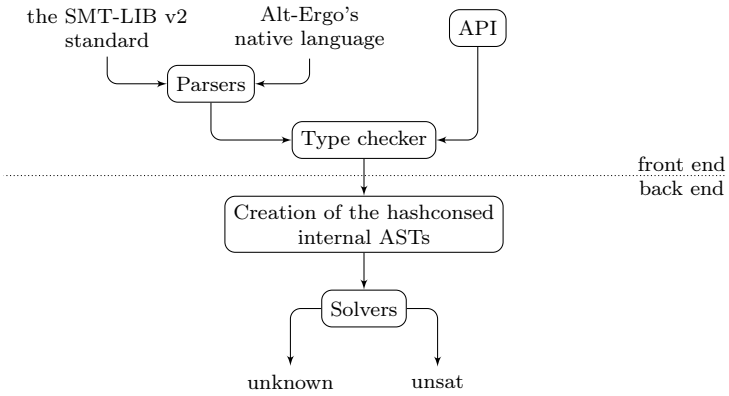


Figure 1: Alt-Ergo's architecture

The Alt-Ergo SMT solver Figure 1 represents the architecture of the Alt-Ergo SMT solver. Alt-Ergo supports both the SMT-LIB v2 standard [1], and its own native language. It can also be used as an OCaml library through its API.

Parsing a file with Alt-Ergo or building formulae using its API produces abstract syntax trees (ASTs) that are type-checked and transformed into typed ASTs. The typed ASTs are then translated into Alt-Ergo's internal language in the form of hashconsed ASTs. Alt-Ergo's core processes the resulting ASTs and uses SAT solvers, decision procedures and quantifier instantiation heuristics to decide on the satisfiability of the input SMT formulas. The result is then printed on the standard output.

Alt-Ergo has two main SAT solvers, the first one uses a Tableaux-like method and the other uses the conflict-driven clause learning (CDCL) algorithm. Each one of these solvers has a variant. In total Alt-Ergo has the following four SAT solvers:

- `Tableaux` – A functional SAT solver with Tableau boolean model simplification.
- `CDCL` – A SAT solver using the CDCL algorithm.
- `Tableaux-CDCL` – The Tableaux solver, but using a CDCL solver for boolean constraints.
- `CDCL-Tableaux` – The CDCL solver, extended with a Tableau boolean model simplification.

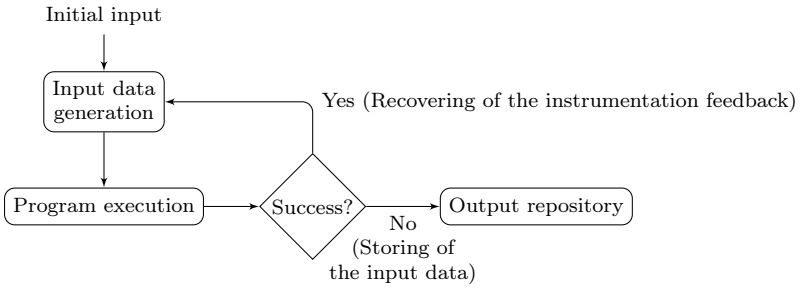


Figure 2: Functioning of AFL

Fuzz testing with AFL Fuzzing or fuzz testing is a very scalable, efficient and widely used software testing technique [8, 9]. It generally functions by continuously generating input data with which to test the software and guiding the data generation in a way that favors bug detection.

AFL [14] is a very popular fuzzer. By using code instrumentation and genetic algorithms, it managed to find many bugs in various kinds of programs. The code instrumentation allows it to get feedback on code and execution path coverage after every execution, while the genetic algorithms are used to exploit that feedback and help produce test cases that expand the code and execution path coverage.

Figure 2 illustrates the functioning of AFL. It starts by taking, as initial input, a folder containing raw data files, which are usually in the form of a random sequence of bytes that it will use as an initial test case, on which to run the targeted software. If the test succeeds, which usually means that the executable didn't crash, then the instrumentation feedback is recovered and exploited to guide future data generations. On the other hand, if the execution fails then the data on which the program was executed is stored in a file located in an output folder and the testing continues.

The Crowbar OCaml library Crowbar [7] is an OCaml library designed to do QuickCheck-like tests [5] and to help with fuzzing OCaml programs. It provides generators for OCaml's basic data types and combiners of generators that can be used to generate more complex data structures. It also allows the definition of properties that the generated data needs to verify for the tests to pass. It can also be combined with a fuzzer like AFL by using the files generated by AFL as a seed for data generation instead of generating data randomly like it's done in the classical QuickCheck testing technique. That, combined with an OCaml compiler switch that supports AFL instrumentation makes it possible to fuzz OCaml programs efficiently.

3 Implementation

In this section we present the structure of our fuzzer and how it works. We start by detailing how the data generation is done, then we describe the body of the fuzzing loop and finally we talk about how we deal with the bugs we detect.

3.1 Data generation

We defined an intermediate AST that we use as a container for the statements we generate. To make our fuzzing as efficient and as fast as possible, we made sure of generating only well-formed formulas. Well-formed means that they have to be well-typed and well-scoped. We are more interested in unsoundness bugs, as they are usually harder to detect and harder to solve, plus they could have worse consequences than other failures. The presence of unsoundness bugs usually means that the solver provides incorrect answers. By generating well-formed formulas, we can translate them directly to Alt-Ergo's internal language and provide them to the solver, without parsing or type checking them since we know that they are well-formed.

The Crowbar library provides a data type for generators that has one parameter which is the type of the data that the generator can produce. In our case, we needed to write a function called `expr_gen` that builds a generator for SMT expressions which can be used to build SMT statements. That function takes as parameters a generation context, the type of the expression we want to build a generator for, and a `fuel` parameter which is an upper bound of the depth that generated expressions can have.

The generation context is used to provide the generator with additional information that can be used in the generation of the expressions. For example, when generating the body of a function. The arguments of the function are provided to the generator so that they can be used during the generation of its body.

When `expr_gen` is called, if the `fuel` is equal to or less than zero, then it selects a generator of a terminal expression of the correct type, so it chooses either a variable generator or a literal or non-literal constant generator. Otherwise, it selects a generator of any expression of the correct type or calls a function that builds that generator. For example, if it chooses to build a generator that generates a call to some function `func`, then a function `func_call_gen` is called with the function `func`'s signature. `func_call_gen` will then call `expr_gen` with the type of each parameter of `func` and a decremented `fuel`. It can then use the generators for the arguments of `func` that it got from calling `expr_gen` to build a generator for a function call to `func`.

Well-scopedness An SMT statement is well-scoped if every identifier in the statement is either a bound variable, a declared constant or a call to a user-defined or uninterpreted function.

Variables can be bound to quantifiers, "let" expressions, patterns from a pattern matching or to a function as its parameters. In the case of quantification, the number of usable universally or existentially quantified variables by type is determined before the generation of the statements. When a statement is generated, it can contain free variables, which are then quantified, by adding a quantifier of each one of the variables. The quantifier will be positioned at the closest common predecessor (of type `bool` to keep the well-typedness) of all the instances of the variable. Which ensures that each quantified formula will be well-scoped.

When it comes to "let" expressions, the variable and its assigned value are first generated, then the body of the expression is generated with a generation context that contains the variable, which makes it possible for it to be used in the generated body. That way, it is guaranteed that each variable bound to a "let" expression will not be out of its scope.

Concerning the variables that are bound as function parameters or to patterns in a pattern matching. They are generated before the rest of the expressions are, therefore adding them as usable variables to the generation context when generating the body of the function or the consequence of the matched pattern is sufficient to guarantee that the variables will be well-scoped in the resulting expression.

The last case is the one of undeclared functions, constants or user-defined data types. The user-defined types are generated and declared before the generation of the statements so that they can be used during it, which makes their scope cover every statement. On the other hand the other identifiers that need to be declared are declared the first time a generated statement uses them. When another statement that comes after uses the same identifiers, they are not redeclared.

3.2 The body of the fuzzing loop

After being able to generate well-formed sequences of statements, it is possible to define the body of the fuzzing loop. The generated sequences of statements need to be initially translated to Alt-Ergo's internal language to make it possible to run Alt-Ergo on them and recover the answers. That makes it possible to test whether or not Alt-Ergo crashes when run on well-formed statements. So it only tests for internal crashes. To check the soundness of Alt-Ergo's answers, CVC5 [4] was chosen as a reference solver, the answers of which will be used to compare Alt-Ergo's answers to.

```
1:  function (stmts: list of SMT statements)
2:      try
3:          ae_stmts ← translate_ae(stmts)
4:          smt2_stmts ← translate_smt2(stmts)
5:
6:          cvc5_ans ← cvc5_solve(smt2_stmts) // CVC5 SMT solver
7:
8:          c_ans ← ae_c_solve(ae_stmts)       // Alt-Ergo with the CDCL solver
9:          ct_ans ← ae_ct_solve(ae_stmts)     // Alt-Ergo with the CDCL-Tableaux solver
10:         t_ans ← ae_t_solve(ae_stmts)       // Alt-Ergo with the Tableaux solver
11:         tc_ans ← ae_tc_solve(ae_stmts)     // Alt-Ergo with the Tableaux-CDCL solver
12:
13:         try
14:             cmp_answers(cvc5_ans, t_ans, tc_ans, c_ans, ct_ans)
15:         catch exn
16:             handle_unsoundness_bug(exn, stmts, cvc5_res, ae_t_res, ae_c_res)
17:         end try
18:     catch exn
19:         handle_failure_bug(exn, stmts)
20:     end try
21: end function
```

Algorithm 1: The testing function

The function represented in Algorithm 1 takes as input a list of statements, in lines 3 and 4 the statements are translated to Alt-Ergo's internal language and the SMT-LIB v2 standard. In line 6, CVC5 is called with the statements in the SMT-LIB v2 standard and its answers are recovered. In lines 8 to 11, Alt-Ergo is called with its four solvers on the statements that are in Alt-Ergo's internal language, the answers are also recovered. If a crash happens in one of the previous steps, an exception is raised, caught at line 18, handled at line 19 and the test fails. Otherwise, the answers of Alt-Ergo's solvers are compared to those of CVC5 in line 14. If there is a contradiction in the answers, then an exception is raised, caught in line 15, handled in line

16 and the test fails, otherwise nothing happens and the test passes.

3.3 Bug management

The management of bugs is the process that is applied when handling the exceptions that are raised when a test fails. In Algorithm 1 it is visible that there are two functions in line 16 and line 19 that handle exceptions. The role of those functions is to make it possible to get information about the bug and reproduce it in a simple manner. To do so, the technique that we chose consists of building a data structure that holds the necessary information about the bug, namely the list of statements that caused the bug and the exception that was raised. In the case in which the exception was raised because of contradictory answers, those answers are also collected in the data structure. That data structure is then stored in a uniquely named file, in a specific folder. The files are split across different folders according to the type of the bug that lead to their creation, whether it is a stack overflow, an unsoundness bug or another kind of bug.

Once that file is stored, it is possible then to read its data and translate the list of statements to rerun the solvers on them in order to reproduce the bug. It's also possible to translate the list of statements to the SMT-LIB v2 standard or Alt-Ergo's native language and store them respectively in a ".smt2" or ".ae" file. The file can be used to modify the statements during the debugging process or to report an issue on Alt-Ergo's issue tracker.[2]

4 Experimentation

The fuzzer was put to practice by running it on Alt-Ergo's 2.4.1 release, by using AFL's parallel mode and launching 4 instances of the fuzzer in parallel on a machine that has four cores for 20 days.

Encountered difficulties Initially the testing showed a significant problem in the efficiency of the fuzzer, its stability would decrease very quickly after it was launched and "Out of memory" exceptions were raised repeatedly. Stability is a measure of the program's determinism when it receives similar raw data. The higher it is, the easier it is for AFL to understand and learn about the behaviour of the program, which in turn improves its capability of generating input data that increases the code and execution path coverage in the tested program. These issues were due to two main reasons, the design of Alt-Ergo and the way in which the testing is done with AFL.

Internally Alt-Ergo relies significantly on side effects with the use of caches that are necessary for its reasoning. These caches come either in the form of hash tables or in the form of references to lists and other data types. That is not a problem when Alt-Ergo is used as it was designed to be used, namely by running its binary on a ".ae" or ".smt2" file, processing the statements of the file in a relational manner and returning answers. What makes it problematic in our case is the second issue that we mentioned earlier, which is how AFL works. When AFL is launched on an executable, it runs one instance of it and provides several generated raw data files to it. As long as there is no crash and no timeout is reached it continues until it reaches its chosen number of iterations. The number of iterations is chosen by AFL internally and it depends on the execution speed of the program and other parameters.

[2] Alt-Ergo's issue tracker: https://github.com/OCamlPro/alt-ergo/issues

Knowing that each raw data file generated by AFL leads to the generation of a list of statements on which Alt-Ergo's four solvers are tested. Doing that repeatedly would quickly lead to too much memory consumption and undesirable behaviour. That is because AFL expects the program to behave in a similar manner when provided with the same or very similar raw data files. That's not the case here, since the way in which a list of statements is processed changes if they are processed in relation to other statements that were processed before or independently.

To solve this issue, we added a resolution context reinitialization functionality that makes it possible to reinitialize the caches that are used by the solvers to the state in which they were initially, before processing any statements. To do so we needed to track these caches, since the functions that have side effects are not always documented as such and because OCaml uses garbage collection, developers don't usually think about freeing memory manually after using it unless they need to. The added functionality made it possible to use one instance of Alt-Ergo on many lists of statements while making sure that they are all processed independently by reinitializing the resolution context after processing each list of statements.

Results During the experimentation, the fuzzer managed to find four unique crashes of Alt-Ergo on well-formed lists of statements that were reported on Alt-Ergo's issue tracker (#474, #475, #481 and #482). There were also false positives of unsoundness bugs which after closer inspection turned out to be due to divisions by zero (#476, #477 and #479), but the fact that Alt-Ergo answered on them in a contradictory manner to CVC5 and without printing a warning or something to say that the answer was influenced by the possible presence of a division by zero is problematic.

5 Limitations

For the fuzzing to be as efficient as possible it needs to perform many tests and be able to generate data that is diverse and complex enough to be able to cover hard to access execution paths and corner cases in the program. When generating statements, the generator requires a certain number of parameters, for example, the maximum depth of a statement, the maximum number of quantified variables by type, the number of statements, etc. Choosing the right parameters to optimize the capability of the executions to find new paths and cover new code while being doable in a reasonable amount of time is not trivial. In the current version of the fuzzer, the values of the various parameters were set in an intuitive manner and chosen after doing extensive testing. It appeared during the testing that with non restrictive parameters, it is very easy to have an explosion of the sizes of the statements and produce statements that take a very long time to be processed by the solvers.

A possible workaround for this issue is to gather some statistics on the structure of SMT files from industrial or research benchmarks, and get from those statistics upper and lower bounds for each one of the parameters. These bounds can then be used to give the fuzzer the possibility of selecting the values of the parameters before generating formulas. Eventually the fuzzer should be able to select the best values for these parameters.

When formulas are too complex, not only do the solvers take too long to process them, but timeouts happen, which is not ideal because the feedback from timed out executions cannot be exploited correctly by AFL. Similarly, when the generated data leads to the detection of bugs that were previously detected. The executions are not likely to be optimal when it comes to the feedback they provide to help with the detection of new bugs. On the other hand, that might be useful because it allows the user to choose the simplest and clearest one of the test

cases that caused the bug to try and resolve it. One way to avoid redetecting bugs is to correct them quickly and relaunch the fuzzer, then again, relaunching the fuzzer means going from scratch and losing the information it gathered on the structure of the code during the time it was running, and regaining that information could require a significant amount of time.

6 Conclusion and future work

We presented in this paper Alt-Ergo-Fuzz, which is the fuzzer we developed for the Alt-Ergo SMT solver. This is the first fuzzing to be done on Alt-Ergo and more extensive testing will hopefully reveal how efficient it can be.

Alt-Ergo-Fuzz still needs a lot of work before becoming a complete software testing tool. One of the things that it needs is a shrinking technique that allows for the diminution of the size of the test cases that cause bugs, which would facilitate finding the bug and solving it. Another one would be a completeness test, for now we only test to see if Alt-Ergo crashes or answers in a contradictory manner to CVC5. Ideally, we should be able as well to test whether Alt-Ergo responds `unknown` for formulas that it is supposed to be able to decide because that might as well be hiding bugs in its reasoning.

References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. 2021. URL: http://smtlib.cs.uiowa.edu/language.shtml (cit. on p. 2).

[2] Clark Barrett and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 305–343. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_11 (cit. on p. 1).

[3] Clark Barrett et al. "CVC4". en. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, pp. 171–177. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_14 (cit. on p. 1).

[4] Clark Barrett et al. *The CVC5 automatic theorem prover*. 2021. URL: https://cvc5.github.io (cit. on p. 5).

[5] Koen Claessen and John Hughes. "QuickCheck: A lightweight tool for random testing of Haskell programs". In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP* 46 (2000). DOI: 10.1145/1988042.1988046 (cit. on p. 3).

[6] Sylvain Conchon et al. *Alt-Ergo*. URL: https://alt-ergo.ocamlpro.com (visited on 2021-10-13) (cit. on p. 1).

[7] Stephen Dolan. *Crowbar*. 2017. URL: https://github.com/stedolan/crowbar (cit. on pp. 2, 3).

[8] Jun Li, Bodong Zhao, and Chao Zhang. "Fuzzing: a survey". In: *Cybersecurity* 1.1 (2018-06), p. 6. ISSN: 2523-3246. DOI: 10.1186/s42400-018-0002-y (cit. on p. 3).

[9] Hongliang Liang et al. "Fuzzing: State of the Art". In: *IEEE Transactions on Reliability* 67.3 (2018-09), pp. 1199–1218. ISSN: 1558-1721. DOI: 10.1109/TR.2018.2834476 (cit. on p. 3).

[10] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2008, pp. 337–340. ISBN: 978-3-540-78800-3. DOI: 10.1007/978-3-540-78800-3_24 (cit. on p. 1).

[11] François Pottier. "Strong Automated Testing of OCaml Libraries". In: *JFLA 2021 - 32es Journées Francophones des Langages Applicatifs, Feb 2021, Saint Médard d'Excideuil, France*. 2021-02. URL: https://hal.inria.fr/hal-03049511 (cit. on p. 2).

[12] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "On the unusual effectiveness of type-aware operator mutations for testing SMT solvers". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020-11), 193:1–193:25. DOI: 10.1145/3428261 (cit. on p. 1).

[13] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Validating SMT solvers via semantic fusion". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020-06, pp. 718–730. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385985 (cit. on p. 1).

[14] Michał Zalewski. *American Fuzzy Lop*. 2013. URL: https://lcamtuf.coredump.cx/afl (cit. on p. 3).

Démonstrations de prototypes

Bécassine à la chasse au Coq *

Démonstration d'un prototype

Valentin Blot[1,2], Louise Dubois de Prisque[1,2], Chantal Keller[1], and Pierre Vial[1,2]

[1] Université Paris-Saclay, CNRS, ÉNS Paris-Saclay, Laboratoire de Méthodes Formelles
[2] Inria

Résumé

Nous présentons une nouvelle tactique Coq (`snipe`) permettant de prouver *automatiquement* des buts Coq en les envoyant à des prouveurs automatiques externes du premier ordre *via* des plugins. Pour ce faire, la tactique `snipe` réduit le but à un énoncé du premier ordre et enrichit le contexte local avec des énoncés explicitant pour le prouveur la sémantique du but. Nous combinons des transformations modulaires et indépendantes permettant chacune de réduire un aspect spécifique du langage de Coq au langage (plus simple) d'un prouveur automatique. À l'heure actuelle, `snipe` utilise des transformations simples mais cruciales qui explicitent des définitions et des types algébriques. Ceci permet de prouver automatiquement des buts mêlant raisonnement au premier ordre, types de données et polymorphisme. Ce prototype de tactique automatique est un premier pas vers l'implantation et la combinaison de transformations plus complexes, qui rendront Coq plus facile d'accès.

1 Preuves automatiques et preuves interactives

Le manque d'automatisation de Coq—et des assistants de preuve en général—est un obstacle à la démocratisation de son utilisation auprès des mathématiciens ou dans l'industrie. La difficulté intrinsèque qu'il y a à automatiser Coq est étroitement liée à la richesse de son langage de spécification, basé sur le **Calcul des Constructions Inductives (CIC)**. À l'inverse, les prouveurs automatiques, qui manipulent des logiques moins expressives (en particulier, **logique du premier ordre**), ne reposent pas sur un *noyau* logiquement robuste (contrairement à Coq) mais ont des heuristiques de recherches de preuve très efficaces.

L'automatisation des démonstrations dans un cadre permettant les fonctions d'ordre supérieur, le polymorphisme et de types dépendants est un problème hautement non-trivial. En revanche, il est surprenant que **Coq n'arrive pas à automatiser la preuve de nombreux énoncés du premier ordre** sur des types décidables, alors qu'elle le serait sans problème dans des prouveurs du premier ordre. Même dans le cadre de la logique du premier ordre, les utilisateurs (particulièrement les débutants) doivent être très précis dans la façon dont ils écrivent leurs preuves : par exemple, il faut spécifier comment les variables des lemmes sont instanciées et régler de nombreux détails triviaux mais fastidieux, qui seraient ignorés sur papier.

Nous allons présenter :

(a) Une méthodologie générale, sur laquelle s'appuie notre prototype, pour faciliter l'appel à des prouveurs automatiques en Coq lorsqu'il s'agit de prouver des buts se réduisant à la logique du premier ordre. Comme nous le verrons, (1) nous prouvons automatiquement en Coq des énoncés de premier ordre variés et procédons à de légères transformations du but (2) nous envoyons tous les énoncés produits et le but à un prouveur externe

(b) Une implantation de cette méthodologie utilisant le plugin Coq SMTCoq à travers la tactique `snipe` (le nom albionique de la *Bécassine des Marais*).

À titre d'exemple, si nous voulons prouver l'énoncé du premier ordre suivant

```
Goal forall (A: Type) l (a:A), hd_error l = Some a → l <> nil.
```

une preuve Coq standard est `intros A l a H. intro H'. rewrite H' in H. simpl in H. inversion H.` Cette preuve repose sur le fait que les constructeurs d'un type inductif sont disjoints. Cependant, en Coq, l'utilisateur doit être très précis dans la façon dont il combine les mots-clés, alors que, sur papier, il ne prendrait même pas la peine d'écrire la preuve.

*Cette contribution est soutenue par un partenariat entre Nomadic Labs et l'Inria.

2 Automatisation par combinaison de transformations logiques

Nous proposons un prototype de tactique permettant de réconcilier les preuves Coq avec la logique des prouveurs du premier ordre. Par exemple, nous rendons possible une preuve complètement automatique de l'exemple ci-dessus. Cette tactique s'appuie sur une méthodologie qui consiste à :

1. Prouver automatiquement en Coq dans le contexte local des énoncés auxiliaires de premier ordre explicitant les propriétés des sous-termes (fonctions, inductifs, égalités d'ordre supérieur...) du but et des hypothèses en réduisant *in fine* ce dernier à un énoncé du premier ordre. Dans notre cas, nous implantons des transformations logiques indépendantes *certifiantes* qui utilisent les outils de métaprogrammation Ltac [2] et MetaCoq [4]. *Certifiant* signifie que chaque transformation prouve aussi la correction du résultat à chaque appel (donc au cours de son exécution, et non pas en amont).
2. Envoyer ce but du premier ordre et tous les lemmes auxiliaires à un prouveur du premier ordre externe. Si ce prouveur résout le but, un terme de preuve Coq est reconstitué (avant d'être *type-checké*) à partir du certificat produit par le prouveur.

Nous appliquons cette méthodologie en Coq dans la nouvelle tactique `snipe`. Les deux phases sont implantées comme suit :

1. Nous spécifions des transformations indépendantes et n'utilisant aucun axiome (1) prouvant les énoncés spécifiant que les constructeurs d'un type inductif sont injectifs et d'images directes disjointes (2) ajoutant (dans le contexte local) les définitions des fonctions qui apparaissent dans les hypothèses ou dans le but (3) transformant les égalités de la forme `f = g` entre deux objets d'ordre supérieur en des énoncés du premier ordre de la forme `forall (x1: A1)... (xn: An): f x1 ... xn = g x1 ... xn` (4) éliminant les points-fixes anonymes (opérateur `fix`) (5) éliminant les pattern-matching par analyse de cas (6) instanciant toutes les hypothèses (et éventuellement, lemmes auxiliaires) polymorphes par tous les sous-termes de type `Type` du but (monomorphisation). Ces transformations sont toutes combinées dans la première phase de la tactique `snipe`.
2. Ensuite, cette tactique fait appel au prouveur SMT veriT [1] comme *back-end* pour décharger le but du premier ordre, à l'aide du plugin SMTCoq [3], qui permet à Coq et des prouveurs SMT de communiquer. Il est important de remarquer que, pour cette étape, SMTCoq pourrait être remplacé par n'importe quelle tactique permettant de résoudre des buts du premier ordre.

La preuve de l'exemple ci-dessus devient tout simplement un appel à la tactique `snipe` (`Proof. snipe. Qed.`).

Le plugin `Sniper` présenté ici est disponible à l'adresse suivante : https://github.com/smtcoq/sniper. Dans notre exposé, nous détaillerons ses mécanismes, que nous avons esquissés ci-dessus.

3 Vers plus d'expressivité

Dans le futur, nous aimerions pouvoir appliquer `Sniper` à des buts plus expressifs, de façon à profiter davantage des heuristiques puissantes des prouveurs automatiques. Une première étape serait de traiter le cas de certains prédicats décidables, en les plongeant dans les booléens. À plus long terme, nous pourrons tenter d'instancier automatiquement des principes d'induction avec certains prédicats booléens, et les envoyer aux prouveurs externes.

Références

[1] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit : An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[2] David Delahaye. A Tactic Language for the System Coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000.

[3] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. Smtcoq : A plug-in for integrating SMT solvers into coq. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 126–133. Springer, 2017.

[4] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *J. Autom. Reason.*, 64(5) :947–999, 2020.

Jouez à Faire Consensus Avec Mitten

Çagdas Bozman[1], Mohamed Iguernlala[1], Michael Laporte[1], Maxime Levillain[1], Alain Mebsout[1], and Sylvain Conchon[2]

Functori(1) & Nomadic Labs(2), Paris, France

Résumé

Cet article présente Mitten, un outil pour décrire finement et jouer des scénarios sur une implémentation de Tenderbake — le prochain protocole de consensus *à la pBFT* de la blockchain Tezos. Mitten est paramétrable pour filtrer et examiner les messages selon des scénarios particuliers écrits dans un DSL construit sur OCaml. Grâce à Mitten, nous avons pu écrire et simuler des scénarios subtils pour reproduire des comportements difficilement atteignables en temps normal. Nous avons également pu simuler des situations permettant d'exhiber des bugs dans l'implémentation en cours et de tester des correctifs proposés.

1 Introduction

La conception d'algorithmes de consensus est une tâche très ardue et complexe. Il convient de prendre en considération de nombreux paramètres pour en assurer la correction et la vivacité, comme le ratio d'acteurs byzantins tolérés, la désynchronisation des participants, les délais et la perte de messages sur le réseau, *etc.* Qu'ils permettent une finalité probabiliste ou déterministe, les algorithmes de consensus sont au cœur de la technologie des chaînes de blocs (blockchains). Par exemple, les algorithmes de consensus de type pBFT, qui permettent d'avoir une finalité déterministe et immédiate, sont au centre des blockchains basées sur le protocole Tendermint [5].

Pour tester Tenderbake [1], le prochain protocole de Tezos [4], nous avons développé Mitten ; un outil pour décrire finement et jouer des scénarios sur une implémentation dudit protocole, car recréer les conditions et les progressions qui amènent à une configuration voulue est particulièrement difficile, ce qui rend ces protocoles de consensus ardus à tester. Mitten est à son cœur un proxy *intercepteur* pour un réseau pair-à-pair, qui peut être paramétré pour filtrer et examiner les messages selon un scénario particulier. Les scénarios sont écrits dans un DSL construit sur OCaml, ce qui apporte une grande souplesse à l'outil. Grâce à Mitten, nous avons pu écrire et simuler des scénarios subtils pour reproduire des comportements difficilement atteignables en temps normal. Nous avons également pu simuler des situations permettant d'exhiber des bugs dans l'implémentation en cours et de tester des correctifs ou améliorations proposés.

2 Description et architecture

Ce travail fait suite à l'implémentation en TLA+ [3] du protocole Tenderbake. Il est possible de vérifier certaines propriétés et de mettre en évidence certains comportements du protocole en utilisant le model checker TLC sur le modèle TLA+. Mitten permet dans un second temps de s'assurer que ces comportements sont réalisables par l'implémentation. Mitten est conçu comme un proxy intercepteur entre différents nœuds d'un réseau pair-à-pair blockchain. Il se connecte au travers de multiples *sockets TCP* à plusieurs nœuds Tezos, configurés pour ne communiquer qu'avec le proxy. Mitten est ainsi capable de voir passer *tous* les messages échangés sur le réseau, et d'agir en conséquence (faire suivre le message, jeter le message, retarder le message, et même le modifier). Cette logique d'action est implémentée par la machine interne de Mitten (voir Figure 1), qui peut, elle-même, être paramétrée par un scénario.

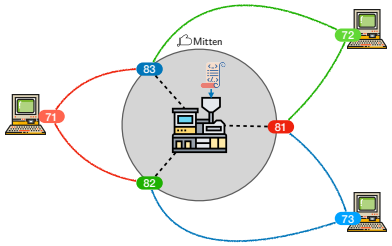


FIGURE 1 – Architecture du proxy Mitten

$$
\begin{aligned}
\langle const \rangle &::= \texttt{N ".*"} \mid [0-9]^{+} \mid \texttt{vh[a-zA-Z]}^{52} \\
\langle v_i \rangle &::= \texttt{\_} \mid \texttt{".*"} \mid \langle const \rangle \\
\langle istep \rangle &::= \texttt{propose} \ \langle v_b \rangle \ \langle v_l \rangle \ \langle v_r \rangle \ \langle v_p \rangle \ \langle v_d \rangle \\
&\quad \mid \texttt{preendorse} \ \langle v_b \rangle \ \langle v_l \rangle \ \langle v_r \rangle \ \langle v_p \rangle \ \langle v_d \rangle \\
&\quad \mid \texttt{endorse} \ \langle v_b \rangle \ \langle v_l \rangle \ \langle v_r \rangle \ \langle v_p \rangle \ \langle v_d \rangle \\
&\quad \mid \langle step \rangle \ \texttt{[\&\& | || | ->?]} \ \langle step \rangle \\
&\quad \mid \texttt{\~{}!} \langle step \rangle \\
&\quad \mid \texttt{seq[} \langle step \rangle^{\star}_{;} \texttt{]} \\
\langle step \rangle &::= \{pre\} \langle istep \rangle \{post\}
\end{aligned}
$$

FIGURE 2 – Langage de scénarios

Les scénarios pour MITTEN sont écrits dans un langage qui décrit quels messages doivent être transmis (voir Figure 2). Une étape élémentaire du scénario représenté par `propose b1 (level 3) (round 0) __ b2` laisse passer un message de (*i.e.* signé par le validateur du nœud) `b1`, qui est un "propose" au niveau 3 et round 0, avec n'importe quel payload hash au nœud `b2`. Le langage permet également de composer ces étapes élémentaires, comme des séquences, conjonctions, disjonctions, mais aussi des boucles d'attente–transmissions. Ce langage est plongé dans OCaml, et permet également de décorer les étapes avec des assertions (sous la forme de *pre* ou *post* actions/vérifications). Bénéficier d'un langage de programmation comme OCaml pour l'écriture d'assertions apporte une grande souplesse.

3 Exemple

Cet extrait de scénario avec deux nœuds `a` et `b` vérifie qu'un validateur qui a observé un *quorum de préendorsements* (PQ) (mais pas de quorum d'endorsements) propose à nouveau au round suivant avec le *même payload hash*. On voit que `a` doit proposer au niveau 3, round 0 avec un payload hash `p1`, que les validateurs s'échangent les préendorsements normalement, que `b` *endorse* (donc il a vu un PQ sur `p1`) et qu'il doit proposer au round 1 avec le même `p1`. MITTEN fait du filtrage par motif (ie. étapes du scénario) sur les messages. La variable `p1` est donc liée à une valeur concrète après le *propose* au round 0.

```
...
let scenario = seq [
  [ preendorse __ (level 3) __ __ [__];
    endorse __ (level 3) __ __ [__];
  ] <>?
  propose a (level 3) (round 0) p1 [a;b]
    ~post:[Exec (Helpers.inject_dummy n_b)];
  preendorse a (level 3) (round 0) __ [a;b] ;
  preendorse b (level 3) (round 0) __ [a;b] ;
  endorse b (level 3) (round 0) __ [a;b];
  propose b (level 3) (round 1) p1 [a;b];
]
let () = run_scenario { code; timeout = None;
    nodes; parameters; constraints = [] }
```

4 Conclusion et travaux futurs

Bien que MITTEN ait été initialement conçu pour Tezos, son architecture modulaire le rend adaptable à d'autres usages. En effet, la partie proxy paramétrable peut être utilisée sur n'importe quel réseau (modulo quelques modifications), ceci afin d'observer, tester et contrôler les échanges. Par exemple, des acteurs byzantins pourraient être simulés par MITTEN, comme dans l'approche de Twins [2]. Il serait aussi pertinent d'observer les échanges de messages en temps réel et de vérifier qu'ils représentent bien des traces possibles de l'automate spécifiant le protocole. Un second objectif serait le développement d'un DSL de plus haut niveau pour décrire les scénarios. Enfin il serait intéressant de rejouer les traces fournies par un model checker.

Références

[1] Lăcrămioara Astefănoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci Piergiovanni, and Eugen Zalinescu. Tenderbake - A Solution to Dynamic Repeated Consensus for Blockchains. In *Fourth International Symposium on Foundations and Applications of Blockchain*, 2021.

[2] Shehar Bano, Alberto Sonnino, Andrey Chursin, Dmitri Perelman, and Dahlia Malkhi. Twins : White-Glove Approach for BFT Testing. *arXiv preprint arXiv :2004.10617*, 2020.

[3] Sylvain Conchon, Alexandrina Korneva, Çagdas Bozman, Mohamed Iguernlala, and Alain Mebsout. Formally Documenting Tenderbake. In *3rd Workshop on Formal Methods for Blockchains (FMBC 2021)*, 2021.

[4] LM Goodman. Tezos—a self-amending crypto-ledger White paper. *URL : https ://www.tezos.com/static/papers/white paper.pdf*, 2014.

[5] Jae Kwon and Ethan Buchman. Cosmos whitepaper, 2019.

Soyez prudent : prenez des photos pour l'assurance avec `osnap`

Valentin Chaboche, Zaynah Dargaye et Arvid Jakobsson

Nomadic Labs, Paris, France `prénom.nom@nomadic-labs.com`

Résumé

Comment s'assurer que nous n'introduisons pas de bogue durant l'évolution d'un logiciel ? Les outils de "snapshot testing" offrent une solution : le résultat d'une fonction est capturé et après modification comme de la refactorisation de code, l'outil détectera des changements. Néanmoins, le snapshot testing oblige le développeur à écrire des scénarios à la main. Nous introduisons `osnap` : une bibliothèque de snapshot testing avec une génération aléatoire de scénarios, d'exécution et de détection de changement de comportement – inspirée par la génération aléatoire de bibliothèque comme QuickCheck en Haskell. En utilisant `osnap`, les développeurs peuvent sans effort générer massivement des tests de régression.

1 Introduction

Les tests de régressions sont un ensemble de tests d'un logiciel préalablement développé, qui après modification, s'assure que les modifications n'introduisent pas des défauts dans le programme. Le snapshot testing est une technique particulière de test de régression : les snapshots sont des artéfacts représentant la capture d'événements ou comportements de programmes. Sa particularité réside dans l'exécution d'une comparaison entre les snapshots avant et après modifications.

Il existe en OCaml la bibliothèque `ppx_expect` [6] qui nous permet d'ajouter directement dans le code le résultat textuel attendu d'un test. Lors de son exécution, une différence textuelle est appliquée sur l'actuel résultat et celui attendu. En revanche, chaque test doit être généré en fournissant des instances unitaires des entrées du programme. Les tests pourront difficilement couvrir tous les chemins d'exécution d'un programme, et souffriront d'un biais de la part du testeur dans le choix des entrées.

`QCheck` [4] est une bibliothèque OCaml de test à base de propriété qui propose une approche différente de test. Elle permet de tester des propriétés sur des programmes face à de nombreuses entrées générées aléatoirement. Elle permet de valider des propriétés sur des programmes en évitant donc le biais dans la sélection des entrées à contrario du test unitaire.

Nous souhaitons alors nous inspirer de la gestion de snapshot de `ppx_expect`, mais en réduisant le biais et en ayant une meilleure couverture de code grâce aux générateurs de `QCheck`. Nous pourrons alors générer un grand nombre de snapshots pour un programme de référence, qui deviendront alors nos tests de régression pour les modifications de ce programme. Ce mécanisme de test nous rappelle la bibliothèque Monolith [5] générant aléatoirement des scénarios afin de confronter une bibliothèque de référence et candidate. Cependant, nous souhaitons ici pouvoir appliquer les tests de régression à un niveau plus fin qu'une bibliothèque : des fonctions OCaml.

2 Présentation des fonctionnalités d'osnap

```
let rec expo x =
  function | 0 -> 1 | 1 -> x | n -> expo x (n-1) * x
in
let spec = Osnap.Spec.(int ^> int ^>> Result.int) in
Test.make ~spec ~count:5 ~name:"expo" expo
```

(a) L'exponentiation, sa spécification et son test osnap

```
{ "name": "expo", "scenarios": [
    expo 2 3 = 8
    expo 5 3 = 125
    expo 5 1 = 5
    expo 1 0 = 1 ] }
```

(b) Snapshot de l'exponentiation [1]

`osnap` [3] prend en entrée une spécification de signature d'une fonction, celle de la fonction à tester, cette spécification servira à la génération aléatoire de snapshot. Un snapshot est la combinaison de scénarios aléatoires appliqués à une fonction et de leurs résultats. Ces snapshots sont enregistrés sur disque afin de devenir des tests de régression.

Les spécifications d'`osnap` sont décrites grâce à un ensemble de combinateurs. Une signature pour une fonction $a_0 \to a_1 \to ... \to a_n \to b$ est composée des générateurs pour les types allant de a_0 à a_n via la bibliothèque `QCheck`. Finalement, nous voulons être capable d'observer les différences dans le résultat des différents scénarios. Il faudra alors fournir un afficheur pour le type b, afin d'appliquer une différence textuelle à la manière de `ppx_expect`.

La spécification de ces fonctions nous permet alors de générer des valeurs pour chaque paramètre d'une fonction et donc de les appliquer sur cette même fonction. Nous pouvons alors, via un test interactif fourni par `osnap`, générer `k` scénarios, que nous allons enregistrer sur disque afin de créer un snapshot, grâce à la bibliothèque `Marshal` [2].

Désormais, les tests et snapshots peuvent être versionnés et intégrés dans la suite de test. Pour chaque exécution de cette suite de test, les scénarios vont être récupérés et réappliqués sur les nouvelles versions des fonctions et vont chercher textuellement des régressions dans les retours de fonctions. Les différences observées devront alors demander une analyse du développeur pour déterminer si ces changements sont souhaités ou signifient l'introduction d'une régression.

`osnap` nous permet alors de rapidement générer un grand nombre de cas de test aléatoire via la génération de `QCheck`, contrairement aux cas unitaires de `ppx_expect`. En revanche, nous ne pouvons pas assurer des propriétés lors des changements, mais seulement détecter les éventuelles régressions dans les résultats. Cependant, nous pouvons limiter les entrées respectant des propriétés via l'écriture des générateurs. D'autre part, l'aspect statique de ces tests rend difficile les adaptations lors d'évolutions des signatures de fonctions, étroitement liées aux spécifications nécessaires à `osnap`.

Enfin, bien que la génération aléatoire des scénarios réduise le biais dans les choix d'entrée des tests unitaires, ce biais peut être introduit dans notre cas par les générateurs eux-mêmes. Nous souhaiterions alors incrémentalement améliorer la couverture du code des scénarios, afin de ne pas enregistrer en mémoire des snapshots suivant les mêmes chemins d'exécution. Ceci pourrait être fait en utilisant des outils de couverture comme `bisect_ppx` [1], et ainsi être capable de détecter des générateurs biaisés.

1. La sortie a été rendu lisible pour la compréhension

3 Conclusion

L'objectif d'`osnap` est de gagner en confiance lors des modifications d'une base de code comme des refactorisations. Les tests de régression comme technique de test permettent de détecter des changements non désirés lors de modifications. Cependant, ils nécessitent une écriture manuelle des scénarios par un développeur, et sont sujets à des biais dans l'écriture de ces derniers. Nous généralisons avec `osnap` le processus d'écriture de test de régression et snapshot testing avec la génération aléatoire de valeurs inspirée des bibliothèques de test à base de propriété comme `QCheck`.

Références

[1] Code coverage for OCaml and ReScript. URL : https://github.com/aantron/bisect_ppx.

[2] Marshaling of data structures. URL : https://ocaml.org/api/Marshal.html.

[3] Valentin Chaboche. Random snapshot testing library for OCaml. URL : https://github.com/vch9/osnap.

[4] Simon Cruanes et Rudi Grinberg et Jacques-Pascal Deplaix et Jan Midtgaard. QuickCheck inspired property-based testing for OCaml. URL : https://github.com/c-cube/qcheck.

[5] François Pottier. Strong Automated Testing of OCaml Libraries, 2021. URL : https://hal.inria.fr/hal-03049511/document.

[6] Jane Street. Expect-test - a cram like framework for OCaml. URL : https://github.com/janestreet/ppx_expect.

Mikino: Induction for Dummies

Adrien Champion, Steven de Oliveira, and Keryan Didier

OCamlPro,
name.last-name@ocamlpro.com

Abstract

Mikino is a simple induction engine over transition systems. It is written in Rust, with a strong focus on ergonomics and user-friendliness. It is accompanied by a detailed tutorial which introduces basic notions such as logical operators and SMT solvers. Mikino and its companion tutorial target developers with little to no experience with verification in general and induction in particular. This work is an attempt to educate developers on what a *proof by induction* is, why it is relevant for program verification, why it can fail to (dis)prove a candidate invariant, and what to do in this case.

1 Introduction

The ambition of this work is to present SMT-based induction to developers with no background in formal verification. We hope to achieve this ambition thanks to our hands-on, novice-friendly tutorial and the efforts invested in making mikino as user-friendly and ergonomic as possible. Note that mikino is available both as a binary[1] and a library[2], which we think can motivate enthusiastic readers to try to build their own analyses.

Mikino is released under MIT/Apache 2.0 (library and binary), while the tutorial itself is under *CC BY-SA* (Creative Commons Attribution-ShareAlike). We hope this encourages teachers/trainers to use this work in relevant classes/training sessions.

The following sections discuss mikino's internals and ambitions. Appendix A goes over the process of installing mikino, and Appendix B illustrates mikino's input and output on three examples. We highlight perspectives for mikino in Appendix C.

2 Declarative Transition Systems

Mikino is essentially a much simpler version of the internal proof engine found, for instance, in the KIND 2 [2] model-checker. It analyzes *declarative transition systems*; such systems have a vector of typed state variables $\bar{v}$ that encapsulate the whole state of the system, *i.e.* no information relevant to the system exists outside of $\bar{v}$. For instance, $\bar{v}_c = [\texttt{count} : \texttt{int}, \texttt{reset} : \texttt{bool}]$. A state $\bar{s}$ is a valuation of the state variables, such as $\bar{s} = [7, \texttt{false}]$.

The *initial state(s)* of the system are specified by a *state predicate* $\texttt{init}(\bar{v})$: a formula over $\bar{v}$. Any state $\bar{s}$ such that $\texttt{init}(\bar{s})$ evaluates to $\texttt{true}$ is a legal initial state for the system. For instance, $\texttt{init}(\bar{v}_c) \equiv \texttt{count} \geq 0 \wedge (\texttt{reset} \Rightarrow \texttt{count} = 0)$. The *transition relation* $\texttt{trans}(\bar{v}, \bar{v}')$ is a formula over unprimed (*current*) and primed (*next*) state variables. State $\bar{s}'$ is a legal successor of $\bar{s}$ for the system if and only if $\texttt{trans}(\bar{s}', \bar{s})$ evaluates to $\texttt{true}$. For instance,

$$\texttt{trans}(\bar{v}_c, \bar{v}'_c) \quad \equiv \quad \texttt{count}' \;=\; \textbf{if } \texttt{reset}' \textbf{ then } \{0\} \textbf{ else } \{\texttt{count} + 1\}$$

where the *next* value of $\texttt{count}$ is 0 if $\texttt{reset}'$, and its old value plus one otherwise. The next value of $\texttt{reset}$ is not constrained at all.

[1] https://github.com/OCamlPro/mikino_bin
[2] https://github.com/OCamlPro/mikino

Last, mikino expects some *candidate properties* or *candidate invariants* which are just called *candidates*. A candidate is a predicate over $\bar{v}$. These candidates are named and mikino uses the name provided when producing feedback. For instance,$\texttt{candidate}_1(\bar{v}_c) \equiv \texttt{cnt} \geq 0$ could be a candidate for our running example.

3 Analysis

Mikino's analysis relies heavily on Smt solvers [4], and more precisely on Z3 [3]. A *proof by induction* over a transition system $(\bar{v}, \texttt{init}, \texttt{trans})$ of a candidate `candidate` consists in showing two things:

- $\texttt{init}(\bar{v}) \wedge \neg\texttt{candidate}(\bar{v})$ is unsat,
  thus proving that $\texttt{init}(\bar{v}) \Rightarrow \texttt{candidate}(\bar{v})$;
- $\texttt{candidate}(\bar{v}) \wedge \texttt{trans}(\bar{v}, \bar{v}') \wedge \neg\texttt{candidate}(\bar{v}')$ is unsat,
  thus proving that $\texttt{candidate}(\bar{v}) \wedge \texttt{trans}(\bar{v}, \bar{v}') \Rightarrow \texttt{candidate}(\bar{v}')$.

If both these formulas are unsat, then `candidate` is an invariant for the system by induction.

Now, if the `init` check fails, we can extract a model from the solver. This model, by construction, falsifies `candidate`: the candidate can be falsified by at least one of the initial states. This is a concrete counterexamples as discussed previously.

If the `trans` check fails however, it only means that `candidate` is not preserved by the transition relation. This result says nothing on whether `trans` can reach a state falsifying `candidate`. The best we can do is to extract a model, which in this case corresponds to a pair of states $(\bar{s}, \bar{s}')$ such that *i)* $\bar{s}$ verifies the candidate, *ii)* $\bar{s}'$ is a legal successor of $\bar{s}$, and *iii)* $\bar{s}'$ falsifies the candidate.

One of the main goals of the mikino tutorial is to bring its intended audience of verification-agnostic developers to fully understand what it means for a `trans` check to fail, and what the model extracted corresponds to. Mikino itself goes to great length to have as readable and user-friendly an output as possible, hopefully making things easier for readers going through the tutorial.

The second main goal of the tutorial is to teach readers how to react to a failed `trans` check. *Property strengthening* —or *candidate strengthening*, here— consists in adding *information* to a candidate when it is not inductive. This consists in adding new candidates that act as lemmas so that the original candidate *in conjunction with the lemmas* is inductive. The pair of succeeding states returned by failed `trans` checks is helpful in deciding what lemma should be added, as long as users have an understanding of what these states represent.

Strengthening can only succeed if the candidate is indeed an invariant (although a non-inductive one) for the system. Bugs happen however, and mikino can find them thanks to its Bounded Model-Checking (Bmc) feature. We do not discuss Bmc further here, but note that one of the mikino runs shown in Appendix B discusses Bmc and uses it to find counterexamples.

4 Conclusion

Mikino was created for verification novices, typically average developers, and is thus focused on user-friendliness and ergonomics. Its companion tutorial introduces Smt solvers, declarative transition systems, Bmc, induction, and candidate strengthening using simple terms and examples that readers can run locally, modify, and rewrite. Mikino is open source and available both as a binary and a library, which encourages experimentation beyond the relatively small scope of its companion tutorial.

References

[1] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, 2010.

[2] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The kind 2 model checker. In *CAV (2)*, volume 9780 of *Lecture Notes in Computer Science*, pages 510–517. Springer, 2016.

[3] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[4] Cesare Tinelli. Foundations of satisfiability modulo theories. In *WoLLIC*, volume 6188 of *Lecture Notes in Computer Science*, page 58. Springer, 2010.

A Installing Mikino

Mikino requires the Z3 SMT solver to run. In practice, this means retrieving a Z3 binary as discussed in the tutorial:

https://ocamlpro.github.io/verification_for_dummies/smt/index.html#z3

The easiest way to do this is on Z3's release page:

https://github.com/Z3Prover/z3/releases

By default, mikino assumes the Z3 binary is in your path and called `z3`. You can use mikino's `--z3_cmd` to specify a different name or path. Refer to `mikino help` for details.
We recommend you refer to tutorial's instructions for installing mikino:

https://ocamlpro.github.io/verification_for_dummies/mikino_bmc/index.html

which, unlike what follows, are kept updated. The easiest way to obtain the latest mikino binary is to go to its release page:

https://github.com/OCamlPro/mikino_bin/releases

At the time of writing, the latest version is `0.5.2`.

Alternatively, if Rust (https://www.rust-lang.org) is installed on your machine, you can run `cargo install mikino` which will compile mikino and put it in you path. In case you want to update to the latest version, run `cargo install --force mikino`.

B Examples: Proof Failure/Success and Bmc

Let us go back to the example system used in the main body of this article. It is defined by its state variables $\bar{v}$, its initial predicate `init`, and its transition relation `trans`. We also define two candidates $\texttt{candidate}_1$ and $\texttt{candidate}_2$:

$$\begin{aligned}
\bar{v} &\equiv [\texttt{count} : \texttt{int}, \texttt{reset} : \texttt{bool}] \\
\texttt{init}(\bar{v}) &\equiv \texttt{count} \geq 0 \wedge (\texttt{reset} \Rightarrow \texttt{count} = 0) \\
\texttt{trans}(\bar{v}, \bar{v}') &\equiv \texttt{count}' = \textbf{if } \texttt{reset}' \textbf{ then } \{0\} \textbf{ else } \{\texttt{count} + 1\} \\
\texttt{candidate}_1(\bar{v}) &\equiv \neg(\texttt{count} = -7) \\
\texttt{candidate}_2(\bar{v}) &\equiv \texttt{reset} \Rightarrow \texttt{count} = 0
\end{aligned}$$

The equivalent in mikino's input format is

```
1  svars {
2      count: int,
3      reset: bool,
4  }
5  init {
6      count ≥ 0,
7      reset ⇒ (count = 0),
8  }
9  trans {
10     'count = if 'reset { 0 } else { count + 1 },
11 }
12 candidates {
13     "candidate 1": ¬(count = -7),
14     "candidate 2": reset ⇒ (count = 0),
15 }
```

Note that:

- *primed* variables have their prime *before* the identifier, not after. This is because we want to use Rust syntax highlighting: in Rust, `'<ident>` is the syntax for *lifetimes* which make `'count` and `'reset` pop out. On the other hand, `<ident>'` would be interpreted by syntax highlighting as an identifier followed by the start of a character literal.
- mikino's Rust expressions are more readable than SMT-LIB's S-expressions.
- mikino supports various UTF-8 operators in addition to their usual ASCII equivalent(s) `>=`, `=>`, `&&`, `||`, `!`, *etc.*
- the `init` and `trans` sections take a list of comma-separated expressions (with optional trailing comma), understood as a conjunction.

Running mikino on this systems yields the following. (Run `mikino help` for details on mikino's CLI.)

```
❯ mikino check rsc/stopwatch.mkn
checking base case...
success: all candidate(s) hold in the base state

checking step case...
failed: the following candidate(s) are not inductive:
- `candidate 1` = (not (= count (- 7)))
  |=| Step k
  | count = (- 8)
  | reset = false
  |=| Step k + 1
  | count = (- 7)
  | reset = false
  |=|

|===| Induction attempt result
| - all candidates hold in the initial state(s)
|
| - the following candidate(s) are not inductive (not preserved by the transition relation)
|   `candidate 1`
|
| - system might be unsafe, some candidate(s) are not inductive
|
| - the following candidate(s) hold in the initial state(s) and are inductive
|   and thus hold in all reachable states of the system:
|   `candidate 2`
|===|
```

The first candidate is not inductive and needs some strengthening. Let us just add the very natural lemma `count` ≥ 0.

```
1  svars {
2      count: int,
3      reset: bool,
4  }
5  init {
6      count ≥ 0,
7      reset ⇒ (count = 0),
8  }
9  trans {
10     'count = if 'reset { 0 } else { count + 1 },
11 }
12 candidates {
13     "candidate 1": ¬(count = -7),
14     "candidate 2": reset ⇒ (count = 0),
15     "lemma": count ≥ 0,
16 }
```

Running mikino again, we see that the strengthening was successful and mikino is able to prove all three candidates.

```
❯ mikino check rsc/stopwatch_2.mkn
checking base case...
success: all candidate(s) hold in the base state

checking step case...
success: all candidate(s) are inductive

|===| Induction attempt result
| - all candidates hold in the initial state(s)
|
| - all candidates are inductive (preserved by the transition relation)
|
| - system is safe, all reachable states verify the candidate(s)
|===|
```

Last, here is an example of asking mikino to check a falsifiable candidate by Bmc. Roughly, Bmc is an iterative process that starts by looking for a falsification of the candidate(s) in the initial state, exactly like induction's `init` check. If none exists, Bmc asks the same questions about the successors of the initial states. More precisely, is

$$\texttt{init}(\bar{v}_0) \wedge \texttt{trans}(\bar{v}_0, \bar{v}_1) \wedge \neg\texttt{candidate}(\bar{v}_1)$$

satisfiable? If it is, Bmc can extract a model corresponding to two succeeding states leading to a falsification of the candidate. If not, Bmc *unrolls* the transition relation again to check the successors of the successors of the initial states.
To showcase Bmc in mikino, let us modify `init` slightly and give ourselves a falsifiable candidate.

```
1  svars {
2      count: int,
3      reset: bool,
4  }
5  init {
6      count = 0,
7  }
8  trans {
9      'count = if 'reset { 0 } else { count + 1 },
10 }
11 candidates {
12     "candidate 1": count ≥ 0,
13     "candidate 2": reset ⇒ (count = 0),
14     "falsifiable": ¬(count = 5),
15 }
```

In `check` mode, Bmc is activated by passing the `--bmc` option to mikino: `mikino check --bmc <file>`. This option makes mikino run Bmc on all non-inductive candidates. The second image only shows the relevant part of the output, after the `trans` check counterexample is reported.

```
|===| Induction attempt result
| - all candidates hold in the initial state(s)
|
| - the following candidate(s) are not inductive (not preserved by the transition relation)
|   `falsifiable`
|
| - system might be unsafe, some candidate(s) are not inductive
|
| - the following candidate(s) hold in the initial state(s) and are inductive
|   and thus hold in all reachable states of the system:
|   `candidate 1`
|   `candidate 2`
|===|

running BMC, looking for falsifications for 1 candidate(s)...
checking for falsifications at depth 0
checking for falsifications at depth 1
checking for falsifications at depth 2
checking for falsifications at depth 3
checking for falsifications at depth 4
checking for falsifications at depth 5
found a falsification at depth 5:
- `falsifiable` = (not (= count 5))
  |=| Step 0
  | count = 0
  | reset = false
  |=| Step 1
  | count = 1
  | reset = false
  |=| Step 2
  | count = 2
  | reset = false
  |=| Step 3
  | count = 3
  | reset = false
  |=| Step 4
  | count = 4
  | reset = false
  |=| Step 5
  | count = 5
  | reset = false
  |=|

|===| Bmc result
| - found a falsification for the following candidate(s)
|   `falsifiable`
|
| - system is unsafe
|===|
```

C Perspectives

Since the first chapters of the tutorial go over Smt checks, we rely on the Smt-Lib standard [1] to write the examples. Smt-Lib is based on S-expressions as a middle ground between human-readability and ease of parsing. While very appropriate as a common input language for Smt solvers and for writing/sharing benchmarks (its main purposes), it can be deterring for novices just starting out: $x + 1$ for instance would be written `(+ x 1)`. Mikino's syntax for transition systems is based on Rust and is much more natural to uninitiated readers.

Depending on the feedback on mikino and its tutorial, we would like mikino to be able to act as a (thin, potentially interactive) frontend for Smt solvers by accepting Rust-style expressions. This would remove the need for showing Smt-Lib code at all and, we think, make it easier for novices to dive into formal verification.

Trakt : Uniformiser les types pour automatiser les preuves

Denis Cousineau[1], Enzo Crance[1,2], and Assia Mahboubi[2]

[1] Mitsubishi Electric R&D Centre Europe (MERCE), Rennes, France
[2] Inria Rennes Bretagne Atlantique, Rennes, France

Résumé

Dans un assistant de preuve comme Coq, un même objet mathématique peut souvent être formalisé par différentes structures de données. Par exemple, le type `Z` des entiers binaires, dans la bibliothèque standard de Coq, représente les entiers relatifs tout comme le type `ssrint`, des entiers unaires, fourni par la bibliothèque MathComp. En pratique, cette situation familière en programmation est un frein à la preuve formelle automatique. Dans cet article, nous présentons `trakt`, un outil dont l'objectif est de faciliter l'accès des utilisateurs de Coq aux tactiques d'automatisation, pour la représentation des théories décidables de leur choix. Cet outil construit une formule auxiliaire à partir d'un but utilisateur, et une preuve que cette dernière implique ce but initial. La formule auxiliaire est conçue pour être adaptée aux outils de preuve automatique (`lia`, SMTCoq, etc). Cet outil est extensible, grâce à une API permettant à l'utilisateur de définir plusieurs natures de plongements dans un jeu de structures de données de référence. Le méta-langage Coq-Elpi, utilisé pour l'implémentation, fournit des facilités bienvenues pour la gestion des lieurs et la mise en œuvre des parcours de termes en jeu dans ces tactiques.

1 Introduction

L'automatisation est un critère majeur dans l'adoption des assistants de preuve, que ce soit pour des applications académiques ou industrielles. Par exemple, les utilisateurs de l'assistant de preuve Isabelle/HOL [1] bénéficient de tactiques d'automatisation puissantes, en particulier celles basées sur le modèle de *marteaux* [2]. En Coq, des procédures de décision variées sont disponibles. Certaines travaillent sur des théories décidables spécifiques, comme la tactique `lia` [3] pour l'arithmétique linéaire entière. D'autres sont conçues pour travailler sur une combinaison de théories, comme la bibliothèque SMTCoq [4]. L'objectif de cette dernière est de connecter Coq à des solveurs SMT, et d'apporter ainsi aux utilisateurs de Coq la puissance de preuve de ces outils optimisés. Un des défis de l'automatisation en Coq est de dompter l'expressivité de Gallina, le langage de Coq, et de rendre les outils de preuve automatique disponibles à toutes les structures de données pouvant représenter une même théorie décidable.

Pour répondre à ce problème, nous proposons un outil de pré-traitement des buts Coq, qui intervient avant les tactiques d'automatisation. Il reformule le but de manière certifiée afin d'en donner une forme canonisée, par exemple adaptée à la spécification d'entrée d'une tactique d'automatisation arbitraire. L'outil se veut simple d'utilisation et extensible, exposant une API pour que l'utilisateur décrive précisément la cible attendue de la traduction.

2 Limites des interfaces des tactiques d'automatisation

L'expressivité offerte par Coq permet aux utilisateurs de représenter un même concept par des types possiblement très différents. Par exemple, la bibliothèque standard de Coq elle-même propose plusieurs types pour représenter les entiers : naturel, relatifs, binaires, machine, etc. Un utilisateur peut aussi choisir d'introduire sa propre formalisation des entiers, et y associer de nouvelles opérations, propriétés, relations, et appareils de notations. Par ailleurs, les développeurs des outils de preuve formelle automatique peuvent également être amenés à faire

des choix sur la forme syntaxique des buts qu'ils souhaitent traiter. Cette double variabilité, à la fois des spécifications d'entrée des tactiques et des buts énoncés par les utilisateurs, peut entraver le bon fonctionnement des tactiques d'automatisation, qui ne parviennent pas toujours à traiter tous les buts Coq attendus, même ceux qui représentent a priori des formules de la théorie décidée par l'algorithme sous-jacent.

Afin d'augmenter la compatibilité des buts avec les procédures de décision, une solution est d'ajouter une phase de pré-traitement (un *proxy*) entre ces buts et les tactiques associées. Par exemple, la tactique `zify` [5] a pour objectif de faire converger une variété de buts arithmétiques et logiques vers une forme canonique de ces buts compréhensible par la tactique `lia`. La tactique `zify` traduit certaines représentations des entiers (`nat`, `positive`, etc) vers `Z`, la représentation des entiers de la tactique ciblée. La traduction est certifiée, la forme canonique étant prouvée équivalente au but d'origine. De plus, l'outil est extensible car il met à disposition des classes de types Coq pour que l'utilisateur puisse déclarer des plongements depuis d'autres types, ainsi que la traduction de symboles et relations associés. La tactique exploite ensuite toutes les instances déclarées. Une autre tactique de pré-traitement est `mczify` [6]. Il s'agit d'un jeu d'instances des classes de types de `zify` conçues spécialement pour traduire un maximum de symboles de la bibliothèque MathComp [7].

L'objectif de ce travail est de généraliser ces interfaces, de sorte qu'elles puissent être exploitées par des outils d'automatisation arbitraires. La bibliothèque SMTCoq dont les tactiques d'automatisation ciblent les théories SMT (arithmétique, égalité, vecteurs de bits, symboles non interprétés, etc) sert ici d'exemple et de motivation. Celle-ci implémente en effet une forme de pré-traitement des buts, mais relativement sommaire et peu extensible. À notre connaissance, il n'existe pas de proxy à la fois extensible *via* des déclarations utilisateur, et générique quant aux tactiques d'automatisation ciblées.

3 Un proxy générique et extensible

L'interface de l'outil que nous proposons, `trakt`[1], permet d'effectuer des déclarations à la `zify`, mais pour des types cibles arbitraires. Par ailleurs, cette interface est complètement générique : elle n'est liée à aucune bibliothèque spécifique et ne cible aucune tactique d'automatisation en particulier. L'outil `trakt` propose ainsi des commandes Coq permettant à l'utilisateur de déclarer facilement des plongements entre types, des symboles connus ou bien des relations, alimentant une base de données. Celle-ci est ensuite exploitée par une tactique de traduction, `trakt`, dont le comportement est paramétré par l'utilisateur grâce à ses déclarations. La traduction est ensuite entièrement automatique. À partir d'un but Coq `G`, elle génère à la fois un nouveau but `G'`, la forme canonisée de `G`, et un terme de preuve de type `G' -> G`.

Cet outil est écrit en Coq-Elpi [8], un méta-langage pour Coq dans le paradigme de la programmation logique. Ce choix de méta-langage s'est révélé pertinent à plusieurs titres. Par exemple, l'encodage des termes Coq en HOAS (*Higher Order Abstract Syntax*) [9] permet de traiter les lieurs sans devoir gérer des indices de De Bruijn [10], et de créer temporairement de nouveaux constructeurs de termes Coq, afin d'annoter librement des nœuds. L'association des variables d'unification de Coq à des variables Prolog permet de forger facilement des termes à trous. Enfin, le niveau d'abstraction offert permet d'éviter d'expliciter tous les détails de l'arbre de syntaxe abstraite lors du traitement d'un terme.

Notre prototype a été testé avec succès sur des exemples de buts variés. Lors de l'exposé, nous en ferons une démonstration en insistant sur la facilité de prise en main pour l'utilisateur et l'augmentation effective du niveau d'automatisation dans Coq.

1. "entonnoir" en norvégien

Références

[1] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL : a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[2] Jasmin Christian Blanchette, Cezary Kaliszyk, Lawrence C Paulson, and Josef Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1) :101–148, 2016.

[3] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *International Workshop on Types for Proofs and Programs*, pages 48–62. Springer, 2006.

[4] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark Barrett. SMTCoq : A plug-in for integrating SMT solvers into Coq. In *International Conference on Computer Aided Verification*, pages 126–133. Springer, 2017.

[5] Frédéric Besson. ppsimpl : a reflexive Coq tactic for canonising goals, 2017.

[6] Kazuhiko Sakaguchi. `mczify`. https://github.com/math-comp/mczify.

[7] Assia Mahboubi and Enrico Tassi. Mathematical components, 2017.

[8] Enrico Tassi. Elpi : an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). 2018.

[9] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. *ACM sigplan notices*, 23(7) :199–208, 1988.

[10] Nicolaas Govert De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

Catala, un langage pour transformer la loi en code

Alain Delaët[1,2] and Denis Merigoux[1]

[1] Inria Paris
[2] ENS de Lyon

Résumé

Le droit codifie et régit de nombreux aspects de la vie quotidienne. Si la plupart du temps les lois sont sujettes à interprétation, dans certains domaines la loi ne permet pas d'interprétation et vise essentiellement à décrire rigoureusement un calcul ou une procédure de décision, c'est-à-dire un algorithme.

Catala est un nouveau langage conçu en collaboration avec des juristes qui permet une compilation depuis une spécification proche de la loi vers un code exécutable. Contrairement aux langages traditionnels, Catala est adapté aux raisonnements cas de base/exceptions présents dans la loi ; il intègre de la programmation littéraire du droit qui facilite les mises à jour du programme. Finalement, Catala compile vers une variété de langages cibles dont OCaml.

Dans cette démonstration de prototype, nous montrerons comment exprimer une partie du calcul des allocations familiales dans Catala.

Contexte. Dans certains domaines juridiques, les textes décrivent comment le calcul d'une quantité doit être effectué. C'est par exemple le cas pour le droit fiscal et le droit des prestations familiales. L'administration doit cependant se charger de l'implémentation effective du processus, c'est-à-dire de transformer la loi en code. La méthode actuelle se décompose en trois étapes [4]. La première correspond à l'écriture de la loi, sous forme exclusivement littéraire, qui est assurée par le législateur. Ensuite, le service juridique de la direction générale des finances publiques doit interpréter le texte, en effectuant des recherches juridiques au besoin, pour en extraire une spécification. Finalement, la construction du logiciel mettant en œuvre cette spécification est soit assurée par le service informatique de l'administration, soit sous-traitée à des entreprises externes, et le logiciel résultant est utilisé en production pour le calcul des impôts par exemple.

Cette démarche pose plusieurs problèmes. Premièrement, la correction fonctionnelle – le fait que le calcul effectué correspond bien à sa spécification juridique – est vérifiée uniquement sur quelques exemples de tests [1]. Ces "cas pratiques" sont rédigés par des juristes qui n'ont pas forcément le temps ni la méthodologie pour tester tous les cas possibles, si bien que certains cas peuvent arriver en production sans avoir été totalement testés. Deuxièmement, lors d'un changement législatif, l'ensemble du processus doit être repris, car une petite modification dans la législation peut changer en profondeur la façon dont le calcul peut être réalisé dans un langage traditionnel. Par exemple, le gouvernement a invoqué des difficultés d'implémentation qui empêcheraient de déconjugaliser l'allocation adultes handicapés [5].

Ces besoins de fiabilité et d'incrémentalité sont des problèmes étudiés en informatique, et les méthodes formelles y apportent partiellement des solutions. Ainsi, la vérification de programme a déjà fait ses preuves pour les programmes critiques.

Introduction d'un nouveau DSL Nous proposons une nouvelle méthode de travail qui s'appuie sur un nouveau langage de programmation dédié, Catala, introduit dans [3]. Le langage a été conçu en collaboration avec des juristes dans le but d'exprimer des spécifications dans un style ressemblant à celui des articles de loi qui servent de référence au programme.

Les fonctionnalités de Catala visent à permettre une relecture du code de la part de professionnels du droit. Par exemple, la syntaxe de Catala est très verbeuse, ce qui facile sa lecture par les juristes. Son *parser* dispose de plusieurs *lexers* pour faciliter le support d'une nouvelle langue : français, anglais et polonais sont actuellement disponibles. Finalement, un programme Catala est composé de code et de prose, utilisant de la programmation littéraire. Les juristes peuvent directement réfléchir sur le code Catala plutôt que sur des cas pratiques, de la même façon qu'un programmeur raisonne sur la sémantique de son programme plutôt que sur des tests.

Pour écrire de nouveaux programmes en Catala, l'approche que nous avons choisie a été de faire de la programmation par binômes pluridisciplinaires. Le travail de l'informaticienne est de repérer les ambiguïtés et de demander au juriste de les résoudre par un raisonnement juridique. Le juriste peut, grâce à la lisibilité du code, vérifier que le code écrit correspond bien à l'esprit et à la lettre de la loi.

La programmation littéraire, en plus de pouvoir générer des documents compatibles avec les habitudes des juristes, relie article par article la loi avec le code Catala. Cela permet d'effectuer localement les modifications rendues nécessaires par un amendement de la loi. Cette approche est garante de l'incrémentalité mentionnée ci-dessus.

Finalement, l'ensemble du compilateur est Open Source et implémenté en OCaml. Conformément aux dispositions législatives sur la publication des codes sources comme documents administratifs, il est également nécessaire que les programmes Catala utilisés par d'éventuelles administrations soient également Open Source.

Caractéristiques du langage Formellement, Catala s'appuie sur la logique par défaut, formalisée en 1980 par [6]. Cette logique sous-tend la structure de la loi, comme montrée en 2018 par [2]. La représentation intermédiaire centrale de Catala est un λ-calcul augmenté d'un terme modélisant une structure cas de base/exceptions, conformément à la logique par défaut.

L'autre spécificité de Catala est sa chaine de compilation qui permet de compiler le code Catala vers plusieurs langages cibles : Python, OCaml et JavaScript (via js_of_ocaml). Ce schéma de compilation permet une interopérabilité virtuellement illimitée, avec par exemple des langages ne disposant même pas de FFI, ou des architectures pré-API.

Pour une description complète du langage, se reporter à [3].

Contenu de la démonstration Dans cette démonstration, nous monterons comment exprimer une partie du calcul des allocations familiales dans Catala. Nous exposerons notamment le travail de recherche juridique qui a été nécessaire pour lever les ambiguïtés des formulations. Nous verrons ensuite comment Catala génère du code exécutable, en suivant sa transformation jusqu'à la mise en production.

Références

[1] Liane Huttner and Denis Merigoux. Traduire la loi en code grâce au langage de programmation Catala. *Revue de droit fiscal*, (5) :121, 2021.

[2] Sarah B. Lawsky. A Logic for Statutes. *Florida Tax Review*, 2018.

[3] Denis Merigoux, Nicolas Chataing, and Jonathan Protzenko. Catala : A programming language for the law. *Proc. ACM Program. Lang.*, 5(ICFP), August 2021.

[4] Denis Merigoux, Raphaël Monat, and Jonathan Protzenko. A modern compiler for the french tax code. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 71–82, New York, NY, USA, 2021. Association for Computing Machinery.

[5] Assemblée nationale. Débats sur la proposition de loi portant diverses mesures de justice sociale. https://www.assemblee-nationale.fr/dyn/15/comptes-rendus/seance/session-ordinaire-de-2020-2021/premiere-seance-du-jeudi-17-juin-2021#2557908, 2021.

[6] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1) :81 – 132, 1980. Special Issue on Non-Monotonic Logic.

Actema : une interface graphique et gestuelle pour preuves formelles

Pablo Donato, Pierre-Yves Strub, and Benjamin Werner

LIX, École Polytechnique, France

Introduction

Le principe des systèmes de preuves interactifs est que l'utilisateur construit incrémentalement une preuve par une boucle d'interaction. On progresse à travers une séquence d'états, chacun correspondant à une preuve incomplète. Chacun de ces états est lui même composé d'un ensemble fini de *buts* ; la preuve est achevée lorsque cet ensemble de buts est vide.

Du point de vue de l'utilisateur, un but apparaît comme un séquent, tel que défini par Gentzen. C'est-à-dire, en logique intuitioniste :

— Une proposition A qu'il s'agit de prouver et qu'on peut appeler la *conclusion* du but ;
— un ensemble de propositions Γ décrivant les *hypothèses*.

L'utilisateur effectue des actions sur un but à la fois. Ces actions transforment le but, ou plus exactement remplacent le but par un ensemble de buts.

Dans le paradigme dominant, ces commandes sont textuelles. Depuis Robin Milner et LCFF [2], elles sont appelées des *tactiques*.

Le prototype que nous présentons propose de remplacer ces commandes textuelles par des actions effectuées sur une interface graphique. En ce sens, c'est une continuation du travail effectué sur le *Proof-by-Pointing* (PbP) initié dans les années 1990 par Gilles Kahn, Yves Bertot, Laurent Théry et leur équipe [1]. Dans les deux cas l'utilisateur effectue des actions sur les éléments *(items)* du but, à savoir sa conclusion et ses hypothèses. Une nouveauté de notre travail est que nous ne nous restreignons pas à des *click* sur des sous-expressions des éléments, mais autorisons d'autres actions, comme le glissé-déposé *(drag-and-drop)* d'un élément sur un autre. Ceci enrichit le langage des actions de manière, nous l'espérons, intuitive. Précisons qu'il ne s'agit pas de remplacer mais de compléter les actions PbP. Il s'agit donc d'aller vers un paradigme plus général de construction de preuves par actions.

Nous avons implémenté un petit prototype pour démontrer et expérimenter cette approche.

Disposition

Les caractéristiques que nous proposons devraient pouvoir être déclinées pour de nombreux formalismes logiques ; le prototype actuel implémente la logique du premier ordre intuitioniste avec support pour l'égalité.

Un avantage du paradigme des preuves-par-actions est qu'il permet une présentation très dépouillée de l'état de preuve ; un but apparaît comme un ensemble d'éléments dont la nature est définie par leur couleurs respectives :

— Un *élément rouge* qui porte la proposition devant être prouvée, c'est-à-dire la *conclusion*,
— des *éléments bleus*, qui sont les *hypothèses* locales.

Chaque but apparaît sur un onglet. Dans ces onglets, les éléments apparaissent donc comme des rectangles bleus ou rouges, munis chacun de la proposition de l'élément. Par défaut, la conclusion rouge est sur la droite et les hypothèses bleues sur la gauche ; voir la figure 1 pour un état possible.

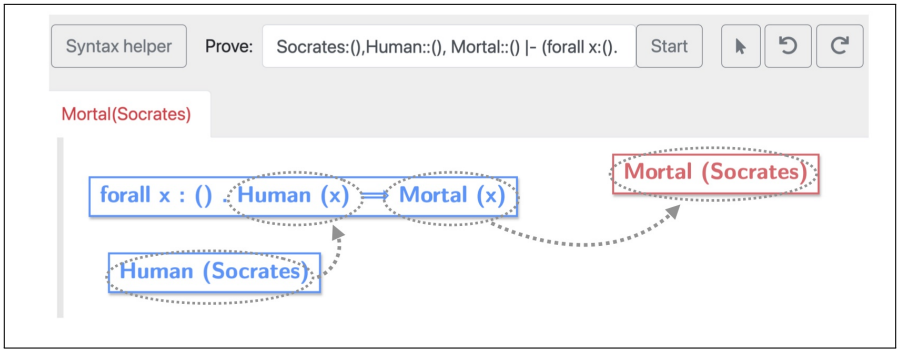


FIGURE 1 – Une capture partielle d'écran montrant le prototype. La conclusion est en rouge à droite, les deux hypothèses en bleu sur la gauche. Les flèches en pointillé ont été rajoutées pour indiquer deux actions de glisser-déposer possibles.

Les éléments sont ce sur quoi l'utilisateur peut agir : soit en cliquant sur l'un d'entre eux, soit en les déplaçant.

L'intuition sous-jacente au glisser-déposer est, nous l'espérons, relativement simple. Elle est basée sur la distinction entre éléments rouges et bleus, et souligne un point simple mais fondamental : même si la conclusion et les hypothèses sont exprimées dans le même langage logique, elles jouent des rôles distincts dans la preuve. Il y a même un symétrie à l'œuvre :

— la conclusion rouge A *attend* une justification de la validité de A ;
— en revanche, une hypothèse bleue B *fournit* une justification de la validité de B dans l'état courant.

Ceci se matérialise par l'instance la plus simple d'action drag-and-drop. Etant donné un but dont la conclusion est une proposition A, si ce but comporte également une hypothèse A, on peut déposer l'un de ces deux éléments sur l'autre, ce qui résout ce but. D'un point de vue logique, il s'agit là d'une utilisation de la règle axiome.

Une grande partie du travail consiste à généraliser cette idée à des situations plus complexes, tout en gardant un comportement intuitif. Sur le plan théorique, cela correspond à généraliser la règle axiome, et est très lié à l'approche d'inférence profonde. Comprendre ceci a permis de mieux définir le comportement du glisser-déposer. Remarquez que souvent, ces actions ne vont pas résoudre le but, mais transformer la conclusion courante. D'autres actions vont combiner deux hypothèses pour en faire apparaître une nouvelle. Dans le cas de la figure 1, il y a deux actions possibles :

— On peut déposer l'hypothèse du haut sur la conclusion, ce qui va transformer en Human(Socrates) (qui peut ensuite être prouvé en utilisant la seconde hypothèse).
— On peut déposer une des deux hypothèses sur l'autre, ce qui va engendrer une nouvelle hypothèse Mortal(Socrates).

Références

[1] Yves Bertot, Gilles Kahn, and Laurent Théry. Proof by pointing. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789, pages 141–160. Springer Berlin Heidelberg, 1994. Series Title : Lecture Notes in Computer Science.

[2] Robin Milner. The use of machines to assist in rigorous proof. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 312(1522) :411–422, 1984.

Démonstration de Steel, une logique de séparation concurrente pour prouver des programmes F⋆

Aymeric Fromherz[1] and Antonin Reitz[2]

[1] Inria Paris
aymeric.fromherz@inria.fr
[2] Inria Paris
antonin.reitz@inria.fr

Abstract

Steel est un framework pour développer et prouver des programmes concurrents écrits en F⋆, un langage de programmation avec types dépendants ainsi qu'un assistant de preuve. Inspiré par Iris, Steel repose sur une logique de séparation concurrente imprédicative qui inclut un modèle mémoire fondé sur des monoïdes commutatifs partiels et permet l'utilisation d'invariants alloués dynamiquement pour raisonner sur des interactions concurrentes sans recourir à des verrous ("locks"). Afin d'offrir une vérification semi-automatique, Steel sépare les obligations de preuves générées en deux parties : une procédure de décision partielle, implémentée à l'aide du moteur de tactiques de F⋆, raisonne sur la logique de séparation tandis qu'un solveur SMT permet de décharger automatiquement des obligations de preuves exprimées en logique du premier ordre, par exemple liées à de l'arithmétique. Dans cette démonstration, nous présentons plusieurs bibliothèques vérifiées qui illustrent l'expressivité et la programmabilité de Steel.

Introduction Steel [10, 1] est une logique de séparation concurrente pour raisonner à propos de programmes F⋆ [9], dévelopée en collaboration étroite avec Microsoft Research. Steel est inspiré par et partage plusieurs similitudes avec Iris [4, 2, 3]: notre logique de séparation concurrente est imprédicative, supporte l'allocation dynamique d'invariants pour modéliser des interactions concurrentes sans verrous ("locks"), et propose un modèle mémoire fondé sur des monoïdes commutatifs partiels, qui permettent d'encoder différents idiomes tels que des permissions fractionnelles ou des références qui évoluent monotoniquement selon un préordre défini par l'utilisatrice. Comparé à Iris, Steel repose sur un plongement de surface ("shallow embedding") d'une logique de séparation en F⋆. Cela permet à Steel d'être applicable directement à F⋆ dans sa globalité, incluant par exemple ses types dépendents et à refinements.

Un Exemple: Swap Pour présenter les spécificités de Steel, nous utilisons un exemple classique, la fonction `swap` en Figure 1, qui échange le contenu de deux références `r1` et `r2`.

Pour spécifier une fonction Steel, nous utilisons un effet [8], `Steel`, indexé par plusieurs éléments. Le premier indice, ici `unit`, correspond au type de retour de la fonction `swap`. Les deux indices suivants, ici `ptr r1 ∗ ptr r2` et $\lambda\_ \rightarrow$ `ptr r1 ∗ ptr r2`, correspondent à une précondition et une postcondition, exprimés à l'aide de notre logique de séparation. Pour `swap`, cette spécification repose sur le prédicat `ptr r`, qui symbolise que la référence `r` est actuellement une référence valide, i.e., elle a été correctement allouée en mémoire, et n'a pas encore été libérée. Cette spécification requiert donc initialement que `r1` et `r2` soient deux références valides, mais également disjointes, tel que signifié par l'utilisation de la conjonction séparatrice $*$ de logique de séparation. La postcondition est similaire, avec une différence: elle peut dépendre de la valeur de retour, et est donc représentée comme une fonction. Pour `swap`, cela n'est pas le cas, et la valeur de retour est donc ignorée (utilisant la syntaxe $\lambda\_ \rightarrow$), mais cette dépendence permet par exemple de spécifier par λ`r` $\rightarrow$ `ptr r` une fonction d'allocation qui retournerait une nouvelle référence `r`.

```
let swap (r1 r2:ref int) : Steel unit
    (ptr r1 ∗ ptr r2) (λ _ → ptr r1 ∗ ptr r2)
    (requires λ_ → ⊤)
    (ensures λs _ s' → s'.[r1] = s.[r2] ∧ s'.[r2] = s.[r1])
  = let x1 = read r1 in
     let x2 = read r2 in
     write r2 x1;
     write r1 x2
```

Figure 1: Un exemple simple de programme Steel: swap. Après exécution, les valeurs stockées en mémoire dans les deux références sont échangées.

Une des particularités de Steel est la présence des deux derniers indices, ici requires $\lambda\_ \rightarrow \top$ et ensures λs _ s' $\rightarrow$ s'.[r1] = s.[r2] $\wedge$ s'.[r2] = s.[r1] Ces deux derniers indices sont des prédicats opérant sur des *sélecteurs*, qui ne dépendent que des fragments de la mémoire correspondant à la spécification utilisant la logique de séparation. Cette restriction est nécessaire pour assurer que ces prédicats interagissent correctement avec notre logique de séparation, et en particulier avec la *frame rule* [7] qui permet un raisonnement modulaire. Le sélecteur d'une référence s.[r] correspond à la valeur qu'elle contient. Pour swap, nous pouvons donc spécifier la correction fonctionnelle de la fonction en utilisant ces prédicats: la valeur s.[r1] de r1 dans l'état initial s correspond à la valeur s'.[r2] de r2 dans l'état final s', et vice-versa.

Automatisation de Steel L'utilisation de logique de séparation et de prédicats opérants sur des sélecteurs permet une séparation des obligations de preuve en deux catégories. En Steel, par construction, toutes les obligations de preuves liées à la logique de séparation sont exprimées comme des équivalences modulo réécritures associatives et commutatives de l'opérateur $*$. Pour décharger ces obligations, nous avons défini une procédure de décision partielle qui utilise l'unificateur d'ordre supérieur de F⋆ pour effectuer une unification modulo réécritures. Cette procédure de décision est implémentée comme une tactique F⋆, ce qui garantit sa validité par rapport à F⋆ même. Cette procédure est partielle, et requiert parfois une application manuelle de lemmes, par exemple pour dérouler un prédicat récursif, ou pour introduire ou éliminer un quantificateur. Les obligations de preuves liées aux sélecteurs sont directement déchargées par un solveur SMT, utilisant le support natif de F⋆ pour la vérification par SMT. Pour swap, la répartition entre prédicats de logique de séparation et prédicats de sélecteurs— qui spécifient respectivement la sûreté mémoire et la correction fonctionnelle— est naturelle, et permet une vérification entièrement automatique de ce programme.

Conclusion Au-delà de cet exemple simple, Steel contient aussi deux effets, SteelAtomic, et SteelGhost. Le premier permet de représenter l'atomicité de programmes, nécessaire pour raisonner sur des invariants partagés entre plusieurs exécutions concurrentes. Le second permet d'opérer sur un état fantôme, souvent utile lors de preuves complexes. À l'aide de ces fonctionnalités, Steel a été utilisé pour vérifier une large variété de programmes [1], tels que des arbres binaires de recherche automatiquement équilibrés, un compteur monotone concurrent initialement proposé par Owicki-Gries [6], une file concurrente à deux verrous proposée par Michael et Scott [5], ou une plateforme pour encoder des sessions à types dépendents entre deux parties. Dans cette démonstration, nous proposons de présenter plusieurs bibliothèques vérifiées en Steel, qui illustrent l'expressivité et la programmabilité de l'outil.

References

[1] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. Steel: Proof-oriented programming in a dependently typed concurrent separation logic. 2021.

[2] Ralf Jung, Robbert Krebbers, Lars Birkedal, and Derek Dreyer. Higher-order ghost state. 2016.

[3] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[4] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. 2015.

[5] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. 1996.

[6] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.

[7] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 2012.

[8] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Programming and proving with indexed effects, 2021. In Submission.

[9] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. 2016.

[10] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. SteelCore: An extensible concurrent separation logic for effectful dependently typed programs. 2020.

Number Notation dans Coq (démonstration)

Pierre Roux[1]

ONERA/DTIS, Université de Toulouse, F-31055 Toulouse - France pierre.roux@onera.fr

Résumé

L'assistant de preuve Coq dispose d'une fonctionnalité permettant à l'utilisateur de définir ses propres notations avec une grande souplesse. Dans ce cadre, les constantes numériques étaient initialement interprétées et affichées par du code dédié dans des plugins OCaml. Depuis Coq 8.9 (janvier 2019), la commande Numeral Notation (aujourd'hui renommée Number Notation) permet d'implémenter ces parsers et printers de constantes numériques directement dans Coq.

Suite entre autre à l'introduction des flottants primitifs, cette commande a depuis connue un certain nombre d'extensions qui seront présentées dans cette démonstration : valeurs à virgules ou exposant, valeurs hexadécimales, interprétation vers des types inductifs paramétrés ou non inductifs.

1 La commande `Number Notation`

Dans Coq [1], depuis janvier 2019 et sa version 8.9, la commande `Number Notation` permet de déclarer directement dans Coq des parsers et printers pour les nombres entiers. Cela les rend plus facile à écrire, modifier et maintenir mais permet également de prouver leur bon comportement, assurant qu'ils n'introduisent pas d'incohérence de Pollack [3].

Ces entiers suivent l'expression régulière `-? [0-9]+` et sont, par défaut, interprétés comme des entiers naturels de type `nat`, comme on peut le constater en demandant à Coq `Check 42`. Cela est obtenu grâce au code suivant dans le prélude de la librairie standard de Coq (fichier `theories/Init/Prelude.v`) :

```
Number Notation nat Nat.of_num_uint Nat.to_num_uint (abstract after 5001)
  : nat_scope.
```

avec un parser `Nat.of_num_uint` de type `Number.uint -> nat` et un printer `Nat.to_num_uint` de type `nat -> Number.uint`. Le type `Number.uint` encapsulant un type `Decimal.uint` représentant les nombres comme listes de chiffres

```
Inductive uint : Set :=
  | Nil : Decimal.uint
  | D0 : Decimal.uint -> Decimal.uint
  | D1 : Decimal.uint -> Decimal.uint
  (* ... *)
  | D9 : Decimal.uint -> Decimal.uint
```

Les parsers/printers peuvent être des fonctions partielles

```
Definition bool_of_uint : Number.uint -> option bool := (* ... *)
Definition uint_of_bool : bool -> Number.uint := (* ... *)
Number Notation bool bool_of_uint uint_of_bool : bool_scope.
```

et on peut parser des valeurs signées en remplacant `Number.uint` par `Number.int`.

2 Nombres décimaux

Depuis Coq 8.10 (octobre 2019), on dispose de nombres avec virgule et exposant par exemple pour les rationnels de la librairie standard

```
Require Import QArith. Local Open Scope Q_scope. Check 1.23e-2.
```

suivant l'expression régulière `-? [0-9]+ ( . [0-9]+ )? ( [eE] [+-]? [0-9]+ )?`. Le type `Decimal.decimal` est alors utilisé, similairement à `Decimal.uint` ci dessus.

3 Nombres hexadécimaux

Depuis Coq 8.12 (juillet 2020), on dispose également d'une syntaxe hexadécimale (par exemple `Check 0x2a.`) suivant l'expression `-? ( 0x | 0X ) [0-9a-fA-F]+ ( . [0-9a-fA-F]+ )? ( [pP] [+-]? [0-9]+ )?` autorisant les virgules et exposant (binaire), utilisés par exemple pour afficher exactement les nombres flottants.

4 Commentaires

Des `_` peuvent être utilisés comme commentaires pour rendre des grandes constantes plus lisibles : `Check 1_000_000.`

5 Types non inductifs

Pour pouvoir facilement appeler les fonctions de print quand un terme est affiché et pouvoir faire des match, les types parsés doivent être des inductifs, éventuellement paramétrés.

Certains types numériques ne sont pas inductifs, par exemple les réels de la librairie standard [2]. La commande `Number Notation` permet de parser vers ces types, en utilisant un type inductif ad hoc et en établissant une correspondance entre cet inductif et des constantes du type concerné. Par exemple pour les réels (c.f., fichier `theories/Reals/Rdefinitions.v`), on déclare le type `IR` puis on décrit une association entre les constructeurs de ce type et les constantes de type `R` qui nous intéressent

```
Require Import Reals.
Print IR.
Number Notation R of_number to_number (via IR
  mapping [IZR => IRZ, Q2R => IRQ, Rmult => IRmult, Rdiv => IRdiv,
           Z.pow_pos => IZpow_pos, Z0 => IZ0, Zpos => IZpos, Zneg => IZneg])
  : R_scope.
```

6 Conclusion

Aujourd'hui dans Coq master, grâce à ces mécanismes, tous les anciens parsers/printers numériques sous forme de plugins OCaml ont pu être remplacés par des implémentations en Coq. Cela les rend plus facile à écrire, modifier et maintenir mais permet également de prouver leur bon comportement, assurant qu'ils n'introduisent pas d'incohérence de Pollack [3]. On peut trouver de telles preuves dans les fichiers `theories/Numbers/Decimal*.v` de la librairie standard par exemple.

Références

[1] The Coq development team. *The Coq proof assistant reference manual*, 2021. Version 8.13.

[2] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, décembre 2001.

[3] Freek Wiedijk. Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science*, 285 :85–100, 2012. Proceedings of the 9th International Workshop On User Interfaces for Theorem Provers (UITP10).

A Fichier Coq de la démonstration

```
(* Démonstration réalisée avec Coq 8.13 *)

(* Jusque récemment (Coq 8.10), le parseur de Coq lisait
   des nombres entiers : -? [0-9]+

   Par défaut interprété comme des nat *)
Check 42.

(* grâce au code suivant dans le prélude de la librairie standard
   (fichier theories/Init/Prelude.v) *)
Number Notation nat Nat.of_num_uint Nat.to_num_uint (abstract after 5001)
  : nat_scope.

(* avec un parseur de type Number.uint -> nat *)
Check Nat.of_num_uint.

(* et un printer de type nat -> Number.uint *)
Check Nat.to_num_uint.

(* avec Number.uint *)
Print Number.uint.
(* et Decimal.uint des listes de chiffres de 0 à 9 *)
Print Decimal.uint.

(* le [abstract after 5001] évite des débordements de pile
   en n'évaluant pas le parseur pour les grands nombres *)
Check 12000.

(* et [: nat_scope] place la notation dans le scope [nat_scope]
   (ouvert par défaut) *)

(* On peut parser des valeurs signées
   en remplaçant Number.uint par Number.int *)
Print Decimal.int.

(* Les parseur/printer peuvent être des fonctions partielles : *)

Definition bool_of_uint : Number.uint -> option bool :=
  fun u =>
    match u with
    | Number.UIntDecimal (Decimal.D0 Decimal.Nil) => Some false
    | Number.UIntDecimal (Decimal.D1 Decimal.Nil) => Some true
    | _ => None
    end.

Definition uint_of_bool : bool -> Number.uint :=
```

```
  fun b =>
    if b then Number.UIntDecimal (Decimal.D1 Decimal.Nil)
    else Number.UIntDecimal (Decimal.D0 Decimal.Nil).

Number Notation bool bool_of_uint uint_of_bool : bool_scope.

Check 0%bool.
Check 1%bool.
Fail Check 2%bool.

(********************)
(* Nombres décimaux *)
(********************)

(* Depuis Coq 8.10, on dispose de nombres avec virgule et exposant :
   -? [0-9]+ ( . [0-9]+ )? ( [eE] [+-]? [0-9]+ )? *)

(* Par exemple pour les rationnels de la lib standard *)
Require Import QArith.
Local Open Scope Q_scope.
Check 1.23e-2.

Goal 1.23e-2 = 0.0123.
Proof. reflexivity. Qed.

Goal 1.23e-2 = 123 / 10000.
Proof. reflexivity. Qed.

(* Les parseurs et printers donnés à Number Notation utilisent alors
   le type Decimal.decimal *)
Print Decimal.decimal.
(* Variant decimal : Set :=
   | Decimal : Decimal.int -> Decimal.uint -> Decimal.decimal
         (* partie entière "." partie fractionnaire *)
   | DecimalExp : Decimal.int -> Decimal.uint -> Decimal.int -> Decimal.decimal
           (* partie entière "." partie fractionnaire "e" exposant *)

  Les entiers peuvent être représentés avec une partie fractionnaire vide :
  Decimal.Nil : Decimal.uint *)

(************************)
(* Nombres hexadécimaux *)
(************************)

(* Depuis Coq 8.12, on dispose également d'une syntaxe hexadécimale *)
Local Open Scope nat_scope.
Check 0x2a.
Check 0X2a.
```

```
(* l'affichage est controlé par le dernier scope ouvert *)
Local Open Scope hex_nat_scope.
Check 42.

(* virgule et exposant (binaire) sont également admis :
   -? ( 0x | 0X ) [0-9a-fA-F]+ ( . [0-9a-fA-F]+ )? ( [pP] [+-]? [0-9]+ )? *)
Local Open Scope Q_scope.
Check 0x0.c.

Goal 0x0.c = 12 / 16.
Proof. reflexivity. Qed.

Check 0x1p8.

Goal 0x1p8 = 256.
Proof. reflexivity. Qed.

(* Les parseurs et printers donnés à Number Notation utilisent alors
   le type HexaDecimal.hexadecimal similaire à Decimal.decimal *)
Print Hexadecimal.hexadecimal.

(* Le choix décimal/hexadécimal est alors disponible
   dans le type somme Number.number *)
Print Number.number.

(*****************)
(* Commentaires *)
(*****************)

(* Des _ peuvent être utilisés comme commentaires
   pour rendre des grandes constantes plus lisibles. *)
Check 1_000_000.

(********************)
(* Types inductifs *)
(********************)

(* Pour pouvoir facilement appeler les fonctions de print
   quand un terme est affiché et pouvoir faire des match,
   les types parsés doivent être des inductifs. *)

(* C.f. ci dessus : nat, bool, Q *)

(*****************************)
(* Types inductifs paramétrés *)
(*****************************)
```

```
(* Exemple : listes de booléens *)

Notation blist := (list bool).

Definition uint_of_blist : blist -> Number.uint :=
  fun l =>
    let fix aux l :=
      match l with
      | nil => Decimal.Nil
      | cons true l => Decimal.D1 (aux l)
      | cons false l => Decimal.D0 (aux l)
      end in
    Number.UIntDecimal (aux l).

Definition blist_of_uint : Number.uint -> option blist :=
  fun u =>
    let fix aux l :=
      match l with
      | Decimal.Nil => Some nil
      | Decimal.D1 l =>
          match aux l with None => None | Some l => Some (cons true l) end
      | Decimal.D0 l =>
          match aux l with None => None | Some l => Some (cons false l) end
      | _ => None
      end in
    match u with
    | Number.UIntDecimal l => aux l
    | Number.UIntHexadecimal _ => None
    end.

Number Notation blist blist_of_uint uint_of_blist : bool_scope.

Local Open Scope bool_scope.

Check 1011.
Set Printing All.
Check 1011.
Unset Printing All.
Check cons true (cons false nil).  (* affiché 10 *)
Check cons 1%nat (cons 0%nat nil).  (* pas affiché *)

(* On peut ignorer un paramètre avec _ *)

(* Exemple : affichage de toutes les listes comme leur longueur *)

Definition uint_of_list : list unit -> Number.uint :=
  fun l => Nat.to_num_uint (length l).
```

```
Definition list_of_uint : Number.uint -> list unit :=
  fun u =>
    let fix aux n := match n with O => nil | S n => cons tt (aux n) end in
    aux (Nat.of_num_uint u).

Notation any_list := (list _).
Number Notation any_list list_of_uint uint_of_list : list_scope.

Local Open Scope list_scope.

Check 3.
Set Printing All.
Check 3.
Unset Printing All.
Check cons tt (cons tt nil).  (* affiché 2 *)
Check cons true (cons false nil).  (* aussi affiché 2 *)
Check cons 1%nat (cons 0%nat nil).  (* aussi affiché 2 *)

(************************)
(* Types non inductifs *)
(************************)

(* Certains types numériques ne sont pas inductifs, par exemple les
   réels de la librairie standard. La commande Number Notation,
   permet de parser vers ces types, en utilisant un type inductif
   ad hoc et en établissant une correspondance entre cet inductif
   et des constantes du type concerné. *)

(* Par example pour les réels (c.f., fichier theories/Reals/Rdefinitions.v),
   on déclare le type IR *)
Require Import Reals.
Print IR.
(* Inductive IR : Set :=
   | IRZ : IZ -> IR  (* constantes entières *)
   | IRQ : Q -> IR  (* constantes rationnelles *)
   | IRmult : IR -> IR -> IR  (* multiplication (exposants positifs) *)
   | IRdiv : IR -> IR -> IR. (* division (exposants négatifs) *) *)
Print IZ.
(* Inductive IZ : Set :=
   | IZpow_pos : Z -> positive -> IZ  (* pour encoder les exposants 10^n *)
   | IZ0 : IZ
   | IZpos : positive -> IZ
   | IZneg : positive -> IZ *)

(* la fonction de parse retourne un IR *)
Check of_number.

(* et la fonction de print prend un IR en entrée *)
```

```
Check to_number.

(* Puis la notation est déclarée avec *)
Number Notation R of_number to_number (via IR
  mapping [IZR => IRZ, Q2R => IRQ, Rmult => IRmult, Rdiv => IRdiv,
           Z.pow_pos => IZpow_pos, Z0 => IZ0, Zpos => IZpos, Zneg => IZneg])
  : R_scope.
(* Ici, l'option "via IR" signifie que le type (non inductif) R est
   parsé au travers du type IR, en associant chacun de ses constructeurs
   à une constante de R suivant la liste donnée :

   constructeur | est traduit en la constante
   -------------+-----------------------------
   IRZ          | IZR (injection de Z dans R)
   IRQ          | Q2R (injection de Q dans R)
   IRmult       | Rmult (multiplication dans R)
   IRdiv        | Rdiv (division dans R)
   IZpow_pos    | Z.pow_pos (puissance dans Z)
   IZ0          | Z0 (0 dans Z)
   IZpos        | Zpos (positifs dans Z)
   IZneg        | Zneg (négatifs dans Z) *)

Local Open Scope R_scope.

Check 1.2e3.
Set Printing All.
Check 1.2e3.
Unset Printing All.

(************************)
(* Arguments implicites *)
(************************)

(* Certains types inductifs ont des arguments implicites.

   Par exemple, le type Vector.Fin.t encode les entiers naturels plus
   petit qu'une certaine borne n. *)
Require Import Vector.
Print Fin.t.
(* Inductive t : nat -> Set :=
   | F1 : ∀n : nat, Fin.t (S n)
   | FS : ∀n : nat, Fin.t n -> Fin.t (S n)

   Arguments Fin.F1 {n}
   Arguments Fin.FS {n} _

   On note que l'argument n est implicite dans les constructeurs. *)
```

```
(* On utilise alors le type inductif nat (qui n'a pas ce paramètre n)
   come proxy *)

Declare Scope fin_scope.
Delimit Scope fin_scope with fin.
Local Open Scope fin_scope.
Number Notation Fin.t Nat.of_num_uint Nat.to_num_uint (via nat
  mapping [[Fin.F1] => 0, [Fin.FS] => S]) : fin_scope.
(* noter les crochets autour des constructeurs Fin.F1 et Fin.FS :
   leurs arguments implicites seront
   - ignorés en traduisant vers nat avant d'appeler le printer
   - remplacés par des trous _ en traduisant depuis nat après le parse *)

(* Maintenant, 2 est parsé comme Fin.FS (Fin.FS Fin.F1),
   soit @Fin.FS _ (@Fin.FS _ (@Fin.F1 _)) *)
Check 2.

(* qui peut être de type Fin.t 3 par exemple (entiers < 3, soit 0, 1 et 2) *)
Check 2 : Fin.t 3.

(* mais pas de type Fin.t 2 (entiers < 2, soit 0 et 1) *)
Fail Check 2 : Fin.t 2.

(***************)
(* Conclusion *)
(***************)

(* Aujourd'hui dans Coq master, grâce à ces mécanismes, tous les
   anciens parsers/printers numériques sous forme de plugins OCaml ont
   pu être remplacés par des implémentations en Coq.  Cela les rend
   plus facile à écrire, modifier et maintenir mais permet également
   de prouver leur bon comportement, assurant qu'ils n'introduisent
   pas d'incohérence de Pollack. On peut trouver de telles preuves
   dans les fichiers theories/Numbers/Decimal*.v de la librairie
   standard par exemple. *)

(* Plus de détails dans le manuel de Coq :
   https://coq.inria.fr/refman/user-extensions/syntax-extensions.html#numbers-
    and-strings *)
```

Auteurs

Le comité d'organisation des JFLA 2022 remercie chaleureusement ses généreux sponsors.

CEA List

GDR GPL

Nomadic Labs

OCamlPro

Fondation OCaml

Tarides

TrustInSoft

Tweag I/O