Journées Francophones des Langages Applicatifs 2019

Nicolas Magaud, Zaynah Dargaye

▶ **To cite this version:**

Nicolas Magaud, Zaynah Dargaye. Journées Francophones des Langages Applicatifs 2019. Nicolas Magaud; Zaynah Dargaye. Journées Francophones des Langages Applicatifs 2019, Jan 2019, Les Rousses, France. publié par les auteurs, 2019. hal-01985195

HAL Id: hal-01985195
https://inria.hal.science/hal-01985195

Submitted on 17 Jan 2019

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

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

JFLA 2019

JOURNÉES FRANCOPHONES DES LANGAGES APPLICATIFS

LES ROUSSES (30 JANVIER - 2 FÉVRIER 2019)

Université de Strasbourg

list cea tech

Préface

En 2019, les 30èmes journées francophones des langages applicatifs (JFLA) se déroulent aux Rousses dans le Jura. Après une édition méditerranéenne à Banyuls-sur-Mer, les JFLA retrouvent donc une nouvelle fois la montagne. Chaque année, les JFLA réunissent, dans un cadre convivial, des concepteurs, des développeurs et des utilisateurs des langages fonctionnels, des assistants de preuve et des outils de vérification de programmes en présentant des travaux variés, allant des aspects les plus théoriques aux applications industrielles.

Cette année, nous avons sélectionné 11 articles de recherche et 4 articles courts. Les thématiques sont variées : preuve formelle, exécution symbolique, vérification de programmes, langages de programmation, mais aussi théorie des catégories et programmation synchrone. La sélection a été faite par les membres du comité de programme à partir des articles reçus après les 18 soumissions de résumés. Ceux-ci ont été aidés dans leur travail par des rapporteurs extérieurs que nous souhaitons remercier ici : Timothy Bourke, Adrien Guatto et Pascal Schreck. Cette année, le format des papiers courts permet de faire connaître à la communauté des JFLA des travaux en cours ou bien récemment publiés.

Pour compléter le programme, nous bénéficions de deux cours, l'un par Guillaume Melquiond sur les liens entre l'arithmétique des ordinateurs et les preuves formelles; l'autre par Pierre-Marie Pédrot sur la prise en compte des effets dans la théorie des types. Nous assisterons également à deux exposés invités, le premier par Ilaria Castellani sur les types de session qui peuvent être vus comme une abstraction pour les protocoles de communication et le second par Pierre Courtieu sur l'étude formelle de protocoles de déplacement de robots.

Enfin cette 30ème édition des JFLA est l'occasion de se retrouver et d'observer le chemin parcouru depuis quelques années. Nous avons sélectionné les 3 orateurs les plus prolifiques des 10 dernières éditions (2009 à 2018). Tous les 3 ont, chacun en ce qui les concerne, présenté au moins 7 contributions aux JFLA en 10 ans. Il s'agit de Jean-Christophe Filliâtre qui nous fera une synthèse de ses 25 années d'expérience(s) en programmation avec OCaml, de Louis Mandel qui fera un tour d'horizon de la programmation synchrone aux JFLA et d'Alan Schmitt qui nous parlera de sémantiques formelles certifiées.

Nous remercions chaleureusement la cellule Congrès de l'Université de Strasbourg qui nous a accompagnée sur le chemin logistique et administratif des JFLA 2019: Christine Guibert, et surtout Marion Oswald, qui a été présente de bout en bout pour gérer tous les aléas administratifs rencontrés.

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

Nicolas Magaud & Zaynah Dargaye
Université de Strasbourg CEA List

Comité de programme

Nicolas Magaud	ICube, Université de Strasbourg (président)
Zaynah Dargaye	CEA List (vice-présidente)
Guillaume Burel	École Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise (ENSIIE)
Evelyne Contejean	CNRS, Université Paris-Sud
Claire Dross	Adacore
Guillaume Dufay	Prove & Run
Benjamin Grégoire	Inria Sophia-Antipolis Méditerranée
Sébastien Hinderer	Inria Paris
Marc Pouzet	ENS
Yann Régis-Gianas	IRIF
Bernard Serpette	Inria Bordeaux Sud-Ouest
Mihaela Sighireanu	IRIF
Julien Tesson	LACL

Comité de pilotage

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

Table des matières

Cours invités 1

Arithmétique des Ordinateurs et Preuves Formelles 3
Guillaume Melquiond

Des Théories des Types qui font de l'Effet 13
Pierre-Marie Pédrot

Exposés invités 15

Types de Session : une Abstraction pour les Protocoles de Communication 17
Ilaria Castellani

Les Protocoles de Déplacement de Robots : l'Algorithmique Distribuée comme Terrain de Jeu pour la Preuve Formelle 19
Pierre Courtieu

Session spéciale : 30 ans de JFLA 21

Retour sur 25 ans de Programmation avec OCaml 23
Jean-Christophe Filliâtre

Programmation Synchrone aux JFLA 25
Guillaume Baudart, Louis Mandel et Marc Pouzet

Sémantiques Formelles et Certifiées 41
Alan Schmitt

Articles 43

Suspension et Fonctorialité : Deux Opérations Implicites Utiles en CaTT . . 45
Thibaut Benjamin et Samuel Mimram

SQL à l'Épreuve de Coq : Une sémantique Formelle pour SQL 61
Véronique Benzaken et Évelyne Contejean

SMTCoq: Automatisation Expressive et Extensible dans Coq 77
Valentin Blot, Amina Bousalem, Quentin Garchery et Chantal Keller

CAMLroot: Revisiting the OCaml FFI 93
Frédéric Bour

Arguments Cadencés dans un Compilateur Lustre Vérifié 109
Timothy Bourke et Marc Pouzet

Formalisation en Coq d'Algorithmes de Filtres Numériques 125
Diane Gallois-Wong

Combinatoire Formelle avec Why3 et Coq 139
Alain Giorgetti, Catherine Dubois et Rémi Lazarini

Axiomes de Continuité en Géométrie Neutre : une Étude Mécanisée en Coq155
Charly Gries, Julien Narboux et Pierre Boutry

Unboxing Mutually Recursive Type Definitions in OCaml 173
Simon Colin, Rodolphe Lepigre et Gabriel Scherer

Typer: ML Boosted with Type Theory and Scheme 193
Stefan Monnier

De l'Assembleur sur la Ligne ? Appelez TInA ! 209
Frédéric Recoules, Sébastien Bardin, Richard Bonichon, Laurent Mounier et Marie-Laure Potet

Articles courts 225

Learn-OCaml : un Assistant à l'Enseignement d'OCaml 227
Çagdas Bozman, Benjamin Canou, Roberto Di Cosmo, Pierrick Couderc, Louis Gesbert, Grégoire Henry, Fabrice Le Fessant, Michel Mauny, Carine Morel, Loïc Peyrot et Yann Régis-Gianas

Formally Verified Decomposition of Non-binary Constraints into Equivalent Binary Constraints 237
Catherine Dubois

En Finir avec les Faux Positifs grâce à l'Exécution Symbolique Robuste . . . 245
Benjamin Farinier, Sébastien Bardin, Richard Bonichon et Marie-Laure Potet

Un Mécanisme de Preuve par Réflexion pour Why3 et son Application aux Algorithmes de GMP 253
Raphaël Rieu-Helft

Cours invités

Arithmétique des Ordinateurs et Preuves Formelles

Guillaume Melquiond

Inria, LRI, Univ. Paris-Saclay
guillaume.melquiond@inria.fr

Résumé

Cet article complète l'exposé du même nom et en reprend les grandes idées. L'exposé s'intéresse aux liens entre arithmétique des ordinateurs et vérification automatique, que ce soit pour de la preuve de programmes ou de théorèmes, le tout dans le cadre formel de l'assistant de preuve Coq. L'exposé est construit en deux parties.

La première s'intéresse à la preuve automatique de théorèmes mathématiques à l'aide de méthodes venues de l'arithmétique des ordinateurs. Il s'agit d'abord d'implanter et de formaliser une arithmétique à virgule flottante simplifiée mais efficace afin d'approcher les calculs réels. Puis une arithmétique d'intervalles peut être implantée au-dessus, offrant un moyen fiable de calculer des bornes sur des expressions à valeurs réelles. L'arithmétique d'intervalles dans sa version naïve est certes fiable mais rarement efficace à cause de l'effet de corrélation. L'étape suivante est donc de construire des approximations polynomiales fiables pour réduire cet effet. En combinant tout cela avec de la preuve par réflexion, il est alors possible de prouver automatiquement des bornes fines sur des expressions à valeurs réelles. Le procédé peut être poussé jusqu'à des intégrales propres voire même impropres. Tout cela a été intégré à la bibliothèque CoqInterval.

La deuxième partie s'intéresse à la vérification formelle d'algorithmes qui mettent en oeuvre de l'arithmétique à virgule flottante. C'était déjà le cas de certains des algorithmes de la première partie, mais l'utilisation de l'arithmétique d'intervalles les rendait en grande partie corrects par construction. Par contre, si l'on s'intéresse à des bibliothèques efficaces d'approximation de fonctions mathématiques (les « libm »), il n'y a plus rien de cela. Sans une vérification formelle, il est alors difficile de se convaincre que le code ne contient pas de nombreux bogues subtils dus à l'arithmétique à virgule flottante. Il est alors nécessaire d'effectuer une analyse fine des erreurs d'arrondi commises afin de s'assurer que les valeurs calculées par le code approchent correctement la fonction mathématique souhaitée. Mais une telle analyse est longue et difficile, surtout dans un cadre formel. C'est pour cela que l'outil Gappa a été conçu. Il permet de vérifier automatiquement des propriétés sur des algorithmes flottants, par une combinaison d'arithmétique d'intervalles, de réécriture et d'analyse de l'erreur directe. Il est aussi capable de générer des preuves formelles pour décharger des buts Coq.

1 Introduction

L'arithmétique à virgule flottante est une arithmétique bien adaptée au calcul en machine et son utilisation très majoritaire consiste en l'approximation d'opérations sur les nombres réels. Inspirée de la notation scientifique, elle offre une plage étendue de valeurs tout en garantissant un nombre conséquent de chiffres significatifs. Ainsi, le format *binary64* est capable de représenter des nombres de 10^{-308} à 10^{308} avec au moins 15 chiffres décimaux significatifs. La norme internationale IEEE 754 décrit précisément les formats et les opérations arithmétiques [8]. Cette norme étant très suivie, il est possible d'effectuer des calculs flottants dans un très grand nombre d'environnements.

On pourrait donc croire que, pour ce qui est d'effectuer des calculs numériques en machine, au moins d'un point de vue arithmétique, la question est réglée. Malheureusement, ce n'est pas

le cas. Tout d'abord, même si la plage de valeurs semble immense, nous ne sommes pas à l'abri d'un programme qui, lors de ses calculs, sortirait de cette plage, au moins temporairement. Ainsi, si la norme d'un vecteur excède 10^{154}, la façon la plus naturelle de calculer cette norme va provoquer un débordement de capacité. Dans ce cas, le calcul flottant va renvoyer $+\infty$, ce qui est assez loin de la valeur attendue.

Un problème plus pernicieux est la question de la qualité numérique des calculs (*accuracy* en anglais, à ne pas confondre avec *precision*). A priori, tout semblait parfait pourtant. Quinze chiffres décimaux sont en effet suffisants pour représenter précisément la plupart des valeurs intéressantes en pratique (oublions les problématiques monétaires). Qui plus est, la norme IEEE-754 garantit que, si le résultat d'une opération flottante n'est pas exactement représentable, le nombre représentable le plus proche du résultat exact doit être renvoyé. Ce nombre le plus proche sera représenté dans la suite par $\circ(\bullet)$. Malheureusement, ce n'est pas parce que chaque opération intermédiaire est correcte avec quinze chiffres que le résultat final l'est. Pour s'en convaincre, il suffit de considérer le calcul approché de $(2^{60}+1)-2^{60}$. La première somme $2^{60}+1$ est très bien approchée par $\circ(2^{60}+1) = 2^{60}$. Par contre, la somme finale $(2^{60}+1)-2^{60} = 1$ est infiniment mal approchée par $\circ(\circ(2^{60}+1)-2^{60}) = \circ(2^{60}-2^{60}) = 0$.

Il existe de nombreux exemples de calcul flottant ayant eu des conséquences désastreuses. Le plus connu, car le plus dramatique, causa la mort de 28 personnes et de nombreux blessés en 1991. Durant la Première Guerre du Golfe, l'armée des États-Unis installa des systèmes Patriot pour intercepter les missiles Scud visant ses bases. Mais la très grande vitesse des missiles Scud rendait l'interception hasardeuse et la décision fut prise d'augmenter la précision de certains calculs de trajectoire (mais pas tous). Cette modification provoqua une dérive des erreurs de calcul. Laisser tourner le système quelques jours sans le redémarrer fut suffisant pour que le missile intercepteur rate sa cible d'une fraction de seconde [12].

Ce qui rend cet exemple particulièrement intéressant est que le système s'est effondré parce que des personnes ont cherché à augmenter la qualité des calculs. Cela montre à quel point les calculs numériques défient l'intuition [11]. Se pose alors naturellement la question de leur fiabilité. Comment s'assurer qu'un programme ne va provoquer aucun comportement exceptionnel, par exemple un débordement ? Comment s'assurer que le résultat calculé est suffisamment proche du résultat attendu pour être utilisable sans risque ?

La section 2 donne quelques définitions et propriétés préliminaires sur l'arithmétique à virgule flottante et l'arithmétique d'intervalles. La section 3 s'intéresse ensuite à des algorithmes arithmétiques simples qui permettent de prouver formellement et automatiquement des propriétés par le calcul numérique. Finalement, la section 4 montre comment vérifier des algorithmes bien plus subtils tels qu'on peut les trouver dans les bibliothèques de fonctions mathématiques.

2 Préliminaires

2.1 Arithmétique à virgule flottante

Pour un format donné, les nombres flottants représentent des nombres réels de la forme $m \cdot \beta^e$. Les entiers m, β et e sont le signifiant, la base et l'exposant du nombre. La base est fixée, généralement 2 ou 10. Pour des raisons matérielles, les valeurs de m et e sont contraintes. En règle générale, nous assimilerons un nombre flottant au nombre réel qu'il représente.

Pour simplifier la formalisation, nous ne considérons que des formats $\mathcal{F}$ pour lesquels il existe une fonction $\varphi \in \mathbb{Z} \to \mathbb{Z}$ telle que

$$\mathcal{F} = \{x \in \mathbb{R} \mid x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \in \mathbb{Z}\}$$

Format	β	ϱ	$e_{\min}$
binary32	2	24	-149
binary64	2	53	-1074
binary128	2	113	-16494
decimal32	10	7	-101
decimal64	10	16	-398
decimal128	10	34	-6176

TABLE 1 – Paramètres des formats décrits par la norme IEEE-754.

avec $\mathrm{mag}(x) = \lfloor \log_\beta |x| + 1 \rfloor$.

Deux familles de formats nous intéressent plus particulièrement ici. Les formats FLX sont décrits par des fonctions $\varphi(e) = e - e_{\min}$ tandis que les formats FLT sont décrits par $\varphi(e) = \max(e - \varrho, e_{\min})$. Cette dernière famille de formats est suffisante pour représenter les formats flottants classiques, si l'on fait abstraction des problèmes liés aux débordements de capacité. La table 1 indique comment choisir les paramètres ϱ et $e_{\min}$. Ceci étant dit, de nombreuses autres fonctions φ sont possibles, donnant des formats plus ou moins exotiques [4, §3.1.3].

La norme IEEE-754 indique que chaque opération flottante doit se comporter comme si elle calculait d'abord le résultat infiniment précis puis elle l'arrondissait au format de destination. Cela justifie l'introduction d'un opérateur d'arrondi $\square(\bullet)$. Une somme flottante entre deux nombres flottants u et v sera ainsi notée $\square(u + v)$.

Les opérateurs d'arrondi qui nous intéressent ont la forme suivante :

$$\square(x) = \lfloor x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \rceil \cdot \beta^{\varphi(\mathrm{mag}(x))},$$

avec $\lfloor \bullet \rceil$ une fonction partie entière.

Si l'on choisit la partie entière inférieure, on obtient l'arrondi vers $-\infty$, tandis que si l'on choisit la partie entière supérieure, on obtient l'arrondi vers $+\infty$:

$$\begin{aligned} \bigtriangledown(x) &= \lfloor x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \rfloor \cdot \beta^{\varphi(\mathrm{mag}(x))}, \\ \bigtriangleup(x) &= \lceil x \cdot \beta^{-\varphi(\mathrm{mag}(x))} \rceil \cdot \beta^{\varphi(\mathrm{mag}(x))}. \end{aligned}$$

Enfin, si l'on choisit l'entier le plus proche du réel (avec priorité à l'entier pair dans le cas ambigu), on obtient l'arrondi au plus près, au sens de la norme IEEE-754.

2.2 Arithmétique d'intervalles

Dénotons $\mathbb{I}$ les sous-ensembles fermés et connectés de $\mathbb{R}$. Il s'agit de $\emptyset$ et des intervalles $(-\infty; v]$, $[u; +\infty)$ et $[u; v]$ avec $u \leq v$ des nombres réels. Dans ce qui suit, nous nous intéresserons principalement aux intervalles dont les deux bornes sont des nombres flottants.

On dira d'une fonction $\mathbf{f} \in \mathbb{I}^n \to \mathbb{I}$ qu'elle est une extension par intervalles de $f \in \mathbb{R}^n \to \mathbb{R}$ si elle satisfait la propriété d'inclusion :

$$\forall \mathbf{x}_1 \in \mathbb{I}, \ldots, \mathbf{x}_n \in \mathbb{I},\ \forall x_1 \in \mathbb{R}, \ldots, x_n \in \mathbb{R},$$
$$x_1 \in \mathbf{x}_1 \wedge \ldots \wedge x_n \in \mathbf{x}_n \Rightarrow f(x_1, \ldots, x_n) \in \mathbf{f}(\mathbf{x}_1, \ldots, \mathbf{x}_n).$$

Une propriété intéressante des opérateurs d'arrondi vers $\pm\infty$ est $\bigtriangledown(x) \leq x \leq \bigtriangleup(x)$. Par conséquent, si l'on a deux encadrements $u \in [\underline{u}; \overline{u}]$ et $v \in [\underline{v}; \overline{v}]$, on peut en déduire les encadre-

ments suivants par monotonie des opérateurs arithmétiques :

$$\begin{aligned} u + v &\in [\nabla(\underline{u} + \underline{v}); \triangle(\overline{u} + \overline{v})], \\ u - v &\in [\nabla(\underline{u} - \overline{v}); \triangle(\overline{u} - \underline{v})], \\ u \cdot v &\in [\min(\nabla(\underline{u} \cdot \underline{v}), \nabla(\underline{u} \cdot \overline{v}), \nabla(\overline{u} \cdot \underline{v}), \nabla(\overline{u} \cdot \overline{v})); \\ &\quad \max(\triangle(\underline{u} \cdot \underline{v}), \triangle(\underline{u} \cdot \overline{v}), \triangle(\overline{u} \cdot \underline{v}), \triangle(\overline{u} \cdot \overline{v}))]. \end{aligned}$$

Cela permet donc de programmer très facilement des extensions par intervalles des opérateurs arithmétiques réels en quelques opérations flottantes. Et comme la propriété d'inclusion est préservée par composition, il est donc facile de borner une expression réelle arbitrairement compliquée.

Cependant, la propriété d'inclusion garantit seulement que l'encadrement $f(\vec{x}) \in \mathbf{f}(\vec{\mathbf{x}})$ est correct. Elle ne dit absolument rien de la finesse de l'intervalle $\mathbf{f}(\vec{\mathbf{x}})$. En particulier, celui-ci peut être trop large pour en déduire une propriété intéressante sur $f(\vec{x})$. Par exemple, de $x \in [0;1]$, on peut déduire $x - x \in [0;1] - [0;1] = [-1;1]$ par arithmétique d'intervalles. C'est un encadrement correct mais assez médiocre puisque $x - x = 0$. C'est le phénomène de perte de corrélation dû à la présence d'occurrences multiples de x.

3 Preuve par le calcul numérique

Comme indiqué dans la section précédente, l'arithmétique d'intervalles offre une façon simple de borner des expressions à valeurs réelles. Voyons voir comment transposer cela dans le cadre formel d'un système comme Coq. L'objectif est d'arriver à prouver formellement et automatiquement une propriété comme

$$\int_{-\infty}^{+\infty} \frac{(0.5 \cdot \log(\tau^2 + 2.25) + 4.1396 + \log \pi)^2}{0.25 + \tau^2} \, d\tau \leq 226.844. \tag{1}$$

3.1 Arithmétiques

La première étape consiste à formaliser une arithmétique à virgule flottante. La bibliothèque Flocq fournit une formalisation multi-base et multi-format de cette arithmétique [3][4, §3]. Mais l'important est qu'elle ne fournit pas seulement des opérateurs d'arrondi non calculables $\square(\bullet) \in \mathbb{R} \to \mathbb{R}$; elle fournit aussi des algorithmes formellement prouvés pour effectuer les opérations flottantes de base.

Dans le cas de la multiplication flottante de $m_1 \cdot \beta^{e_1}$ par $m_2 \cdot \beta^{e_2}$, l'algorithme est assez simple. Il commence par calculer le résultat exact $(m_1 m_2) \cdot \beta^{e_1+e_2}$. Puis il calcule un décalage d qui dépend de φ, de $e_1 + e_2$ et du nombre de chiffres de $m_1 m_2$ en base β. Et enfin il renvoie $\lfloor m_1 m_2 \beta^{-d} \rceil \cdot \beta^{e_1+e_2+d}$. Pour des opérations flottantes comme la division et la racine carrée, les algorithmes sont un peu plus subtils puisque le résultat exact n'est pas représentable sous la forme $m \cdot \beta^e$ [4, §3.3.2].

Une fois que l'on sait effectuer toutes les opérations flottantes de base, il est facile de construire une arithmétique d'intervalles effective sur le modèle de celle de la section 2.2. La bibliothèque CoqInterval fournit une telle arithmétique ainsi que tous les théorèmes d'inclusion correspondants [10]. Nous avons maintenant suffisamment de matériel pour prouver automatiquement et formellement au sein de Coq une propriété comme $u \cdot (v + \sqrt{v}) \geq -13$ avec $u \in [-3;4]$ et $v \in [1;2]$. Pour aller un peu plus loin, il nous faut enrichir le catalogue de fonctions disponibles.

Considérons le cas de la fonction arctan. Elle est monotone croissante, donc pour $x \in [\underline{x}; \overline{x}]$, nous avons $\arctan x \in [\arctan \underline{x}; \arctan \overline{x}]$. Pour calculer chacune des bornes de l'intervalle, nous pouvons utiliser la décomposition en série entière de arctan en 0:

$$\arctan x = x \cdot \sum_{i=0}^{\infty} (-1)^i \cdot \frac{(x^2)^i}{2i+1}.$$

Le premier problème est que le rayon de convergence de cette série ne permet de l'utiliser que si $|x| < 1$. Le second problème est que la série est infinie ; il faut donc la tronquer et pouvoir borner la valeur du reste. Fort heureusement, les termes tronqués étant décroissants en valeur absolue et de signes alternés, il suffit d'évaluer le premier terme tronqué pour borner leur somme infinie. Le dernier problème est que cette série converge lentement, donc se restreindre à $|x| < 1$ ne suffit pas, il vaut mieux avoir $|x| \leq \frac{1}{2}$ pour obtenir des bornes précises en un temps raisonnable. Pour cela, nous commençons par faire une réduction d'argument. La fonction étant impaire, nous nous ramenons au cas $x \geq 0$ puis nous utilisons les identités suivantes pour traiter le cas $x \geq \frac{1}{2}$:

$$\arctan x = \begin{cases} \frac{\pi}{4} + \arctan \frac{x-1}{x+1} & \text{pour } x \in [\frac{1}{2}; 2], \\ \frac{\pi}{2} - \arctan \frac{1}{x} & \text{pour } x \geq 2. \end{cases}$$

Pour obtenir des extensions par intervalles de exp et log, l'approche est similaire mais avec une réduction d'argument un peu plus compliquée. Pour cos et sin, cela devient encore plus compliqué puisque les fonctions ne sont pas monotones.

3.2 Approximations polynomiales et intégration

Maintenant que nous avons les briques de base pour borner de façon garantie l'intégrande de la formule (1), voyons comment passer à l'intégrale. Ignorons pour l'instant le fait que c'est une intégrale impropre et supposons plutôt que le but d'intégrer une fonction f entre $u \in \mathbf{u}$ et $v \in \mathbf{v}$. Quelques manipulations d'inégalités conduisent à l'encadrement suivant :

$$\int_u^v f \in (\mathbf{v} - \mathbf{u}) \cdot \mathbf{f}(\mathrm{hull}(\mathbf{u}, \mathbf{v})).$$

Malheureusement, cet encadrement est très grossier. Il donne par exemple $\int_{-1}^{1} x\,dx \in [-2; 2]$. Nous pouvons améliorer les choses en subdivisant l'intervalle d'intégration $[u; v]$ et en utilisant la relation de Chasles. Par exemple, $\int_{-1}^{1} x\,dx = \int_{-1}^{0} x\,dx + \int_{0}^{1} x\,dx \in [-1; 0] + [0; 1] = [-1; 1]$. Cela permet d'obtenir des bornes arbitrairement fines sur l'intégrale. Mais cette approche n'est en général pas utilisable à cause du nombre démesurément élevé de subdivisions nécessaires.

Le problème est ici que l'arithmétique d'intervalles approche le graphe d'une fonction par des rectangles ayant des côtés alignés avec les axes. Si la fonction n'est pas (presque) constante, le résultat est désastreux. Une meilleure approche consiste, étant donné $x_0 \in [u; v]$, à rechercher un polynôme p et un intervalle $\mathbf{\Delta}$ tels que

$$\forall x \in [u; v],\ f(x) - p(x - x_0) \in \mathbf{\Delta}.$$

Nous pouvons alors calculer P une primitive de p. Soit $\mathbf{P}$ une extension par intervalles de P. Nous obtenons ainsi

$$\int_u^v f \in \mathbf{P}(\mathbf{v}) - \mathbf{P}(\mathbf{u}) + (\mathbf{v} - \mathbf{u}) \cdot \mathbf{\Delta}.$$

Plutôt que de subdiviser $[u;v]$, nous pouvons augmenter le degré de p pour réduire la largeur de Δ et donc améliorer la qualité de l'encadrement final. La question est donc de savoir comment construire ces polynômes p. CoqInterval utilise pour cela des modèles de Taylor [10]. La terminologie vient du fait que, si f est une fonction de base comme exp, alors le polynôme correspondant est obtenu par la formule de Taylor-Lagrange :

$$\forall x \in [u;v],\ \exists \xi \in [u;v],\ f(x) = \sum_{k=0}^{n} \frac{f^{(k)}(x_0)}{k!}(x-x_0)^k + \frac{f^{(n+1)}(\xi)}{(n+1)!}(x-x_0)^{n+1}.$$

Et si f est une fonction composée, alors son polynôme s'obtient par composition de polynômes et un peu d'arithmétique d'intervalles donne $\mathbf{\Delta}$.

Il nous reste alors à régler le cas des intégrales impropres. Supposons cette fois que le but soit de borner $\int_u^{+\infty} fg$ avec g une fonction intégrable de signe constant sur $[u;+\infty)$. Encore une fois, quelques manipulations d'inégalités permettent d'obtenir un encadrement :

$$\int_u^{+\infty} fg \in \mathbf{f}(\mathrm{hull}(\mathbf{u}, +\infty)) \cdot \int_u^{+\infty} g.$$

Dans le cas de la formule (1), la fonction g à choisir est $\log^2 \tau/\tau^2$ dont CoqInterval sait borner l'intégrale impropre [9]. CoqInterval s'appuie sur la bibliothèque Coquelicot pour toutes les définitions et théorèmes d'intégration [2].

4 Preuve de calcul numérique

Dans la section 3, l'arithmétique à virgule flottante s'appuyait sur des formats de type FLX et les fonctions élémentaires étaient implantées de façon assez naïve. Tout cela est bien adapté pour faire de la preuve formelle par le calcul mais c'est finalement assez peu représentatif des algorithmes qui sont présents dans les bibliothèques de fonctions mathématiques. En particulier, quand le format flottant est fixé, de nombreuses optimisations sont possibles. La figure 1 présente ainsi une approximation flottante relativement efficacement de la fonction exp [6, p. 65].

La structure générale de la fonction est la même que celle présentée en section 3.1 : une réduction d'argument ramène l'entrée x dans un petit intervalle autour de zéro, puis la fonction y est évaluée, et enfin une reconstruction inverse l'effet de la réduction d'argument. Mais dans les détails, cela n'a pas grand chose à voir avec la façon dont CoqInterval implante exp. Premièrement, la réduction d'argument calcule un entier k et un nombre flottant $t \simeq x - k \cdot \log 2$ tel que $|t| \leq 355 \cdot 2^{-10}$. Deuxièmement, ce n'est pas la série entière tronquée de exp qui est évaluée mais une fonction rationnelle sortie du chapeau. Troisièmement, il n'y a aucun intervalle nulle part, donc ce n'est pas la propriété d'inclusion qui garantira la correction du résultat. Il va falloir prouver que la valeur calculée est proche de la valeur exacte, et cela en prenant en compte toutes les erreurs d'arrondi. Plus précisément, l'objectif est de prouver formellement que, si la toute dernière opération ne provoque ni *underflow* ni *overflow*, alors l'erreur relative finale est inférieure à 2^{-51}.

4.1 Erreur de méthode

Il est généralement possible de considérer séparément deux types d'erreur quand on analyse un algorithme flottant. D'un côté il y a l'erreur de méthode, c'est-à-dire l'erreur commise entre le résultat idéal et le résultat de l'algorithme si ce dernier était exécuté avec une précision

```
double cw_exp(double x) {
  if (x < -746.) return 0.;
  if (x > 710.) return INFINITY;
  // argument reduction
  double k = nearbyint(x * InvLog2);
  double t = x - k * Log2h - k * Log2l;
  // rational function evaluation
  double t2 = t * t;
  double p = 0.25 + t2 * (p1 + t2 * p2);
  double q = 0.5  + t2 * (q1 + t2 * q2);
  double f = t * (p / (q - t * p)) + 0.5;
  // reconstruction
  return ldexp(f, (int)k + 1);
}
```

FIGURE 1 – Approximation de exp au format *binary64*.

infinie. De l'autre côté il y a l'erreur d'arrondi commise entre des exécutions de l'algorithme avec des précisions soit infinie soit fixée.

Commençons par l'erreur de méthode. Elle concerne ici la fonction rationnelle qui a été choisie pour approcher exp. L'objectif est de borner la distance entre exp et $2f$ avec

$$f(t) = \frac{t \cdot p(t^2)}{q(t^2) - t \cdot p(t^2)} + 0.5,$$

où p et q sont des polynômes de degré 2 à coefficients *binary64*. Plus précisément, l'objectif est de prouver formellement

$$\forall t \in \mathbb{R},\ |t| \leq 355 \cdot 2^{-10} \Rightarrow \left| \frac{2f(t) - \exp t}{\exp t} \right| \leq 23 \cdot 2^{-62}.$$

Cette formule a une structure adaptée pour être prouvée par arithmétique d'intervalles. Malheureusement, comme $2f$ est choisie pour approcher précisément exp, c'est un cas typique de perte de corrélation (section 2.2). La Terre risque donc d'être engloutie par le Soleil avant même que la preuve soit terminée.

Fort heureusement, CoqInterval ne sait pas seulement calculer des encadrements à base d'intervalles, il sait aussi en calculer à base de polynômes, comme expliqué en section 3.1. En fait, c'est précisément pour vérifier ce genre de propriétés que les modèles de Taylor ont été ajoutés à CoqInterval. Si nous demandons à la tactique `interval` d'utiliser des polynômes de degré 9, elle produit automatiquement une preuve que Coq vérifie en une poignée de secondes.

4.2 Erreur d'arrondi

La fonction f approche $\frac{1}{2}\exp$ très précisément, mais ce n'est pas elle qui sera effectivement exécutée. C'est une version $\tilde{f}$ de f dans laquelle chaque opération arithmétique sera arrondie et donc la source d'une petite erreur. L'objectif est donc maintenant de vérifier une borne fine sur l'accumulation de toutes ces erreurs.

Tout comme pour l'erreur de méthode, la propriété semble prouvable par arithmétique d'intervalles, mais l'évaluation par intervalles de $\tilde{f}(t) - f(t)$ subit, là encore, une perte de corrélation. Nous allons cette fois utiliser l'outil Gappa pour borner l'expression [7]. Cet outil est basé sur l'arithmétique d'intervalles, ce qui lui permet de générer des preuves formelles vérifiables par Coq, mais il est aussi capable de transformer les expressions pour limiter la perte de corrélation.

Considérons le cas d'une multiplication entre deux sous-expressions u et v. Une fois exécutée en arithmétique à virgule, ces sous-expressions seront entachées d'une erreur ; notons les $\tilde{u}$ et $\tilde{v}$. Si l'on évalue directement $\tilde{u} \cdot \tilde{v} - u \cdot v$ par arithmétique d'intervalles, toute corrélation sera perdue. La réécriture suivante permet de grandement limiter cette perte :

$$\frac{\tilde{u} \cdot \tilde{v} - u \cdot v}{u \cdot v} = \frac{\tilde{u} - u}{u} + \frac{\tilde{v} - v}{v} + \frac{\tilde{u} - u}{u} \cdot \frac{\tilde{v} - v}{v}.$$

Autrement dit, plutôt que d'avoir à borner directement l'erreur relative entre $\tilde{u} \cdot \tilde{v}$ et $u \cdot v$, cette réécriture permet de considérer séparément les erreurs relatives entre $\tilde{u}$ et u et entre $\tilde{v}$ et v. Si tout se passe bien, d'autres réécritures permettront alors de limiter la perte de corrélation pour ces sous-problèmes.

Concrètement, Gappa dispose d'une base de données de quelques centaines de théorèmes et il l'utilise pour saturer la négation du but jusqu'à en déduire une contradiction. Ces théorèmes sont choisis pour reproduire les raisonnements que l'on pourrait faire à la main pour vérifier des propriétés sur des algorithmes flottants. Cette saturation a lieu hors de Coq et seule la succession de théorèmes qui mène à une contradiction est rejouée en Coq.

4.3 Réduction d'argument

Pour l'instant, nous n'avons considéré qu'un petit intervalle autour de zéro. Il reste à expliquer pourquoi la réduction d'argument est correcte et ne provoque pas une explosion de l'erreur finale. Or il s'avère que cette réduction est extrêmement subtile. C'est même le principal intérêt du code de Cody et Waite.

Pour s'en rendre compte, il suffit de remplacer `(x - k * Log2h - k * Log2l)` par `(x - k * (Log2h + Log2l))`. A priori ça ne devrait pas beaucoup changer le résultat, tout en accélérant le code. Et pourtant, après une telle modification, la valeur calculée par la fonction n'a plus grand chose à voir avec l'exponentielle, pour certaines valeurs de x.

La particularité de cette réduction d'argument est que la constante `Log2h` a été choisie de telle sorte que la multiplication par k est en fait une opération exacte. Mais encore faut-il réussir à prouver que ça rend la réduction d'argument correcte. Nous allons à nouveau utiliser Gappa mais il va cette fois falloir l'aider. Par exemple, le phénomène de perte de corrélation fait que Gappa est incapable de borner finement

$$x - \lfloor x \cdot \texttt{InvLog2} \rceil \cdot \texttt{Log2h}.$$

Une façon de résoudre le problème est de fournir l'égalité $x = x \cdot \texttt{InvLog2} \cdot \texttt{InvLog2}^{-1}$ à Gappa. L'outil se retrouve alors à devoir borner

$$(x \cdot \texttt{InvLog2}) \cdot \texttt{InvLog2}^{-1} - \lfloor x \cdot \texttt{InvLog2} \rceil \cdot \texttt{Log2h},$$

qui est une différence entre deux expressions ayant une structure similaire. Les méthodes décrites dans la section 4.2 peuvent alors s'appliquer. Un autre exemple est l'expression

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - (x - k \cdot \log 2).$$

Là encore il y a perte de corrélation. Cette fois, il faut remplacer $\log 2$ par $\texttt{Log2h} + \delta$ et distribuer un peu les opérations pour que Gappa se retrouve à devoir borner

$$((x - k \cdot \texttt{Log2h}) - k \cdot \texttt{Log2l}) - ((x - k \cdot \texttt{Log2h}) - k \cdot \delta).$$

Il suffit alors d'indiquer à Gappa quelle est la différence entre `Log2l` et $\delta = \log 2 - \texttt{Log2h}$, différence que `interval` n'a aucune difficulté à borner. Au final, il aura suffi de fournir trois indications (les deux égalités et cette borne) pour que l'outil vérifie automatiquement toutes les propriétés nécessaires de la réduction d'argument.

Il ne reste alors plus qu'à utiliser Gappa pour combiner l'erreur de méthode, l'erreur d'arrondi de l'évaluation de la fonction rationnelle et l'erreur de la réduction d'argument et ainsi finir la vérification formelle de la fonction.

5 Conclusion

Nous avons vu comment prouver formellement et (presque) automatiquement des propriétés sur des expressions aussi bien à valeurs réelles qu'à valeurs flottantes. Et dans un cas comme dans l'autre, la preuve s'appuie sur l'exécution d'un algorithme à base d'arithmétique d'intervalles à bornes flottantes. Ces algorithmes ont été vérifiés en Coq et sont exécutés au sein de Coq, dans le style traditionnel de la preuve par réflexion [5]. Dans un cas, la finalité est la preuve de théorèmes mathématiques, et dans l'autre, il s'agit de la vérification d'algorithmes numériques.

De nombreuses pistes sont à explorer pour améliorer les bibliothèques et outils plus avant. Dans le cas de CoqInterval, une des limitations est l'absence d'automatisation concernant les fonctions définies implicitement, par exemple par équation différentielle. Il s'agirait de concevoir et de vérifier des algorithmes capables de construire des approximations polynomiales de telles fonctions. Pour ce qui est de Gappa, les résultats vérifiés automatiquement peuvent être puissants mais au prix d'indications parfois absconses de la part de l'utilisateur pour limiter la perte de corrélation. L'utilisation de formes affines symboliques pour représenter les erreurs d'arrondi ou d'un mécanisme équivalent permettrait peut-être de diminuer le travail de l'utilisateur [13].

Il y a un point qui n'a pas été abordé. C'est bien beau de vérifier formellement une fonction C, mais qu'est-ce qui nous garantit que c'est effectivement elle qui sera exécutée au bout du compte ? Par exemple, un compilateur ne risquerait-il pas de faire l'optimisation décrite en section 4.3 ? C'est pour se prémunir de ce genre de problèmes que nous avons défini une sémantique précise pour le C et que nous avons prouvé formellement que le compilateur CompCert la préservait lors de la compilation [1].

Références

[1] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[2] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

[3] 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, *20th IEEE Symposium on Computer Arithmetic (Arith)*, pages 243–252, Tübingen, Germany, 2011.

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

[5] Samuel Boutin. Using reflection to build efficient and certified decision procedures. In Martín Abadi and Takayasu Ito, editors, *3rd International Symposium on Theoretical Aspects of Computer Software (TACS)*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529, Sendai, Japan, 1997.

[6] William J. Cody, Jr. and William Waite. *Software Manual for the Elementary Functions*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

[7] Marc Daumas and Guillaume Melquiond. Certification of bounds on expressions involving rounded operators. *ACM Transactions on Mathematical Software*, 37(1):1–20, 2010.

[8] IEEE Computer Society. IEEE standard for floating-point arithmetic. Technical Report 754-2008, 2008.

[9] Assia Mahboubi, Guillaume Melquiond, and Thomas Sibut-Pinote. Formally verified approximations of definite integrals. *Journal of Automated Reasoning*, 2018.

[10] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, 2016.

[11] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser Basel, 2nd edition, 2018.

[12] Robert Skeel. Roundoff error cripples Patriot missile. *SIAM News*, 25(4):11, 1992.

[13] Alexey Solovyev, Charles Jacobsen, Zvonimir Rakamarić, and Ganesh Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic Taylor expansions. In Nikolaj Bjørner and Frank de Boer, editors, *20th International Symposium on Formal Methods (FM)*, volume 9109 of *Lecture Notes in Programming and Software Engineering*, pages 532–555, Oslo, Norway, 2015.

Des Théories des Types qui font de l'Effet

Pierre-Marie Pédrot

Inria Rennes-Bretagne-Atlantique & Laboratoire des Sciences du Numérique de Nantes

Résumé

Il est difficilement réfutable que la théorie des types (TdT) constitue le pinacle de la correspondance de Curry-Howard. Pour les informaticiens, elle s'avère être un langage de programmation aux types tellement riches qu'elle réalise effectivement le dicton ≪ si ça compile, ça marche ≫ parfois appliqué à des langages pourtant moins expressifs comme OCaml. Du côté mathématique, elle fournit une fondation alternative à la théorie des ensembles dans lequel le calcul est primitif, permettant d'énoncer et de formellement prouver des théorèmes dans un cadre plus naturel. Ces deux faces de la même pièce sont matérialisées par des assistants à la preuve basés sur la TdT, comme Coq, Agda ou Lean.

Pourtant, tout n'est pas rose au royaume de la correspondance preuve-programme. Ainsi, Coq est un langage fonctionnel tellement pur qu'il n'autorise pas les programmes qui bouclent. Ne parlons même pas des vrais effets qui provoqueraient un choc anaphylactique fatal chez le Haskelleux moyen, comme l'état mutable global ou les continuations de première classe. Les mathématiciens ne sont pas en reste, réclamant à cor et à cris de la logique classique, qui nécessite aujourd'hui l'ajout brutal d'axiomes cassant les bonnes propriétés calculatoires de la TdT.

Si les solutions aux problèmes exposés ci-dessus sont bien connus dans des langages plus simples comme ML, par exemple au travers d'interprétations monadiques, la complexité du typage de la TdT rend ces solutions inapplicables dans le cadre dépendant. Pire encore, la justification même non-calculatoire de certains principes de raisonnement en TdT repose sur l'existence de modèles parfaitement scabreux, qui sont souvent formulés à l'aide de structures catégoriques particulièrement incompréhensibles pour un informaticien et qui sont aux antipodes de l'intuition dynamique du λ-calcul.

Heureusement, nous ne sommes plus condamnés à cette malédiction de la pureté. Mieux encore, les solutions dont nous disposons depuis peu sont parfaitement expressibles dans un cadre orienté langage de programmation. Ce cours vise à démystifier l'existence de TdT étendue avec des primitives impures, et à propager les techniques conceptuellement simples qui permettent de les concevoir. L'outil central sur lequel elles reposent est la notion de *modèle syntaxique*, qui est l'analogue en logique du bête compilateur de grand-papa. En particulier, exeunt les catégories à poil bleu, l'ensemble des constructions présentées pouvant être décrites uniquement au travers de la TdT, un avantage pédagogique indéniable.

Nous rappellerons dans un premier temps les points précis de la théories des types qui la différencient de systèmes plus simples, ce qui nous permettra d'introduire en douceur les modèles syntaxiques et de cerner leurs spécificités et leurs points forts. La seconde partie du cours se focalisera sur la présentation de plusieurs modèles de ce genre, permettant d'étendre le comportement logique et calculatoire de la TdT, et de là argumentera sur les problèmes de cohabitation entre effets et types dépendants.

Exposés invités

Types de Session : une Abstraction pour les Protocoles de Communication

Ilaria Castellani

INRIA, Université Côte d'Azur, Sophia Antipolis, France

Les types de session décrivent des protocoles de communication entre deux ou plusieurs participants en spécifiant le séquencement des communications ainsi que la fonctionnalité de chaque communication : émetteur, récepteur, type du message échangé. Ils font partie de la famille des "types comportementaux", qui décrivent le déroulement pas à pas d'une interaction multi-parties, tout en faisant abstraction des valeurs échangées et des calculs internes des participants.

Les types de session peuvent être vus comme l'analogue, pour la concurrence et la distribution, des types de données pour le calcul séquentiel. D'abord introduits dans une variante du pi-calcul [4, 5] dans le but de spécifier abstraitement la structure des protocoles de communication et des services web, les types de session ont connu un essor important dans les deux dernières décennies, notamment dans le cadre du réseau européen BETTY [1]. Cela a donné lieu non seulement à un approfondissement et à une diversification de la théorie des types de session [6, 3], mais aussi à leur adoption dans un certain nombre de langages de programmation [2] et au développement d'outils spécifiques pour en faciliter l'usage dans la pratique [7].

Nous retracerons la genèse des types de session, illustrerons leurs principales caractéristiques et esquisserons certaines évolutions récentes de la recherche dans ce domaine.

References

[1] COST Action IC1201: Behavioural Types for Reliable Large-Scale Software Systems (BETTY), 2012-2016, http://www.dcs.gla.ac.uk/research/betty/www.behavioural-types.eu/.

[2] D. Ancona, V. Bono, M. Bravetti, G. Castagna, J. Campos, S. J. Gay, E. Giachino, E. Broch Johnsen, V. Mascardi, N. Ng, L. Padovani, P.-M. Deniélou, N. Gesbert, R. Hu, F. Martins, F. Montesi, R. Neykova, V. T. Vasconcelos, and N. Yoshida. Behavioral Types in Programming Languages. *Foundations and Trends in Programming Languages*, 3(2-3), 2016.

[3] M. Bartoletti, I. Castellani, P.-M. Deniélou, M. Dezani-Ciancaglini, S. Ghilezan, J. Pantovic, J. A. Pérez, P. Thiemann, B. Toninho, and H. Torres Vieira. Combining behavioural types with security analysis. *Journal of Logical and Algebraic Methods in Programming*, 84(6), 2015. available at https://doi.org/10.1016/j.jlamp.2015.09.003.

[4] K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *Proc. ESOP'98*, volume 1381 of *LNCS*, 1998.

[5] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *Proc. POPL'08*, pages 273–284, 2008.

[6] H. Hüttel, I. Lanese, V. T. Vasconcelos, L. Caires, M. Carbone, P.-M. Deniélou, D. Mostrous, L. Padovani, A. Ravara, E. Tuosto, H. Torres Vieira, and G. Zavattaro. Foundations of Session Types and Behavioural Contracts. *ACM Computing Surveys*, 49(1), 2016.

[7] Antonio Ravara Simon Gay, editor. *Behavioural Types: from Theory to Tools*. River Publishers, 2017. https://doi.org/10.13052/rp-9788793519817.

Les Protocoles de Déplacement de Robots : l'Algorithmique Distribuée comme Terrain de Jeu pour la Preuve Formelle

Pierre Courtieu

CNAM Paris

Résumé

L'algorithmique distribuée est communément reconnue comme l'un des domaines où la conception et surtout la certification d'algorithmes est la plus difficile. Si de surcroît les algorithmes (souvent appelés protocoles) doivent être auto-stabilisants ou résistants à la présences d'agents byzantins, la difficulté devient abyssale. Il est donc naturel qu'émergent des solutions permettant de s'assurer formellement de manière *assistée* (comprendre par du logiciel) des propriétés d'un algorithme.

Nous présenterons l'approche consistant à utiliser un assistant de preuve (Coq en l'occurrence) pour rendre l'activité de preuve non seulement correcte mais plus facile que sur papier.

Ceci a été peu essayé à notre connaissance, la communauté du distribué a depuis longtemps plutôt penché pour la solution consistant à utiliser le model checking couplé à une abstraction (cette dernière étant en général prouvée sur papier). Notre solution ne comporte pas d'abstraction et consiste à effectuer une preuve complète du protocole avec l'aide d'un assistant de preuve.

Un protocole est dit correct s'il est prouvé qu'il résoud le problème dans tous les cas voulus. Il arrivera même qu'un algorithme soit déclaré « solution » d'un problème P s'il résoud P dans tous les cas où il n'a pas été prouvé que P est impossible à résoudre. Nous montrerons des exemples de preuves de protocoles solutions, accompagnées des preuves d'impossibilité pour les cas impossibles. Dans ces dernières, l'expressivité des assistants de preuve est un atout indéniable pour prouver des propriétés de type « pour tout protocole P... »

Nous nous concentrerons principalement mais pas exclusivement sur un certain type de protocoles : les déplacement de robots dans le plan.

Session spéciale : 30 ans de JFLA

Retour sur 25 ans de programmation avec OCaml

Jean-Christophe Filliâtre

Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
INRIA Saclay – Île-de-France, Orsay, F-91893

Résumé

Le retour sur quelques communications faites aux JFLA par le passé est l'occasion pour moi d'analyser plus largement de nombreuses années de programmation avec OCaml et les raisons pour lesquelles j'affectionne ce langage. J'essayerais également d'être critique, en dégageant des aspects qui pourraient être améliorés.

Programmation synchrone aux JFLA

Guillaume Baudart[1], Louis Mandel[1], et Marc Pouzet[2]

[1] IBM Research
[2] Sorbonne Universités, UPMC Univ. Paris 06
École normale supérieure, PSL University
Inria Paris

Résumé

Depuis 1999, 39 articles et exposés liés à la programmation synchrone ont été présentés aux JFLA. Ces articles couvrent de nombreux aspects qui illustrent les liens étroits qui existent entre la programmation synchrone et les langages applicatifs : conception de langages, sémantique, typage, compilation, exécution, analyse de programmes, certification de compilateurs. Dans cet article nous revenons sur quelques uns de ces résultats qui illustrent la proximité des deux domaines.

1 Contexte historique

Les langages de programmation synchrones sont dédiés à la conception et l'implémentation de systèmes concurrents. Ils reposent sur un modèle de temps et de parallélisme idéal qui fait l'hypothèse que l'exécution du système est rythmée par une horloge globale commune à tous les processus [1].

L'appel à communication des Journées Francophones des Langages Applicatifs (JFLA) indique qu'« elles ont pour ambition de couvrir les domaines des langages applicatifs, de la preuve formelle, de la vérification de programmes, et des objets mathématiques qui sous-tendent ces outils ». Le développement des langages synchrones s'est confronté à toutes ces problématiques.

Nous avons identifié 39 articles, cours ou exposés invités présentés aux JFLA de 1999 (début des archives en ligne) à 2019 qui se rapportent à la programmation synchrone [2–40] [1]. Ces articles traitent entre autre des thèmes suivants.

Modèles et langages de programmation La conférence invitée qui ouvrait les JFLA en 1999 [2] présentait le langage Lucid Synchrone [41], un langage synchrone flot de données étendu avec des principes de ML. D'autres articles étendant ce modèle synchrone flot de données ont été présentés aux JFLA [18, 20, 29]. Nous reviendrons sur Lucid Synchrone dans la section 2. Toujours en suivant une approche flot de données, il y a eu des articles présentant des langages pour l'exécution parallèle [4, 11, 17, 23, 24], le traitement du signal [7], les circuits [25], les microcontrôleurs [33,38], ou l'analyse de logs [36]. Mais la programmation synchrone ne se limite pas à l'approche flot de données. Il y a aussi une approche plus impérative avec ReactiveML [16,21,31] et Pendulum [32] ou par réécriture avec MGS [10]. Enfin, il y a aussi des travaux plus théoriques comme ceux sur les fonctions causales [35] ou les domaines spatio-temporels [37].

Typage Les langages synchrones sont généralement des langages fortement typées (typage de données). Ils bénéficient également d'analyses de typage dédiés. Par exemple, les analyses de causalité [9] ou de réactivité [27] garantissent que les programmes produisent des données. Des analyses sur les rythmes de production et de consommation de flots peuvent garantir une exécution en mémoire bornée [18,20,22,29]. Enfin, des systèmes de types peuvent aider à générer du code efficace [14, 16].

1. Veuillez nous excusez pour les articles que nous avons oublié d'inclure dans cette liste.

Compilation et exécution La notion d'exécution concurrente est au coeur du modèle synchrone, les bibliothèques comme BSML [23] profitent de cette concurrence pour avoir une exécution parallèle. Mais cette concurrence peut aussi être exécutée séquentiellement en étant ordonnancée dynamiquement par des bibliothèques de runtime [12, 26, 28, 30]. Enfin, la concurrence peut également être compilée. Par exemple, Caspi et Pouzet [2, 18] ou Kieburtz [8] montrent comment utiliser des techniques de co-itération pour compiler des programmes flot de données en des fonctions de transitions efficaces.

Vérification et test Les langages synchrones sont souvent utilisés pour réaliser des applications dans des domaines critiques comme l'avionique où l'absence de bogues est cruciale [13]. Ces langages sont utilisés dans ce cadre car le modèle de concurrence synchrone a l'avantage d'être à la fois simple à modéliser mathématiquement et est expressif. Il peut par exemple être utilisé pour modéliser des systèmes asynchrones [6]. Ainsi, le test et la vérification formelle, qui sont aussi des thèmes important des JFLA, sont largement étudiés à la fois pour la vérification des outils et des applications.

Frédéric Gava a présenté une librairie certifiée de BSML en Coq [15]. Dans [19], nous avons prouvé en Coq la correction de l'abstraction qui est à la base du système de type qui permet de garantir qu'un programme synchrone peut s'exécuter en mémoire bornée. Timothy Bourke *et. al.*, travaillent sur la preuve formelle complète d'un compilateur Lustre [34, 40].

GATeL [5] est un outil qui permet de générer automatiquement des tests à partir de programmes Lustre. Enfin, Grégoire Hamon [3] a montré comment programmer un environnemt de simulation de programmes synchrone en utilisant un langage synchrone.

Dans cet article, nous revenons sur le mélange entre programmation synchrone et programmation fonctionnelle typée pour le développement de systèmes réactifs, et illustrons deux approches présentées dans les éditions passées des JFLA : l'intégration de traits fonctionnels dans un langage synchrone à flot-de-données, d'une part (Section 2) ; l'intégration du parallélisme synchrone dans un langage fonctionnel, d'autre part (Section 3). Chacune répond à un objectif différent. La première vise à augmenter l'expressivité et la modularité d'un langage synchrone en offrant les mêmes garanties, en particulier la possibilité de générer du code séquentiel s'exécutant en temps et mémoire bornés. La seconde, plus permissive, s'adresse à une classe plus vaste d'applications réactives, non nécessairement temps réel, en cherchant à faire cohabiter toute l'expressivité d'un langage fonctionnel, et le parallélisme synchrone. Dans cet article, nous limitons nos citations principalement aux travaux présentés aux JFLA ignorant ainsi une part importante des travaux reliés.

2 Lucid Synchrone : étendre Lustre avec des traits de ML

Le parallélisme synchrone a connu un grand succès dans la programmation des applications de contrôle/commande les plus critiques dès son introduction [42] : dans les avions (commande de vol, contrôle moteur, freinage), les trains (contrôle de bord, localisation), et les centrales (contrôle et arrêt d'urgence), etc [2]. Ces applications sont développées aujourd'hui avec SCADE [3], un environnement de développement qui s'appuie sur Scade 6, un langage synchrone flot-de-données dont le compilateur répond aux normes les plus strictes de l'avionique civile [43].

Dans un programme flot-de-données, un signal à temps discret est représenté par une suite infinie de valeurs et un système (ou schéma-bloc) est une fonction sur des suites. Le système est

2. Cf. cours de Gérard Berry (http://www.college-de-france.fr/site/gerard-berry/course-2012-2013)
3. http://www.esterel-technologies.com/products/scade-suite/

temps	0	1	2	3	4	5	6	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	...
y	y_0	y_1	y_2	y_3	y_4	y_5	y_6	...
$x + y$	$x_0 + y_0$	$x_1 + y_1$	$x_2 + y_2$	$x_3 + y_3$	$x_4 + y_4$	$x_5 + y_5$	$x_6 + y_6$	...
`pre` y	*nil*	y_0	y_1	y_2	y_3	y_4	y_5	...
$x \rightarrow y$	x_0	y_1	y_2	y_3	y_4	y_5	y_6	...
x `fby` y	x_0	y_0	y_1	y_2	y_3	y_4	y_4	...

FIGURE 1 – Les primitives flot-de-données

synchrone parce que tous les calculs sont datés par une horloge globale commune. À un dessin correspond une spécification mathématique précise sous la forme d'un ensemble d'équations. La programmation synchrone se trouve à la confluence de plusieurs courants d'idées : 1/ des questions anciennes de sémantique et les travaux de Gilles Kahn sur les réseaux de processus communicant par FIFOs ; 2/ la transcription d'automatismes du continu à l'échantillonné ; 3/ la programmation fonctionnelle, en particulier les techniques d'élimination de structures intermédiaires telles que la *déforestation* de Wadler et la modélisation des structures de données infinies. Or le langage Lustre, par exemple, est un petit langage fonctionnel manipulant des suites ; il est de premier ordre et sans récursivité. En étudiant les liens entre programmation synchrone et programmation fonctionnelle [44], nous cherchions à répondre à deux types de questions : peut-on, d'une part, faire bénéficier le temps réel de facilités de programmation des langages fonctionnels comme la synthèse des types ou l'ordre supérieur ; et, d'autre part, la programmation en général peut-elle bénéficier de l'efficacité apportée par le synchronisme ?

Le langage Lucid Synchrone [41], imaginé par Paul Caspi [45] a été conçu comme outil d'exploration de ces questions, c'est-à-dire un "laboratoire" dans lequel implémenter de nouvelles constructions de programmes ou analyses statiques [2, 44]. Des constructions nouvelles ont été explorées : la réinitialisation modulaire [46] a été le point de départ du travail de Grégoire Hamon sur l'ajout de structures de contrôle à Lustre [47] qui a abouti à une proposition pour combiner des équations flot-de-données avec des automates hiérarchiques [48]. Les analyses statiques telles que l'analyse des boucles causales réalisée par Pascal Cuoq [9, 49], le calcul d'horloges [44], exprimés tous deux sous forme d'un système de types dédiés. Puis une analyse par typage pour vérifier l'absence d'erreurs d'initialisation [50]. Toutes ces extensions ont été développées dans le cadre d'un langage synchrone fonctionnel plus expressif que Lustre, dans lequel les fonctions de suites pouvaient être des citoyens de première classe. Le langage s'est développé continûment, de 1996 à 2006, avec trois principales versions.

Exemple : le pendule inversé

Écrivons un programme qui décrit le mouvement d'une tige tenue en équilibre sur un chariot qui peut se déplacer de gauche à droite [4]. On écrit d'abord une fonction auxiliaire dans un fichier `misc.ls` : une fonction qui intègre un signal par la méthode d'Euler implicite.

```
(* module Misc *)
(* backward Euler: x(0) = x0(0); x(n) = x(n-1) + x'(n) * h(n) *)
let node backward_euler (h, x0, x') = x where
  rec x = x0 -> (pre x) +. x' *. h
```

La fonction `backward_euler (h, x0, x')` a trois arguments qui sont des suites infinies : `h` est le

4. C'est une variation du premier vrai exemple écrit avec la toute première version du compilateur, en 1996.

pas d'intégration, `x0` donne la valeur initiale de la sortie ; `x'` est le signal à intégrer. La fonction calcule la suite `x`, résultat de l'intégration de `x'` et défini par une équation de suite. **`pre`** `x` désigne la valeur de `x` à l'instant précédent. Ainsi, `x` peut être calculé séquentiellement car sa valeur à un instant n ne dépend pas d'elle-même. La figure 1 décrit les primitives de base de Lucid Synchrone.

Dans un fichier `pendulum.ls`, on peut instancier le noeud `backward_euler` pour avoir un intégrateur à pas fixe :

```
let h = 0.02 (* pas *)
let node integr(x0, xprime) = Misc.backward_euler(h, x0, xprime)
```

Le pendule est de longueur `l`, de masse `m`. Nommons `g` la gravitation, `h`, le pas d'échantillonnage et `b` le coefficient de friction entre le chariot et le pendule.

```
let l = 0.5  (* longueur du pendule *)
let m = 1.1  (* masse du pendule *)
let g = 9.8  (* gravitation *)
let b = 0.5  (* frottement *)

(* Pendulum with friction *)
let node pendulum (theta0, x'') = theta where
  rec theta = integr(theta0,
                     integr(0.0, sin (pre theta) *. g *. m -. cos (pre theta) *. x'')
                       /. l -. b *. pre theta)
```

La fonction `pendulum (theta0, x'')` calcule l'angle `theta` du pendule avec la verticale lorsque celui-ci démarre avec un angle `theta0`, une vitesse angulaire initiale nulle et est soumis à une force horizontale continue `x''` à la base du pendule.

Définissions maintenant le comportement du chariot qui maintient le pendule et sur lequel on peut transmettre les commandes `Left` ou `Right` pour appliquer une force vers la gauche ou la droite.

```
type cart_pole = { cart_position: float; pole_angle: float; }
type action = Left | Right | No

let node cart_pole (state_init, action) =
  let force = match action with Right -> 10. | Left -> -. 10. | No -> 0. end in
  let x'' = force /. m in
  let x = integr (state_init.cart_position, integr (0.0, x'')) in
  let theta = pendulum (state_init.pole_angle, x'') in
  { cart_position = x; pole_angle = theta; }
```

Comme en OCaml, on peut définir des types structurés. Le noeud `cart_pole(state_init, action)` calcule la position du chariot et l'angle du pendule à partir d'un état initial `state_init` et d'un flot de commande `action`.

Supposons que nous disposions d'une fonction `random_state` qui crée un état initial aléatoire et d'une fonction `draw` qui affiche le chariot avec le pendule. Nous pouvons alors définir une simulation paramétrée par un contrôleur qui envoie les commandes au chariot.

```
let node simulation controller =
  let state_init = random_state () in
  let rec a = run controller (state_init fby obs)
  and obs = cart_pole (state_init, a) in
  draw obs
```

h	t	t	t	t	t	t	t	...
x	x_0	x_1	x_2	x_3	x_4	x_5	x_6	...
c	t	t	f	t	f	t	t	...
x **when** c	x_0	x_1		x_3		x_5	x_6	...
z			z_0		z_1			...
merge c (x **when** c) z	x_0	x_1	z_0	x_2	z_1	x_3	x_4	...

FIGURE 2 – Les opération de filtrage et de complétion

Définissons un contrôleur manuel qui utilise les touches `'l'` et `'r'` pour émettre les actions `Left` et `Right`.

```
let node manual (key, obs) =
  match key with
  | Some 'l' -> Left
  | Some 'r' -> Right
  | _ -> No
  end
```

Nous ne pouvons pas utiliser ce contrôleur directement pour instancier le simulateur. Pour satisfaire la signature attendue, nous créons un noeud anonyme qui prend en entrée uniquement le flot d'observation de la position du pendule et utilise le noeud `get_key` le lire les touches du clavier.

```
let node main () =
  simulation (fun obs => manual (get_key(), obs))
```

Si on dispose d'un noeud `automatic` qui contrôle automatiquement le chariot. On peut définir un noeud qui peut basculer entre un mode manuel et automatique en appuyant sur la touche `'s'`.

```
let node two_modes (key, obs) = action where
  rec automaton
      | Manual ->
          do action = manual (key, obs)
          until (key = Some 's') then Auto
      | Auto ->
          do action = automatic obs
          until (key = Some 's') then Manual
      end
```

La question des horloges : de Lucid Synchrone à Lucy-n

Le programmeur Lustre verra trois différences avec son langage favori : les types n'ont pas besoin d'être écrits (cela ne surprendra pas le programmeur OCaml !), il peut combiner arbitrairement équations de suites et automates hiérarchiques, et il peut écrire des fonctions en paramètre ou en résultat d'autres fonctions. En interne, le compilateur implémente quatre systèmes de types dédiés et produit des signatures pour chacune des variables déclarées et traduit les construction de haut niveau, par exemple les automates, dans un langage noyau flot-de-donnees reposant sur le mécanisme des *horloges*. En effet, comme en Lustre, une suite peut être filtrée selon une condition booléenne pour construire une sous-suite, et deux sous-suites peuvent être combinées pour construire une suite plus longue. Les deux opérations de Lucid Synchrone sont **`when`** et **`merge`**, décrites dans la figure 2. La combinaison de suites et de sous-

suites doit cependant être contrainte. Dans un langage synchrone flot-de-données, on impose que tous les calculs soient datés par rapport à un référentiel de temps global qui bat la mesure : à toute suite s est associée une horloge h qui est une suite booléenne, de sorte que s est présent, c'est-à-dire disponible, lorsque h est vrai. Ainsi, si `x` et `c` ont une horloge h (dans la figure 2, $h(n)$ est vraie), l'horloge h' de `x when c` est vraie lorsque c est présent et vrai. Le but du calcul d'horloge est de synthétiser une expression booléenne. Celui de Lucid Synchrone, comme celui de Lustre, rejette la situation où un opérateur reçoit une entrée qui ne doit pas être présente, et réciproquement ; deux situations qui, si elles arrivaient, obligeraient à insérer un buffer avec un risque de débordement à l'exécution. Ainsi, l'expression `(x when c) + x` est rejetée statiquement. Les expression d'horloges associées à chaque expression et qui sont calculées durant le typage sont exploitées pour générer du code séquentiel efficace [51].

La notion d'horloge, et son système de type associé ont eu un rôle clef dans Lucid Synchrone pour concevoir les constructions de haut niveau, tels que les automates par exemple, définir leur sémantique et leur compilation. Toutes les constructions de haut niveau sont traduites dans un noyau flot-de-donnée avec horloges.

L'idée du n-synchrone est d'assouplir les contraintes d'horloges pour permettre de composer des suites dont les horloges ne sont pas égales mais "synchronisables" au moyen d'un buffer borné. En somme, retrouver le style de programmation des réseaux de Kahn mais en conservant la sûreté des restrictions synchrones : calculer automatiquement par typage la taille des buffers et générer du code séquentiel à temps et mémoire bornés. Par exemple, si une fonction f produit une sortie un instant sur deux, ce que l'on peut décrire par la signature suivante (écrite dans le langage Lucy-n [20]) :

```
val f :: 'a -> 'a on (1 0)
```

où (1 0) désigne la suite binaire (booléenne) périodique infinie 1 0 1 0 ..., et une fonction :

```
val g :: 'a -> 'a on (1 1 0 0)
```

qui signifie que `g` produit une sortie aux instants 1 1 0 0 .... Le programme suivant :

```
let node not_synchronous(x) = f(x) + g(x)
```

est rejeté (en Lucid Synchrone) car l'horloge `'a on (1 0)` n'est pas égale à `'a on (1 1 0 0)`. Pourtant, il suffit d'insérer un buffer qui convertit la suite d'horloge `(1 1 0 0)` en `(1 0)` pour que le programme soit accepté. Il s'écrit ainsi en Lucy-n :

```
let node nsynchronous(x) = f(x) + buffer (g(x))

val nsynchronous :: 'a -> 'a on (1 0)
```

Une fois le calcul d'horloge effectué, l'horloge effective de lecture et d'écriture de `buffer` sont déterminés ainsi que sa taille, et on obtient un programme synchrone. Les travaux sur le n-synchrone ont été présentés a plusieurs reprises aux JFLA [19,20,22] et ils ont été étendus pour gérer des flots qui peuvent produire plusieurs valeurs par instants [29]. Ces travaux reprennent de la vigueur aujourd'hui sur le sujet de la génération de code parallèle multi-coeur.

Lustre et Lucid Synchrone reposent sur un modèle de temps discret synchrone. Ils ne permettent pas d'exprimer fidèlement ni de simuler le plus efficacement possible un modèle a temps continu, ni un système mixte ou *hybride* combinant du temps discret et du temps continu. On doit écrire une version approximée des équations différentielles, sous forme d'équations de suites, fixant ainsi très tôt le pas d'échantillonnage et le schéma d'intégration. Dans les étapes préliminaires de conception et les phases de test, il est utile de décrire des modèles mixtes, de plus haut niveau, en utilisant les outils de simulation et d'analyse les plus efficaces. Un

solveur numérique d'équations différentielles tel que SUNDIALS CVODE [5], par exemple, est d'une efficacité redoutable par rapport à sa transcription synchrone à pas fixe, ce qui signifie que l'on pourra simuler des dynamiques beaucoup plus compliquées et sur une période beaucoup plus longue. Le langage Zélus [52] [6] a été conçu dans le but de pouvoir écrire des modèles mixtes exécutables. C'est essentiellement une extension d'un langage synchrone, c'est-à-dire que l'on peut y écrire des fonctions de suite comme en Lustre et Lucid Synchrone, celles-ci ayant la même sémantique statique et dynamique, et des fonctions sur des signaux à temps continu définis par des équations différentielles. Un typage ad-hoc permet de cloisonner les deux mondes et le compilateur génère du code séquentiel lié à un solveur numérique sur étagère.

3 Le synchrone dans ML : ReactiveML

Plutôt que d'intégrer des traits fonctionnels dans un langage synchrone, une approche antagoniste propose d'étendre un langage fonctionnel généraliste avec des aspects synchrones : temps logique global, composition parallèle de processus, communication par signaux, et structures de contrôle temporel. ReactiveML [16, 21] est ainsi une extension synchrone d'OCaml.

Le modèle synchrone est en effet utile à la programmation de tout type de systèmes concurrents, pas seulement les systèmes temps-réels. Cette idée a été introduite à l'origine par Frédéric Boussinot dans les langages ReactiveC [53] (extension de C) et SugarCubes [54] (extension de Java).

Comparé à ces prédécesseurs, ReactiveML étend un langage fonctionnel. La simplicité de la sémantique du langage hôte a permis de formaliser les interactions entre les parties algorithmiques et réactives et ainsi d'avoir des processus et de signaux comme valeurs de première classe [16]. ReactiveML hérite de l'inférence de type d'OCaml qu'il étend pour garantir que les programmes ne contiennent pas de boucle instantanées. Cette analyse de réactivité [27] permet de vérifier statiquement que tous les instants logiques terminent. Enfin, le moteur d'exécution de ReactiveML repose sur l'utilisation de continuations [30].

ReactiveML a été utilisé pour la programmation d'applications variées [31,39] : simulateur de réseaux de capteurs, simulateur de système complexes, programmation musicale, art, interface interactive pour le jeux d'échecs, et interface conversationnelle [39].

Exemple : Les Boids

Introduits par Reynolds [55], les boids permettent de simuler le comportement de groupes d'animaux tels que les bancs de poissons ou les nuées d'oiseaux. Le comportement de chaque individu est modélisé par trois règles :

— cohésion : les boids s'attirent pour former des nuées.

— séparation : les boids se repoussent pour éviter les collisions.

— alignement : les boids alignent leur vitesse pour avancer dans la même direction.

Chacune de ces règles est associée à un champ de vision différent. Un boid repousse ses plus proches voisins, s'aligne sur les membres de son groupe, et tente de rejoindre les boids plus éloignés. Malgré la relative simplicité de ce modèle, l'interaction des boids permet de faire émerger des comportements de groupe convaincants.

La Figure 3 présente le cœur de l'implémentation des boids en ReactiveML. Un boid est caractérisé par trois vecteurs (position, vitesse et accélération) et trois signaux qui correspondent

5. https://computation.llnl.gov/projects/sundials/cvode
6. http://zelus.di.ens.fr

```
type boid =
    { id: int;
      position: vect;
      speed: vect;
      acceleration: vect;
      s_separation: (boid * boid, vect) event;
      s_cohesion: (boid * boid, vect) event;
      s_alignment: (boid * boid, vect) event; }

let rec process boid me flock =
  emit flock me;
  await me.s_separation(f_separation)
     /\ me.s_cohesion(f_cohesion)
     /\ me.s_alignment(f_alignment) in
  let next_me = next_position me
    f_repulsion f_separation f_cohesion f_alignment in
  run boid next_me flock

let process main =
  signal flock, key default [] gather (fun x y -> x :: y) in
  do
    run window flock key ||
    control
      for i=1 to boids_number dopar
        run boid (new_random_boid ()) flock
      done ||
      loop
        await flock (all) in
        dispatcher all
      end
    with key(['p']) done
  until key(['q']) done
```

FIGURE 3 – Les Boids en ReactiveML

à chacune des forces qui s'appliquent sur le boid (séparation, cohésion et alignement). Les boids peuvent donc être représentés par le type enregistrement `boid`.

Le type des signaux de forces est `(boid * boid, vect) event`. Ce type signifie que l'on doit émettre des paires de boids sur le signal et que l'on va lire en réception du signal un vecteur représentant la force à appliquer. Les fonctions de transformation des pairs en boids en vecteurs de force est définie lors de la déclaration des signaux. Par exemple, la déclaration du signal `s_separation` est la suivante.

```
signal separation
    default vzero
    gather (fun (me, boid) f_sep ->
      let sep = boid.position -: me.position in
      f_sep -: sep)
```

La valeur `vzero` est le vecteur $(0, 0)$. La fonction **`gather`** définie comment lors d'un instant combiner chaque valeur émise avec la valeur courante du signal. Ici, le valeur émise doit être

la pair `(me, boid)` où `me` est le boid sur lequel la force doit être appliquée et `boid` est un de ses voisins. `f_sep` est la valeur courante du signal en construction (c'est-à-dire `vzero` avant la première émission pendant l'instant, puis le résultat de l'appel de la fonction lors de le précédente émission). Le corps de la fonction calcule le vecteur `sep` entre les deux boids (`-:` est la différence entre deux vecteurs) et soustrait cette valeur à `f_sep`.

Le comportement d'un boid est implémenté par le processus `boid` (Figure 3). Un boid émet son état (`me`) sur le signal global `flock` puis récupère la valeur des signaux de force[7]. Il peut alors calculer la position suivante avant de relancer récursivement le processus avec le nouvel état pour le prochain pas de simulation.

La fonction de simulation `main` définie les signaux globaux `flock` et `key` et se décompose en trois parties exécutées en parallèle (opérateur `||`). 1/ Le processus `window` gère l'affichage graphique de la nuée en récupérant les positions des boids sur le signal `flock` et émet le signal `key` lorsqu'une touche du clavier est enfoncée. 2/ Une boucle **`for/dopar/done`** qui exécute en parallèle `boids_numbers` processus `boid`. 3/ Une boucle infinie (**`loop/end`**) qui à chaque émission de valeurs sur le signal `flock`, exécute la fonction `dispatcher` sur l'ensemble de la nuée. Cette fonction calcule les paires de boids qui sont à portée de vue et émet ces paires sur les signaux de force des boids concernés.

Le corps du processus `main` est englobé dans une construction de préemption (**`do/until`**). Ainsi, lorsque le signal `key` est émis avec la valeur `['q']` le programme se termine. La construction **`control/with`**, qui englobe les boids et la fonction de dispatche, suspend et reprend l'exécution de son corps lorsque le signal `key` est émis avec la valeur `['p']`. Cela permet de mettre la simulation en pause.

Un des points singulier de ReactiveML est la création dynamique de processus qui est possible par le mélange de récursion et de composition parallèle. Illustrons cela avec un processus `add_boids` qui crée un nouveau boid à chaque fois qu'un signal `new_position` est émis.

```
let rec process add_boid new_position flock =
  await new_position (position) in
  run add_boid new_position flock ||
  await immediate flock;
  run boid (new_boid position) flock
```

Ce processus est donc paramétré par le signal `new_position` qui déclenche la création d'un nouveau boid et le signal `flock` sur lequel le nouveau boid va communiquer avec les autres boids. C'est un processus récursif qui attend le signal `new_position` pour ensuite en parallèle exécuter un nouveau boid (on attend l'émission du signal `flock` avant d'exécuter `boid` pour s'assurer qu'il n'y a pas de décalage de phase entre les boids) et faire un appel récursif pour se mettre en attente à nouveau sur le signal `new_position`.

4 Aspects dynamiques dans les langages synchrones

On peut se poser la question de savoir si Lucid Synchrone et ReactiveML sont des langages si différents. Premièrement, nous avons vu qu'ils sont de natures différentes : Lucid Synchrone est un langage flot de données alors que ReactiveML est fondée sur Esterel qui est impératif. Mais l'ajout d'automates à Lucid Synchrone a rapproché les deux modèles comme cela a été illustré par Scade 6.

L'exemple du processus `add_boid` a montré que l'on peut créer dynamiquement des processus en ReactiveML. On peut se demander s'il est possible de faire la même chose en Lucid

7. **`await`** `s(x)` **`in`** `e` lie la valeur du signal `s` à la variable `x` dans l'expression `e`

```
Process INTEGERS out QO;
     Vars N ; 1 -> N ;
     repeat INCREMENT N ; PUT(N, QO) forever
Endprocess ;
Process FILTER PRIME in QI out QO ;
     Vars N ;
     repeat GET(QI) -> N ;
            if (N MOD PRIME) <> 0 then PUT(N, QO) close
     forever
Endprocess ;
Process SIFT in QI out QO ;
     Vars PRIME ; GET(QI) -> PRIME ;
     PUT (PRIME, QO) ; Comment emit a discovered prime ;
     doco queues Q ;
       FILTER(PRIME, QI, Q) ; SIFT(Q, QO)
     closeco
Endprocess ;
Process OUTPUT in QI ; Comment this is a library process ;
     repeat PRINT(GET(QI))
Endprocess ;
start doco queues Q1 Q2 ;
     INTEGERS(Q1) ; SIFT(Q1,Q2) ; OUTPUT(Q2)
     closeco ;
```

FIGURE 4 – Crible d'Érathosthène de Kahn et MacQueen [56]

Synchrone ? Lucid Synchrone est fondé sur des idées des réseaux de Kahn, or dès 1976 Kahn et MacQueen donnait un exemple de réseau avec création dynamique [56]. Il s'agit du crible d'Érathosthène qui est reproduit Figure 4 où le processus `SIFT` utilise la composition parallèle et la récursion pour créer dynamiquement des processus. Lucid Synchrone autorise aussi la récursion, il est donc possible de réécrire ce programme :

```
let node integers () = n where rec n = 1 fby n + 1

let rec node sift n =
  let rec prime = n fby prime in
  let clock filter = (n mod prime) <> 0 in
  merge filter (sift (n when filter)) (true fby false)

let node output n = print_int n; print_string " "; flush stdout

let node main () =
  let n = integers () in
  let clock primes = sift n in
  merge primes (output (n when primes)) ()
```

Ainsi, même si Lucid Synchrone s'intéresse principalement à la conception de systèmes temps-réel, il ne se limite pas à ces derniers [8]. Néanmoins, on peut constater que même si le programme grossi dynamiquement, toutes les dépendances sont connues statiquement.

8. L'option `-realtime` du compilateur garantie l'exécution en temps et mémoire bornés.

En ReactiveML, les signaux sont des valeurs de première classe. Il est donc possible de changer dynamiquement les dépendances dans les programmes. On peut par exemple simuler la mobilité du π-calcul. Reprenons l'exemple du chapitre 9.3 de [57].

```
let process p x z = emit x z; run p'
let process q x = await x(y) in run q' y
let process r z = await z(v) in run r' v
let process mobility x z = run p x z || run q x || run r z
```

Trois processus `p`, `q` et `r` sont exécutés en parallèle. D'une part `p` et `q` peuvent communiquer en utilisant un signal `x` et d'autre part `p` et `r` communiquent par le signal `z`. Les processus `p` et `q` peuvent être définis de telle sorte que `q` et `r` puissent communiquer en utilisant `z`.

Ce type de dynamicité n'est pas possible en Lucid Synchrone. Mais il est possible d'étendre le langage avec une notion de canaux partagés qui sont des valeurs de première classe. Ces canaux ont des propriétés similaires aux signaux de ReactiveML. Lorsqu'ils sont déclarer, il faut associer une fonction indiquant comment combiner plusieurs écritures pendant le même instant.

```
share x default 0 gather (+)
```

La lecture de la valeur d'un canal partagé se fait toujours à travers un retard (**`last`** `!x`). Enfin, comme les canaux partagés sont des valeurs de première classe, ils peuvent échapper à leur portée. Ainsi, l'horloge du contenu des canaux est toujours l'horloge de base du programme pour pouvoir faire des écritures et lectures à tout instant.

Enfin, le dernier aspect dynamique est la suppression de processus. De part la nature impérative du langage et son runtime interprété, la suppression de processus en ReactiveML est très naturelle. Lorsqu'un processus termine, son contrôle n'est plus accessible et ses ressources peuvent être désallouées. En Lucid Synchrone, tous les flots sont infinis. Il est possible de simuler la terminaison avec un flot dont l'horloge devient fausse pour toujours mais cette condition ne peut pas être testée dynamiquement. Par ailleurs, le compilateur actuel de Lucid Synchrone, le code généré est ordonnancé statiquement. Cela crée du code très efficace, mais il est difficile de libérer toutes les ressources liées aux processus qui terminent.

5 Conclusion

Lucid Synchrone et ReactiveML sont deux exemples de langages synchrones fonctionnels combinant, chacun différemment, le parallélisme synchrone et les traits d'un langage *a la ML*. Ils se sont développés au moment où apparaissaient plusieurs langages embarqués en Haskell, notamment Lava [58] pour les circuits synchrones, Fran [59] puis FRP [60] pour la programmation de systèmes réactifs et des mécanismes pour plonger un langage dédié dans ML et le compiler par macro-génération [61]. Nous avons suivi une voie différente pour mieux maîtriser les programmes synchrones devant être acceptés — détecter par typage les programmes non réactifs ou ne respectant pas les constraintes d'horloges, par exemple — et générer du code efficace en exploitant les propriétés de la sémantique statique. ReactiveML a été utilisé dans plusieurs domaines inattendus, tels que la musique mixte [62]. De nombreuses constructions et principes introduits dans Lucid Synchrone ont été repris dans SCADE. Et plus récemment, dans le langage Stimulus [9] pour la simulation des exigences de haut niveau. Zélus est directement issu des travaux sur Lucid Synchrone et ReactiveML. Les travaux actuels portent sur l'extension probabiliste d'un langage synchrone et sa mise en oeuvre dans Zélus.

9. https://www.argosim.com

Remerciements Nous souhaitons remercier les organisateurs des JFLA de nous inviter à la trentième édition de la conférence. C'est un honneur de pouvoir y participer. Cela a toujours été un grand plaisir soumettre et venir à cette conférence : la qualité des rapports fournit des conseils qui nous permettent d'améliorer grandement nos articles, la qualité scientifique des exposés sont source d'inspiration, l'ambiance conviviale de la conférence en fait une raison supplémentaire de vouloir y revenir. Nous souhaitons également remercier tous nos coauteurs sans qui nous n'aurions pas pu être invité et avec qui cela a été un grand plaisir de travailler.

Références

[1] G. Berry. Real time programming : Special purpose or general purpose languages. *Information Processing*, 89 :11–17, 1989.

[2] Paul Caspi. Lucid synchrone. In Weis [72]. Conférence invitée.

[3] Grégoire Hamon and Marc Pouzet. Un simulateur synchrone pour Lucid Synchrone. In Weis [72].

[4] Frédéric Loulergue. Extension du BSlambda-calcul. In Weis [72].

[5] Bruno Marre and Agnès Arnould. Génération automatique de séquences de tests à partir de descriptions Lustre : GATeL. In Dubois [71].

[6] Frédéric Boniol, Gérard Bel, and Jack Foiseau. Modélisation et vérification de systèmes intégrés asynchrones dans le langage synchrone Lustre : application aux systèmes avioniques. In Dubois [71].

[7] Jocelyn Serot. Un compilateur CAML → SYNDEX pour les applications de traitement de signal temps-réel distribués. In Dubois [71].

[8] Richard Kieburtz. Coalgebraic techniques for reactive functional programming. In Dubois [71]. Conférence invitée.

[9] Pascal Cuoq and Marc Pouzet. Causalité modulaire dans un langage de flots synchrone. In Pierre Castéran, editor, *Douzièmes Journées Francophones des Langages Applicatifs (JFLA 2001)*, Pontarlier, France, January 2001. INRIA.

[10] Olivier Michel, Jean-Louis Giavitto, and Julien Cohen. MGS : transformer des collections complexes pour la simulation en biologie. In Rideau-Gallot [70].

[11] Armelle Merlin and Gaétan Hains. La machine abstraite catégorique BSP. In Rideau-Gallot [70].

[12] Julien Cohen, Olivier Michel, and Jean-Louis Giavitto. Filtrage et règles de réécriture sur des structures indexées par des groupes. In Filliatre [69].

[13] Y. Ait Ameur, F. Boniol, S. Pairault, and V. Wiels. Analyse de robustesse de systèmes avioniques. In Filliatre [69].

[14] Frédéric Gava and Frédéric Loulergue. Synthèse de types pour Bulk Synchronous Parallel ML. In Filliatre [69].

[15] Frédéric Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. In Valérie Ménissier-Morain, editor, *Quinzièmes Journées Francophones des Langages Applicatifs (JFLA 2004)*, Sainte-Marie-de-Ré, France, January 2004. INRIA.

[16] Louis Mandel and Marc Pouzet. ReactiveML, un langage pour la programmation réactive en ML. In Olivier Michel, editor, *Seizièmes Journées Francophones des Langages Applicatifs (JFLA 2005)*, Obernai, France, March 2005. INRIA.

[17] R. Benheddi and F. Loulergue. Sémantiques de MSPML avec composition parallèle. In Thérèse Hardin, editor, *Dix-septièmes Journées Francophones des Langages Applicatifs (JFLA 2006)*, Pauillac, France, January 2006. INRIA.

[18] Marc Pouzet. Programmation synchrone fonctionnelle. In Pierre-Etienne Moreau, editor, *Dix-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2007)*, Aix-les-Bains, France, January 2007. INRIA. Cours.

[19] Louis Mandel and Florence Plateau. Abstraction d'horloges dans les systèmes synchrones flot de données. In Alan Schmitt, editor, *Vingtièmes Journées Francophones des Langages Applicatifs (JFLA 2009)*, Saint-Quentin sur Isère, France, January 2009. INRIA.

[20] Louis Mandel, Florence Plateau, and Marc Pouzet. Lucy-n : une extension n-synchrone de Lustre. In Mayero [68].

[21] Louis Mandel. Cours de ReactiveML. In Mayero [68].

[22] Louis Mandel and Florence Plateau. Typage des horloges périodiques en Lucy-n. In Conchon [67].

[23] Frédéric Gava and Sovanna Tan. Implémentation et prédiction des performances de squelettes data-parallèles en utilisant un langage BSP de haut niveau. In Conchon [67].

[24] Wadoud Bousdira, Frederic Loulergue, and Louis Gesbert. Syntaxe et sémantique de revised Bulk Synchronous Parallel ML. In Conchon [67].

[25] Thomas Braibant. De coquets circuits. In Conchon [67].

[26] Christophe Deleuze. Concurrence et continuations en OCaml. In Assia Mahboubi, editor, *Vingt troisièmes Journées Francophones des Langages Applicatifs (JFLA 2012)*, Carnac, France, January 2012.

[27] Louis Mandel and Cédric Pasteur. Réactivité des systèmes coopératifs : le cas de ReactiveML. In Pous [66].

[28] Christophe Deleuze. Concurrence légère en OCaml : muthreads. In Pous [66].

[29] Adrien Guatto and Louis Mandel. Réseaux de Kahn à rafales et horloges entières. In Tasson [65].

[30] Louis Mandel and Cédric Pasteur. Exécution efficace de programmes ReactiveML. In Tasson [65].

[31] Louis Mandel and Marc Pouzet. ReactiveML, un langage de programmation réactif. In Tasson [65]. Exposé invité parmi les contributions marquantes des dix dernières années).

[32] Rémy El Sibaïe and Emmanuel Chailloux. Pendulum : une extension réactive pour la programmation Web en OCaml. In David Baelde, editor, *Vingt-sixièmes Journées Francophones des Langages Applicatifs (JFLA 2015)*, Val d'Ajol, France, January 2015.

[33] S. Varoumas, B. Vaugon, and E. Chailloux. OCaLustre : une extension synchrone d'OCaml pour la programmation de microcontrôleurs. In Signoles [64].

[34] T. Bourke, P.-É. Dagand, M. Pouzet, and L. Rieg. Vérification de la génération modulaire du code impératif pour Lustre. In Signoles [64].

[35] B. P. Serpette et D. Janin. Causalité dans les calculs d'événements. In Signoles [64].

[36] Guillaume Baudart, Louis Mandel, Olivier Tardieu, and Mandan Vaziri. Cloudlens, un langage de script pour l'analyse de données semi-structurées. In Signoles [64].

[37] David Janin. Domaines spatio-temporels : un tour d'horizon. In Boldo [63].

[38] Steven Varoumas, Benoit Vaugon, and Emmanuel Chailloux. La programmation de microcontrôleurs dans des langages de haut niveau. In Boldo [63]. Cours.

[39] Guillaume Baudart, Louis Mandel, and Jérôme Siméon. Programmer des chatbots en OCaml avec Watson Conversation Service. In Boldo [63].

[40] Timothy Bourke and Marc Pouzet. Arguments cadencés dans un compilateur Lustre vérifié. In Nicolas Magaud, editor, *Trentièmes Journées Francophones des Langages Applicatifs (JFLA 2019)*, Les Rousses, France, January 2019. INRIA.

[41] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual.* Université Paris-Sud, LRI, April 2006.

[42] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[43] 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.

[44] Paul Caspi et Marc Pouzet. Réseaux de Kahn Synchrones. In *Journées Francophones des Langages*

Applicatifs (JFLA), Val Morin (Québec), Canada, 28-30 janvier 1996.

[45] Paul Caspi. Lucid synchrone. In *Actes du colloque INRIA OPOPAC*, Lacanau, 1993.

[46] Grégoire Hamon and Marc Pouzet. Modular Resetting of Synchronous Data-flow Programs. In *ACM International conference on Principles of Declarative Programming (PPDP'00)*, Montreal, Canada, September 2000.

[47] Grégoire Hamon. *Calcul d'horloge et Structures de Contrôle dans Lucid Synchrone, un langage de flots synchrones à la ML.* PhD thesis, Université Pierre et Marie Curie, Paris, France, 14 novembre 2002.

[48] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.

[49] Pascal Cuoq. *Ajout de synchronisme dans les langages fonctionnels typés.* PhD thesis, Université Pierre et Marie Curie, Paris, France, october 2002.

[50] Jean-Louis Colaço and Marc Pouzet. Type-based Initialization Analysis of a Synchronous Data-flow Language. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(3) :245–255, August 2004.

[51] Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.

[52] Timothy Bourke and Marc Pouzet. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems : Computation and Control (HSCC 2013)*, Philadelphia, USA, April 8–11 2013. ACM.

[53] Frédéric Boussinot. Reactive C : An extension of C to program reactive systems. *Software Practice and Experience*, 21(4) :401–428, April 1991.

[54] Frédéric Boussinot and Jean-Ferdy Susini. The SugarCubes tool box : A reactive java framework. *Software Practice and Experience*, 28(4) :1531–1550, 1998.

[55] Craig Reynolds. Flocks, herds and schools : A distributed behavioral model. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87. ACM, 1987.

[56] Gilles Kahn and David Macqueen. Coroutines and Networks of Parallel Processes. Research report, 1976.

[57] Robin Milner. *Communicating and Mobile Systems : The π-Calculus.* 1999.

[58] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava : Hardware Design in Haskell. In *International Conference on Functional Programming (ICFP).* ACM, 1998.

[59] C. Elliot and P. Hudak. Functional reactive animation. In *International Conference on Functional Programming (ICFP).* ACM, 1997.

[60] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *International Conference on Programming Language, Design and Implementation (PLDI)*, 2000.

[61] Walid Taha. Multi-stage programming : Its theory and applications. Technical Report CSE-99-TH-002, Oregon Graduate Institute of Science and Technology, November 1999.

[62] Louis Mandel, Cédric Pasteur, and Marc Pouzet. ReactiveML, Ten Years Later. In *ACM International Conference on Principles and Practice of Declarative Programming (PPDP)*, Siena, Italy, July 2015. Invited paper for PPDP'05 award.

[63] Sylvie Boldo, editor. *Vingt-neuvièmes Journées Francophones des Langages Applicatifs (JFLA 2018)*, Banyuls-sur-Mer, France, January 2018. INRIA.

[64] Julien Signoles, editor. *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA 2017)*, Gourette, France, January 2017. INRIA.

[65] Christine Tasson, editor. *Vingt-cinquièmes Journées Francophones des Langages Applicatifs (JFLA 2014)*, Fréjus, France, January 2014.

[66] Damien Pous, editor. *Vingt-quatrièmes Journées Francophones des Langages Applicatifs (JFLA 2013)*, Aussois, France, February 2013.

[67] Sylvain Conchon, editor. *Vingt deuxièmes Journées Francophones des Langages Applicatifs (JFLA 2011)*, La Bresse, France, January 2011.

[68] Micaela Mayero, editor. *Vingt et unièmes Journées Francophones des Langages Applicatifs (JFLA 2010)*, Vieux-Port La Ciotat, France, January 2010.

[69] Jean-Christophe Filliatre, editor. *Quatorzièmes Journées Francophones des Langages Applicatifs (JFLA 2003)*, Chamrousse, France, January 2003. INRIA.

[70] Laurence Rideau-Gallot, editor. *Treizièmes Journées Francophones des Langages Applicatifs (JFLA 2002)*, Anglet, France, January 2002. INRIA.

[71] Catherine Dubois, editor. *Onzièmes Journées Francophones des Langages Applicatifs (JFLA 2000)*, Mont Saint-Michel, France, January 2000. INRIA.

[72] Pierre Weis, editor. *Dixièmes Journées Francophones des Langages Applicatifs (JFLA 1999)*, Morzine-Avoriaz, France, February 1999. INRIA.

Sémantiques Formelles et Certifiées

Alan Schmitt

Inria

Résumé

Les sémantiques de langages de programmation peuvent être particulièrement complexes. Un exemple est la sémantique de JavaScript, dont la taille est conséquente et dont les règles sont massivement interdépendantes les unes des autres. Prouver des propriétés sur de telles sémantiques, comme la correction d'analyses, nécessite un effort conséquent. Nous travaillons depuis plusieurs années sur des techniques permettant de maîtriser la complexité de ces sémantiques. En particulier, nous avons appliqué l'approche Pretty Big Step développée par Charguéraud à une sémantique mécanisée de JavaScript ainsi qu'à des manières systématiques de prouver la correction de sémantiques abstraites. Plus récemment, nous avons développé un langage très simple de description de sémantiques, appelé sémantique squelettique, permettant de significativement réduire les efforts de preuves en mettant en avant la structure commune partagée par de nombreuses sémantiques.

Articles

Suspension et Fonctorialité : Deux Opérations Implicites Utiles en CaTT

Thibaut Benjamin and Samuel Mimram

École Polytechnique

Résumé

La notion de ω-catégorie faible est une vaste généralisation de celle de catégorie, qui revêt une importance croissante en mathématiques. Cependant, les définitions qui en ont été proposées à ce jour à ce jour restent difficiles à manipuler en pratique : on ne connaît pas de liste explicite des cohérences qui doivent être présentes dans ces structures et il devient rapidement difficile de vérifier tous les détails qui doivent l'être lors de leur utilisation. Partant de ce constat, Finster et Mimram ont récemment reformulé la définition de Grothendieck et Maltsiniotis des ω-catégories faibles sous forme d'une théorie des types, appelée CaTT. Ceci a posé les bases d'un assistant de preuve, développé par les auteurs, dans lequel les termes typables sont ceux qui correspondent à des cohérences. La formalisation de preuves non-triviales dans ce système a montré la nécessité d'avoir des outils permettant de les automatiser en partie. Dans cette optique, nous présentons ici deux méthodes permettant de générer automatiquement des cohérences de dimension supérieure à partir de cohérences existantes : la suspension et la fonctorialité. Nous montrons qu'elles sont admissibles (c'est-à-dire que leur présence ne modifie pas le pouvoir expressif de la théorie des types) et qu'elles sont utiles en pratique pour alléger les preuves en évitant des redondances.

1 Introduction

Au cours des dernières décennies, les catégories se sont imposées comme un cadre unificateur en mathématiques. Cependant, afin de pouvoir prendre en compte les situations rencontrées lors de l'étude de structures, il apparaît nécessaire de généraliser cette notion de catégorie selon deux axes. Dans un premier temps, on souhaite considérer non seulement les objets et les morphismes, mais aussi les morphismes entre morphismes (appelés 2-cellules), les morphismes entre 2-cellules (appelés 3-cellules), etc., ainsi que leurs compositions, aboutissant à la notion de catégorie supérieure stricte, aujourd'hui bien connue et étudiée. Dans un second temps, il se trouve que l'axiomatique de ces structures est souvent trop contraignante au vu des situations rencontrées et l'on souhaite la relâcher. Typiquement, au lieu d'imposer que la composition de trois n-cellules f, g, h soit associative, on va demander l'existence d'une $(n+1)$-cellule

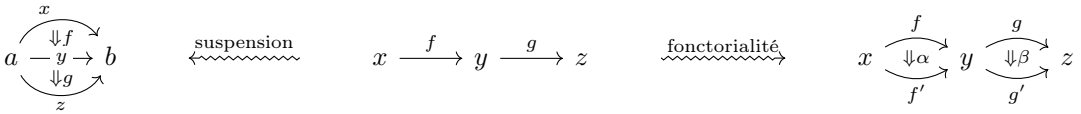
$$\alpha \quad : \quad h \circ (g \circ f) \quad \to \quad (h \circ g) \circ f$$

que l'on verra comme un « témoin » de l'associativité de la composition. De plus, on souhaite que ce choix de témoins soit cohérent, c'est-à-dire que deux témoins canoniques sont toujours reliés par une $(n+2)$-cellule qui témoigne de leur égalité, ces témoins devant eux-mêmes être cohérents, etc. La structure résultante est appelée *ω-catégorie faible* et on imagine sans peine la difficulté de l'axiomatiser complètement, en fournissant une liste exhaustive de tous les morphismes, appelés *cohérences*, qu'une telle catégorie doit nécessairement contenir.

Plusieurs définitions ont été proposées pour ces structure. En particulier, une définition satisfaisante a été introduite par Grothendieck [4] pour les ω-groupoïdes (dans lesquels toutes les cellules sont inversibles) et généralisée par Maltsiniotis [5] afin d'obtenir une définition des

ω-catégories. Celle-ci a récemment été reformulée par Finster et Mimram [3] sous la forme d'une théorie des types. Dans celle-ci, un terme correspond à une cohérence valide précisément lorsque le terme est typable ; autrement dit, les *modèles* de cette théorie des types (dans lesquels on interprète un type par un ensemble, un terme par un élément de l'interprétation de son type, etc.) correspondent précisément aux ω-catégories. De façon générale, l'idée de cette théorie est d'introduire des types correspondant aux ensembles de morphismes ($\star$ est le type des 0-cellules et si f et g sont des n-cellules de même source et même but alors $f \to g$ est le type des $(n+1)$-cellules de f vers g). Un contexte peut alors être vu comme une famille de cellules formelles et les règles expriment essentiellement qu'une famille « composable » admet une composée. En particulier, les opérations que l'on trouve habituellement dans des catégories (les compositions, les identités) ne sont pas posées comme structure mais sont des cas particuliers de cohérences que l'on peut déduire des règles.

Des implémentations de cette théorie des types ont été réalisées [3, 2] et la formalisation partielle d'exemples conséquents (comme la définition des syllepses [1]) ont fait ressortir la difficulté de mener à bien des preuves dans la pratique. Notre conviction est que ceci n'est pas dû au système de types lui-même mais à l'absence de support de haut niveau dans l'assistant de preuve : il apparaît que de nombreuses définitions ou détails des preuves pourraient être inférés automatiquement. Dans cette optique, nous étudions ici deux familles d'opérations qui permettent de construire automatiquement des termes à partir de termes préexistants : la suspension et la fonctorialité. Celles-ci sont fondées sur l'observation que dès que l'on définit des opérations sur des n-cellules alors on peut (de deux façons différentes) généraliser ces opérations à des $(n+1)$-cellules. Ainsi, par exemple, étant données de deux 1-cellules composables f et g comme dans le diagrammes du milieu, on peut définir une cohérence qui correspond à leur composition $g \circ f$.



Une fois celle-ci définie, par suspension, le système va automatiquement aussi définir la composée verticale de deux 2-cellules (diagramme de gauche ci-dessus). De même, par fonctorialité, nous allons pouvoir déduire de la composition de 1-cellules, la composition horizontale de 2-cellules (diagramme de droite). Ces opérations sont implémentées dans notre prototype d'assistant de preuve [2] et dans la pratique, ces constructions permettent de réduire de façon importante la taille des preuves en en automatisant des parties.

Dans cet article, nous commençons par rappeler la théorie des types CaTT(Section 2), puis nous définissons les opérations de suspension (Section 3) et de fonctorialité (Section 4), et esquissons finalement en conclusion une approche générale de ce type d'opérations (Section 5).

2 Théorie des types pour les ω-catégories faibles

Dans cette section, nous présentons la théorie des types, appelée CaTT, proposée par Finster et Mimram [3] : les termes dérivables dans théorie correspondent aux cohérences présentes dans toute ω-catégorie ; autrement dit, les modèles de cette théorie sont précisément les ω-catégories. Dans la suite de l'article toutes les ω-catégories considérées sont implicitement supposées faibles. Tout d'abord, nous introduisons une théorie dont les modèles sont les ω-graphes, et la théorie complète est obtenue en enrichissant celle-ci. La présentation donnée ici se veut introductive, et nous enjoignons le lecteur avide de détails à lire l'article [3].

2.1 Théorie des types pour les ω-graphes

ω-graphes. De même qu'une catégorie peut être vue comme un graphe (avec les objets pour sommets et les morphismes pour arêtes) muni de compositions satisfaisant des axiomes, une ω-catégorie consiste en un ω-graphe muni d'opérations (les cohérences). Par un *ω-graphe* (aussi appelé ensemble globulaire), on entend ici une famille $(C_n)_{n\in\mathbb{N}}$ d'ensembles, les éléments de C_n étant appelés *n-cellules*, l'entier n étant leur *dimension*. Pour $n > 0$, les n-cellules sont munies d'une source et d'un but qui sont des $(n-1)$-cellules qui ont elles-mêmes même source et même but. On a par exemple, le ω-graphe suivant avec $C_0 = \{x, y, z\}$, $C_1 = \{f, f', f'', g\}$, $C_2 = \{\alpha, \beta\}$ et $C_n = \emptyset$ pour $n > 2$:

$$\begin{tikzcd} x \arrow[r, bend left=60, "f"] \arrow[r, "f'" description] \arrow[r, bend right=60, "f''"'] \arrow[r, phantom, shift left=3, "\Downarrow\alpha"] \arrow[r, phantom, shift right=3, "\Downarrow\beta"] & y \arrow[r, "g"] & z \end{tikzcd} \tag{1}$$

la source de α étant f et son but f', etc. Comme suggéré par la représentation graphique ci-dessus, on appellera parfois *diagramme* un ω-graphe fini.

La théorie. Nous allons décrire une théorie des types ayant pour modèles les ω-graphes. Celle-ci fera intervenir des variables (x, y, ...), des termes (t, u, ...), des types (A, B, ...), des contextes (Γ, Δ, ...) et des substitutions (σ, τ, ...). Comme attendu, un *contexte* est une liste $\Gamma = (x_1 : A_1, \ldots, x_n : A_n)$ de variables x_i munies de types A_i (on note $\varnothing$ le contexte vide) et une *substitution* $\sigma = \langle t_1, \ldots, t_n\rangle$ est une liste de termes t_i (on note $\langle\rangle$ la substitution vide et $\langle\sigma, t\rangle$ l'ajout d'un terme t à la fin d'une substitution σ). Notre théorie consistera en un ensemble de règles d'inférence permettant de dériver l'une des quatre formes de jugements suivantes, dont nous précisons la signification :

$\Gamma \vdash$	Le contexte Γ est bien formé
$\Gamma \vdash A$	Le type A est bien formé dans le contexte Γ
$\Gamma \vdash t : A$	Le terme t a pour type A dans le contexte Γ
$\Gamma \vdash \sigma : \Delta$	La substitution σ a pour type Δ dans le contexte Γ

On note $\mathrm{Var}(t)$ (resp. $\mathrm{Var}(A)$, $\mathrm{Var}(\Gamma)$, $\mathrm{Var}(\sigma)$) l'ensemble des *variables libres* d'un terme t (resp. type A, contexte Γ, substitution σ), définies de la façon attendue. Une variable qui n'est pas libre dans un terme t (resp. ...) est dite *liée* dans ce terme (resp. ...). La théorie comporte deux constructeurs de type notés $\star$ et $t \underset{A}{\longrightarrow} u$, ce dernier dépendant d'un type A et de deux termes t et u et aucun constructeur de terme (les seuls termes seront donc les variables, bien sûr cela ne sera plus le cas lorsque nous nous intéresserons aux ω-catégories au lieu des ω-graphes). Les règles d'inférence sont les suivantes.

Termes :
$$\frac{\Gamma, x : A \vdash}{\Gamma, x : A \vdash x : A}\,(\mathrm{AX}) \qquad \frac{\Gamma \vdash t : B}{\Gamma, x : A \vdash t : B}\,(\mathrm{WK})$$

Types :
$$\frac{}{\Gamma \vdash \star}\,(\mathrm{OBJ}) \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t \underset{A}{\longrightarrow} u}\,(\mathrm{HOM})$$

Contextes :
$$\frac{}{\varnothing \vdash}\,(\mathrm{EC}) \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash}\,(\mathrm{CE})$$

Substitutions :
$$\frac{}{\Delta \vdash \langle\rangle : \varnothing}\,(\mathrm{ES}) \qquad \frac{\Delta \vdash \sigma : \Gamma \qquad \Gamma \vdash A \qquad \Delta \vdash t : A[\sigma]}{\Delta \vdash \langle\sigma, t\rangle : (\Gamma, x : A)}\,(\mathrm{SE})$$

où l'on suppose vérifiées, dans la règle (WK) la condition de bord $x \notin \mathrm{Var}(t)$, et dans la règle (CE) la condition de bord $x \notin \mathrm{Var}(\Gamma)$. Remarquons que a notation $A[\sigma]$ n'est pas encore introduite, mais elle le sera en Section 2.3. Par concision, on s'autorisera dans la suite à omettre l'indice A dans les types flèches. Étant donné un contexte bien formé $\Gamma = (x_1 : A_1, \dots, x_n : A_n)$, on note $\mathrm{id}_\Gamma = \langle x_1, \dots, x_n \rangle$ la substitution identité dessus.

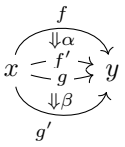
Modèles. Un *modèle* de cette théorie consiste en la donnée d'un ensemble $[\![A]\!]$ pour chaque type A (cet ensemble dépendant de l'interprétation des variables du contexte dans lequel est défini A) et d'un élément $[\![t]\!]$ de cet ensemble pour chaque terme t de ce type, satisfaisant des propriétés adéquate. Ici, un modèle va consister précisément en un ω-graphe C : l'interprétation de $\star$ nous fournit l'ensemble des objets $C_0 = [\![\star]\!]$ et, étant données deux n-cellules $f, g \in C_n$, l'interprétation du type $x \to y$, lorsque x et y sont respectivement interprétés par f et g, nous fournit l'ensemble $[\![x \to y]\!]_{\substack{x \mapsto f \\ y \mapsto g}}$ des $(n+1)$-cellules de source f et de but g.

Contextes. Il sera très utile pour la suite de remarquer qu'un contexte bien formé dans notre théorie peut être vu comme un diagramme : une variable $x : \star$ donne une 0-cellule et une variable $z : x \to y$ donne une $(n+1)$-cellule entre les deux n-cellules x et y. Ainsi, le diagramme (1) est décrit par le contexte suivant :

$$x : \star\ ,\ y : \star\ ,\ z : \star\ ,\ f : x \to y\ ,\ f' : x \to y\ ,\ f'' : x \to y\ ,\ g : y \to z\ ,\ \alpha : f \to f'\ ,\ \beta : f' \to f'' \tag{2}$$

2.2 Contextes de composition

La théorie CaTT est obtenue à partie de la théorie ci-dessus en ajoutant des règles dont l'effet va être le suivant : étant donné un contexte Γ correspondant à un diagramme qui est « composable », sa composition doit être définie, c'est-à-dire que l'on doit avoir un terme correspondant à cette composition. Il nous faut donc tout d'abord spécifier ce que l'on entend par composable : intuitivement, un diagramme est composable lorsque l'on peut définir de façon non-ambiguë sa composition. Par exemple, on s'attend à ce qu'un diagramme de la forme (1) soit composable, mais en revanche on ne s'attend pas à ce que des diagrammes de la forme

$f \circlearrowright x \circlearrowleft g$ ou $x \xrightarrow{f} y \xleftarrow{g} z$ ou 

soient composables. En effet, dans le premier cas on ne sait pas l'ordre dans lequel composer les flèches et dans le deuxième et le troisième, les flèches ne sont pas ordonnées “séquentiellement”.

Contextes de composition. Nous appelons *contexte de composition*, un contexte correspondant à un diagramme composable. Formellement, on peut montrer que ce sont les contextes Γ tels que le jugement $\Gamma \vdash_{\mathsf{ps}}$ est dérivable à l'aide de la règle (PS) ci-dessous, ainsi que des trois

premières règles qui permettent de dériver des jugements de la forme $\Gamma \vdash_{\mathsf{ps}} x : A$.

$$\frac{}{x : \star \vdash_{\mathsf{ps}} x : \star}(\text{PSI}) \qquad \frac{\Gamma \vdash_{\mathsf{ps}} x : A}{\Gamma, y : A, f : x \underset{A}{\longrightarrow} y \vdash_{\mathsf{ps}} f : x \underset{A}{\longrightarrow} y}(\text{PSE})$$

$$\frac{\Gamma \vdash_{\mathsf{ps}} f : x \underset{A}{\longrightarrow} y}{\Gamma \vdash_{\mathsf{ps}} y : A}(\text{PSD}) \qquad \frac{\Gamma \vdash_{\mathsf{ps}} x : \star}{\Gamma \vdash_{\mathsf{ps}}}(\text{PS})$$

Dans la règle (PSE), on supposera toujours $y, f \notin \mathrm{Var}(\Gamma)$. Intuitivement, ces règles décrivent le fait que les diagrammes de composition sont ceux obtenus à partir d'un point en attachant, à l'un des buts du diagramme, de nouvelles n-cellules dont le but est libre. Le fait que l'on attache sur les buts permet d'assurer la « séquentialité » du diagramme et la liberté des variables ajoutées son « acyclicité ».

Source et but. Pour les contextes de composition Γ non réduits à un point, on peut définir une notion de source (resp. de but) qui correspond au contexte obtenu en enlevant les cellules de dimension maximale ainsi que leur but (resp. source). Ainsi le contexte Γ décrit en (2) et figuré en (1) aura pour source et but les diagrammes $\partial^-(\Gamma)$ et $\partial^+(\Gamma)$ suivants, représentés sous formes de diagrammes :

$$\partial^-(\Gamma) = \; x \overset{f}{\frown} y \xrightarrow{g} z \qquad\qquad \partial^+(\Gamma) = \; x \underset{f''}{\smile} y \xrightarrow{g} z$$

Ces opérations peuvent se définir simplement directement sur les contextes de composition, c'est tout ce que nous aurons besoin de savoir dans cet article.

2.3 Règles pour les cohérences

Nous pouvons maintenant formuler les règles à ajouter au système de la Section 2.1 afin d'avoir une théorie dont les modèles sont les ω-catégories. Intuitivement, il suffit d'ajouter une règle de la forme

$$\frac{\Gamma \vdash_{\mathsf{ps}} \qquad \Gamma \vdash A}{\Gamma \vdash \mathsf{coh}_{\Gamma,A} : A}$$

qui stipule que lorsque Γ est un contexte de composition et A un type dans ce contexte alors une cellule type A peut s'obtenir en « composant » le contexte Γ, la cohérence qui témoigne de cette composition étant dénotée $\mathsf{coh}_{\Gamma,A}$. Cependant, il apparaît que ceci axiomatise les groupoïdes et qu'il faut de fait deux variantes de cette règle avec des conditions de bord différentes. De plus, il faut que ces variantes intègrent l'action des substitutions. Afin de les formuler précisément, il nous faut donc tout d'abord définir l'action d'une substitution formelle.

Dans la suite, nous supposons que les termes sont soit des variables, soit de la forme $\mathsf{coh}_{\Gamma,A}(\sigma)$ où Γ est un contexte, A un type et σ une substitution (un tel terme correspond à une cohérence sous une substitution formelle). On définit les variables libres d'un terme construit avec coh par $\mathrm{Var}(\mathsf{coh}_{\Gamma,A}(\sigma)) = \mathrm{Var}(\sigma)$ (on considère que les variables de Γ sont liées dans A).

Action des substitutions. Supposons donnée une substitution σ, avec $\Delta \vdash \sigma : \Gamma$ dérivable. Étant donné un type A, avec $\Gamma \vdash A$ dérivable (resp. un terme t avec $\Gamma \vdash t : A$ dérivable) on définit l'action de σ sur A (resp. sur t) notée $A[\sigma]$ (resp. $t[\sigma]$) par induction par

$$\star[\sigma] = \star \qquad (t \underset{A}{\to} u)[\sigma] = t[\sigma] \underset{A[\sigma]}{\longrightarrow} u[\sigma] \qquad x_i[\sigma] = t_i \qquad \mathsf{coh}_{\Gamma',B}(\delta)[\sigma] = \mathsf{coh}_{\Gamma',B}(\delta \circ \sigma)$$

où $\Gamma = (x_1 : A_1, \dots, x_n : A_n)$ et $\sigma = \langle t_1, \dots, t_n \rangle$. De plus, pour une substitution $\delta = \langle u_1, \dots, u_m \rangle$, sa composition avec σ est définie par $\delta \circ \sigma = \langle u_1[\sigma], \dots, u_m[\sigma] \rangle$. On peut alors montrer que $\Delta \vdash A[\sigma]$ (resp. $\Delta \vdash t[\sigma] : A[\sigma]$) est toujours dérivable.

Règles pour les cohérences. Le système CaTT contient les deux règles suivantes pour l'introduction de cohérences.

— *Cohérences d'opérations* :

$$\frac{\Gamma \vdash_{\mathsf{ps}} \qquad \Gamma \vdash A \qquad \partial^-(\Gamma) \vdash t : A \qquad \partial^+(\Gamma) \vdash u : A \qquad \Delta \vdash \sigma : \Gamma}{\Delta \vdash \mathsf{coh}_{\Gamma, t \underset{A}{\to} u}(\sigma) : t[\sigma] \underset{A[\sigma]}{\longrightarrow} u[\sigma]}(\text{OP})$$

où l'on suppose $\mathrm{Var}(\partial^-(\Gamma)) = \mathrm{Var}(t) \cup \mathrm{Var}(A)$ et $\mathrm{Var}(\partial^+(\Gamma)) = \mathrm{Var}(u) \cup \mathrm{Var}(A)$.

— *Cohérences d'égalité* :

$$\frac{\Gamma \vdash_{\mathsf{ps}} \qquad \Gamma \vdash A \qquad \Delta \vdash \sigma : \Gamma}{\Gamma \vdash \mathsf{coh}_{\Gamma,A}(\sigma) : A[\sigma]}(\text{EQ})$$

où l'on suppose $\mathrm{Var}(\Gamma) = \mathrm{Var}(A)$.

Intuitivement, la première règle (cohérence d'opérations) va permettre de définir des opérations, telles que les opérations de composition dans une ω-catégorie. Par exemple, dans le contexte Γ suivant, qui correspond au diagramme de droite

$$\Gamma = (x : \star, y : \star, f : x \to y, z : \star, g : y \to z) \qquad\qquad x \xrightarrow{f} y \xrightarrow{g} z \tag{3}$$

on va pouvoir appliquer cette règle avec $A = x \to z$, $\Delta = \Gamma$, $\sigma = \mathrm{id}_\Gamma = \langle x, y, f, z, g \rangle$ et obtenir une cohérence qui correspond à la composition des 1-cellules f et g, qu'on notera ci-dessous $\mathsf{comp}(f, g)$ au lieu de $\mathsf{coh}_{\Gamma,A}(\mathrm{id}_\Gamma)$. De façon plus générale, lorsque σ n'est pas l'identité, cette règle permet de définir de façon similaire la composition de cellules de Γ.

La seconde règle (cohérence d'égalité) permet de définir des cellules qui sont le témoin d'axiomes vérifiés dans les ω-catégories. Par exemple, en utilisant la règle avec Γ correspondant au diagramme

$$x \xrightarrow{f} y \xrightarrow{g} z \xrightarrow{h} w$$

on va pouvoir appliquer la règle avec

$$A \quad = \quad \mathsf{comp}(\mathsf{comp}(f, g), h) \to \mathsf{comp}(f, \mathsf{comp}(g, h))$$

ainsi que $\Delta = \Gamma$ et $\sigma = \mathrm{id}_\Gamma$. On obtient alors une 2-cellule du type A ci-dessus qui témoigne de l'associativité de la composition (on peut de plus montrer que cette cellule est réversible, c'est-à-dire qu'elle admet un inverse à cellule réversible près).

Insistons sur le fait que ces règles sont les seules présentes dans le système. Toutes les autres structures que l'on s'attend à trouver dans une ω-catégorie peuvent se déduire de ces dernières (nous avons vu ci-dessus que c'était par exemple le cas pour la composition et l'associativité, d'autres exemples sont donnés ci-dessous).

2.4 Exemples de dérivation

Pour comprendre ces règles de cohérences que l'on vient d'introduire, donnons quelques exemples de dérivations. On utilisera la syntaxe concrète de notre assistant de preuve. Par exemple, dans celui-ci la composition des 1-cellules formalisée à la section précédente est définie par

```
coh comp (x:*) (y:*) (f:x->y) (z:*) (g:y->z) : x->z
```

Ici, `coh` est un mot-clé indiquant qu'on va définir une cohérence, `comp` est le nom de la cohérence, la liste d'arguments typés (`x`, `y`, `f`, `z`, `g`) est une notation pour le contexte de composition Γ figuré en 3 et le type `x->z` à droite de « : » indique que l'on utilise la règle (OP) avec $t = \mathtt{x}$ et $u = \mathtt{z}$. De même l'associativité, sera obtenue par

```
coh assoc (x:*) (y:*) (f:x->y) (z:*) (g:y->z) (w:*) (h:z->w) :
    comp x z (comp x y f z g) w h -> comp x y f w (comp y z g w h)
```

dont la typabilité sera vérifié en utilisant la règle (EQ). On voit que de nombreux arguments dans le type de retour sont redondants et en pratique celui-ci peut être simplement noté

```
    comp (comp f g) h -> comp f (comp g h)
```

les autres arguments étant inférés automatiquement. Notons d'autre part que la notation `comp` $\mathtt{t}$ $\mathtt{u}$ permet de spécifier la substitution σ utilisée dans la cohérence `comp` : celle-ci remplace `f` par le terme $\mathtt{t}$ et `g` par $\mathtt{u}$ (encore une fois, la valeur des autres variables est inférée automatiquement).

Une entrée de l'assistant de preuve consiste en une suite de telles déclarations, et notre programme va automatiquement vérifier qu'elles peuvent bien être inférées dans le système de types décrit ci-dessus. Notons qu'un séquent ne peut jamais être dérivé à la fois par les règles (OP) et (EQ), il n'y a donc pas d'ambiguïté sur ce point.

Quelques cohérences d'opérations. La règle (OP) permet définir d'autres compositions ; donnons quelques exemples à titre d'illustration. La composition verticale de deux 2-cellules est obtenue par

```
coh vcomp (x:*) (y:*) (f:x->y) (g:x->y) (a:f->g) (h:x->y) (b:g->h) : f->h
```

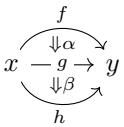
la composition horizontale par

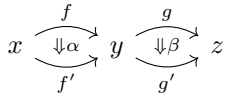
```
coh hcomp (x:*) (y:*) (f:x->y) (f':x->y) (a:f->f')
                (z:*) (g:y->z) (g':y->z) (b:g->g') :
    comp f g -> comp f' g'
```

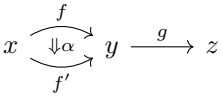
le moustachement à droite par

```
coh rwhisk (x:*) (y:*) (f:x->y) (f':x->y) (a:f->f') (z:*) (g:y->z) :
    comp f g -> comp f' g
```

Graphiquement, ces opérations correspondent respectivement aux compositions des diagrammes suivants :







Quelques cohérences d'égalité. À titre d'illustration, les identités peuvent être définies en utilisant la règle (EQ) :

```
coh id  (x:*) : x->x
coh id2 (x:*) (y:*) (f:x->y) : f->f
```

et on peut de même montrer qu'elle sont des éléments neutres à gauche pour la composition (de façon cohérente)

```
coh unitl   (x:*) (y:*) (f:x->y) : f -> comp (id x) f
coh unitl'  (x:*) (y:*) (f:x->y) : comp (id x) f -> f
coh unitl'' (x:*) (y:*) (f:x->y) : vcomp (unitl f) (unitl' f) -> id2 f
```

etc.

Règle de coupure. La règle de coupure

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash \sigma : \Gamma}{\Delta \vdash t[\sigma] : A[\sigma]}(\text{CUT})$$

est admissible. Dans la pratique, notre implémentation ajoute cette règle comme primitive, ce qui permet de faire des déclarations qui sont indiquées par le mot-clé `let` ; par exemple, la composition d'un endomorphisme avec lui-même peut être définie par

```
let square (x:*) (f:x->x) : x->x = comp f f
```

Au passage, ce morphisme ne peut pas directement être défini comme une cohérence car le contexte $\Gamma = (x : \star, f : x \to x)$ n'est pas un contexte de composition.

3 Suspension

Dans l'exemple en fin de section précédente, nous avons dû définir deux fois la cohérence identité : sur les 1-cellules (`id`) et sur les 2-cellules (`id2`). De même, la composition des 1-cellules (`comp`) et la composition verticale des 2-cellules (`vcomp`) sont des opérations très similaires. Plus généralement, dans la pratique, on est souvent amené à effectuer les « mêmes » développements plusieurs fois afin qu'ils s'appliquent à des cellules de dimensions différentes, ce qui freine considérablement le développement dès que la dimension des cellules augmente, voir [1] pour un exemple concret. On souhaiterait bien sûr automatiser ce processus de « généralisation à des cellules de dimension supérieure » : nous expliquons ici comment mener à bien cette tâche que nous appelons la *suspension* des morphismes.

Afin de comprendre le principe de la suspension, comparons les définitions de `comp` et de `vcomp`, où l'on a renommé les variables de la seconde pour mettre en avant la similitude :

```
coh comp              (x:*)    (y:*)    (f:x->y) (z:*)    (g:y->z) : x->z
coh vcomp (a:*) (b:*) (x:a->b) (y:a->b) (f:x->y) (z:a->b) (g:y->z) : x->z
```

On remarque que la définition de la seconde peut essentiellement être obtenue à partir de celle de la première en ajoutant deux variables `a` et `b` et en remplaçant les occurrences du type `*` par `a->b`. La suspension consiste à effectuer (automatiquement) cette transformation à un morphisme. En pratique, cela signifie que l'on pourra définir une fois la composition `comp` et l'appliquer à une paire de 1-cellules, de 2-cellules, ou de n-cellules, notre outil calculant la bonne suspension à la volée.

3.1 Définition

Définissons maintenant formellement cette transformation et montrons qu'elle est admissible dans notre système, c'est-à-dire qu'elle ne modifie pas fondamentalement notre théorie des types. Étant données deux variables fraîches a et b, on définit la suspension Σt, ΣA, $\Sigma\Gamma$ et $\Sigma\sigma$ d'un terme t, d'un type A, d'un contexte Γ et d'une substitution σ par induction par

$$\begin{aligned} \Sigma x &= x & \Sigma(\mathsf{coh}_{\Gamma,A}(\sigma)) &= \mathsf{coh}_{\Sigma\Gamma,\Sigma A}(\Sigma\sigma) \\ \Sigma\star &= a \underset{\star}{\to} b & \Sigma(x \underset{A}{\to} y) &= \Sigma x \underset{\Sigma A}{\longrightarrow} \Sigma y \\ \Sigma\varnothing &= (a:\star, b:\star) & \Sigma(\Gamma, x:A) &= (\Sigma\Gamma, x:\Sigma A) \\ \Sigma\langle\rangle &= \langle a, b\rangle & \Sigma(\langle\sigma, t\rangle) &= \langle\Sigma\sigma, \Sigma t\rangle \end{aligned}$$

La validité de cette transformation réside dans le fait qu'elle préserve la prouvabilité :

Théorème 1. *Pour tout jugement $\Gamma \vdash t : A$ dérivable, le jugement $\Sigma\Gamma \vdash \Sigma t : \Sigma A$ est aussi dérivable.*

Démonstration. La preuve se fait par induction mutuelle sur les règles d'inférences. On montre l'admissibilité des règles suivantes :

$$\frac{\Gamma \vdash}{\Sigma\Gamma \vdash} \qquad \frac{\Gamma \vdash A}{\Sigma\Gamma \vdash \Sigma A} \qquad \frac{\Delta \vdash \sigma : \Gamma}{\Sigma\Delta \vdash \Sigma\sigma : \Sigma\Gamma} \qquad \frac{\Gamma \vdash_{\mathsf{ps}}}{\Sigma\Gamma \vdash_{\mathsf{ps}}} \qquad \frac{\Gamma \vdash t : A}{\Sigma\Gamma \vdash \Sigma t : \Sigma A}$$

ainsi que les égalités $\partial^-(\Sigma\Gamma) = \Sigma(\partial^-(\Gamma))$ et $\partial^+(\Sigma\Gamma) = \Sigma(\partial^+(\Gamma))$ □

3.2 Application

Nous venons de définir une transformation sur les expressions de cette théorie des types. À tout jugement de typage bien formé $\Gamma \vdash t : A$, cette transformation associe un nouveau jugement de typage $\Sigma\Gamma \vdash \Sigma t : \Sigma A$, qui est lui aussi bien formé, et dont l'arbre de dérivation se déduit de celui du jugement original. Toutes ces définitions peuvent être implémentées. L'utilité de ce principe réside dans le fait qu'il permet de générer automatiquement de nouveaux termes de la théorie, sans avoir à les définir. En pratique, l'implémentation calcule combien de fois appliquer la transformation de suspension en regardant la différence entre la dimension de l'argument fourni et celle de l'argument attendu.

Dimension. La *dimension* d'un type est définie inductivement par

$$\dim(\star) = 0 \qquad \dim(t \underset{A}{\to} u) = 1 + \dim A$$

On parlera plus généralement de la dimension d'un terme dans un contexte pour parler de la dimension de son type dans ce contexte : dans les modèles, un terme de dimension n sera interprété par une n-cellule. Par induction, on montre alors que l'opération de suspension incrémente de 1 la dimension d'un terme. Remarquons que, pour être valide, une substitution doit toujours associer à chaque variable un terme de la même dimension que cette variable.

Implémentation. On peut maintenant donner une version modifiée de la règle (CUT), qui prend en compte les suspensions

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash \sigma : \Sigma^n \Gamma}{\Delta \vdash (\Sigma^n t)\,[\sigma] : (\Sigma^n A)\,[\sigma]}\,(\mathrm{CUT}_n)$$

Lors de la recherche de preuve, une seule valeur de n est susceptible de convenir (c'est-à-dire de mener à une dérivation correcte). En effet, étant donnée la substitution σ, considérons le terme u qu'elle associe à une variable x quelconque de son domaine : l'entier n est nécessairement égal à la différence entre la dimension de x dans Γ et la dimension u. Notre système utilise ce principe lors de la vérification de type. Par ailleurs, afin de rester le plus possible fidèle à la théorie originale, nous avons séparé un noyau, qui calcule sans utiliser la suspension ni la fonctorialité, et une interface pour les gérer de façon externe.

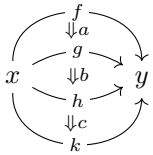
Utilisation. Si l'on reprend quelques définitions données en Section 2.4, en prenant en compte la suspension, on peut redéfinir la composition, l'identité, ainsi qu'un témoin que l'identité est un élément neutre pour la composition :

```
coh comp    (x:*) (y:*) (f:x->y) (z:*) (g:y->z) : x -> z
coh id      (x:*) : x -> x
coh unitl   (x:*) (y:*) (f:x->y) : f -> comp (id x) f
coh unitl'  (x:*) (y:*) (f:x->y) : comp (id x) f -> f
coh unitl'' (x:*) (y:*) (f:x->y) : comp (unitl f) (unitl' f) -> id f
```

Comparé à l'exemple précédent, pour la définition de `unitl''`, il n'est plus nécessaire de définir `id2` : il suffit de suspendre la cohérence `id`. De plus, comme `id` attend une 1-cellule en argument, et qu'on lui fournit une 2-cellule, le système vérifie la dérivation en suspendant une fois. Il en va de même pour `vcomp`, qui est une suspension de `comp`. Par ailleurs, la suspension permet de transférer immédiatement toutes les cohérences d'égalité faisant intervenir cette `comp` à des égalité sur `vcomp`. Par exemple, on avait défini un témoin d'associativité

```
coh assoc (x:*) (y:*) (f:x->y) (z:*) (g:y->z) (w:*) (h:z->w) :
    comp (comp f g) h -> comp f (comp g h)
```

Si maintenant on souhaite obtenir un témoin d'associativité pour la composition verticale des 2-cellules, par exemple dans le contexte correspondant au diagramme



il suffira d'écrire l'expression suivante `assoc a b c` et la suspension se chargera de générer un témoin d'associativité pour la composition verticale.

4 Fonctorialisation

L'idée de cette section est similaire à celle de la précédente, il s'agit de refléter par un calcul méta-théorique un certain principe que l'on souhaite pouvoir utiliser afin de raccourcir

les preuves. Elle aboutit cependant à des généralisations différentes des opérations de celles de la partie précédente. La transformation que nous étudions ici est fondée sur l'observation que les cohérences dans les ω-catégories se comportent de façon analogue aux foncteurs usuels entre catégories. En effet, un foncteur $F : C \to D$ entre deux catégories C et D associe à tout objet c de C un objet Fc de D et à tout morphisme $f : c \to d$ de C un morphisme $Ff : Fc \to Fd$ de D. Ceci peut être figuré par le diagramme :

$$\begin{array}{ccc} c & \longmapsto & Fc \\ {\scriptstyle f}\downarrow & \longmapsto & \downarrow{\scriptstyle Ff} \\ d & \longmapsto & Fd \end{array}$$

Par analogie, on peut représenter une cohérence, par exemple la composition, comme une association $(f, g) \mapsto \texttt{comp}\ f\ g$ et l'opération de fonctorialisation, par rapport à la première variable, permet, pour toute cellule $\alpha : f \to f'$, de définir une composition

$$\texttt{comp}\ \alpha\ g \quad : \quad \texttt{comp}\ f\ g \quad \to \quad \texttt{comp}\ f'\ g$$

On peut bien sûr définir une généralisation analogue vis-à-vis de la seconde variable, ou des deux à la fois :

$$\begin{array}{ccc} f, g & \longmapsto & \texttt{comp}\ f\ g \\ {\scriptstyle \alpha,\beta}\downarrow & \longmapsto & \downarrow{\scriptstyle \texttt{comp}\ \alpha\ \beta} \\ f', g' & \longmapsto & \texttt{comp}\ f'\ g' \end{array}$$

En pratique, nous allons noter entre crochets `[]` les variables par rapport auxquelles ont fonctorialise. Notre but est donc de définir la signification de l'instruction suivante :

```
let comp2 (x:*) (y:*) (f:x->y) (f':x->y) (a:f->f')
                (z:*) (g:y->z) (g':y->z) (b:g->g')
    : comp f g -> comp f' g'
    = comp [a] [b]
```

Au passage, on voit la différence avec l'opération de suspension de la section précédente qui permet de définir la composition verticale à partir de `comp` : on cherche ici à définir la composition horizontale. Notre but, dans cette section, est de généraliser cette observation à toutes les cohérences d'opérations et de formaliser la transformation associée.

4.1 Fonctorialisation d'un contexte

Définition. Dans un premier temps, nous allons définir la transformation que l'on cherche à effectuer sur les contextes. Celle-ci va, partant d'une variable f d'un contexte Γ, ajouter une nouvelle variable f' de même source et même but que f, ainsi qu'une cellule $\alpha : f \to f'$ entre les deux. On notera $\mathrm{Fun}_f(\Gamma)$ le contexte résultant, appelé *fonctorialisation* de Γ par rapport à f. On aura par exemple, en représentant les contextes sous forme de diagrammes,

$$\Gamma = \ x \xrightarrow{f} y \xrightarrow{g} z \qquad\qquad \mathrm{Fun}_f(\Gamma) = \ x \underset{f'}{\overset{f}{\Downarrow \alpha}} y \xrightarrow{g} z \tag{4}$$

Plus formellement, étant donné un contexte $\Gamma = (x_1 : A_1, \cdots, x_n : A_n)$ et une variable $x_i \in \mathrm{Var}(\Gamma)$, on définit

$$\mathrm{Fun}_{x_i}(\Gamma) = (x_1 : A_1, \cdots, x_{i-1} : A_{i-1}, x_i : A_i, x_i' : A_i, f : x_i \underset{A_i}{\longrightarrow} x_i', x_{i+1} : A_{i+1}, \ldots, x_n : A_n) \tag{5}$$

où les variables x_i' et f sont fraîches dans Γ. Cette opération préserve la bonne formation des contextes :

Proposition 2. *La règle suivante est admissible, pour $x \in \mathrm{Var}(\Gamma)$:*

$$\frac{\Gamma \vdash}{\mathrm{Fun}_x(\Gamma) \vdash}$$

Démonstration. Par induction sur la dérivation de la prémisse □

Inclusions dans le fonctorialisé. Étant donné un contexte Γ admettant x comme variable libre, on définit le contexte $\Gamma[x'/x]$, qui est obtenu à partir de Γ en remplaçant toutes les occurrences de x par x'. Notons que les contextes Γ et $\Gamma[x'/x]$ sont inclus de façon canonique dans le fonctorialisé $\mathrm{Fun}_x(\Gamma)$ (voir (4) par exemple), ce qui peut être formalisé par l'existence de deux substitutions canoniques ι_x^Γ et $\mathfrak{s}_x^\Gamma$ qui sont telles que les séquents

$$\mathrm{Fun}_x(\Gamma) \vdash \iota_x^\Gamma : \Gamma \qquad\qquad \mathrm{Fun}_x(\Gamma) \vdash \mathfrak{s}_x^\Gamma : \Gamma[x'/x]$$

sont dérivable. En utilisant les notations de (5), on a

$$\iota_{x_i}^\Gamma = \langle x_1, \ldots, x_n \rangle \qquad\qquad \mathfrak{s}_{x_i}^\Gamma = \langle x_1, \ldots, x_{i-1}, x_i', x_{i+1}, \ldots, x_n \rangle.$$

Par la suite, on s'autorisera à laisser implicites les indices x et Γ de ι et $\mathfrak{s}$ quand il n'y a pas d'ambiguïté.

Notons que l'action de ι et $\mathfrak{s}$ sur un terme ou un type ne modifie pas ce dernier, mais simplement le contexte dans lequel il est défini. On s'autorisera donc à simplement noter t (resp. A) au lieu de $t[\iota]$ (resp. $A[\iota]$) pour un terme t (resp. un type A) sous l'action de ι, et pareillement pour $\mathfrak{s}$.

Proposition 3. *Les règles suivantes sont admissibles :*

$$\frac{\Gamma \vdash A}{\mathrm{Fun}_x(\Gamma) \vdash A} \qquad \frac{\Gamma \vdash A}{\mathrm{Fun}_x(\Gamma) \vdash A[x'/x]} \qquad \frac{\Gamma \vdash t : A}{\mathrm{Fun}_x(\Gamma) \vdash t : A} \qquad \frac{\Gamma \vdash t : A}{\mathrm{Fun}_x(\Gamma) \vdash t[x'/x] : A[x'/x]}$$

où x' est la variable fraîche de même source et but que x ajoutée au contexte.

Démonstration. Ce sont essentiellement des simplifications des jugements dérivables

$$\mathrm{Fun}(\Gamma) \vdash A[\iota] \quad \mathrm{Fun}(\Gamma) \vdash t[\iota] : A[\iota] \quad \mathrm{Fun}(\Gamma) \vdash A[x'/x][\mathfrak{s}] \quad \mathrm{Fun}(\Gamma) \vdash t[x'/x][\mathfrak{s}] : A[x'/x][\mathfrak{s}]$$

□

Fonctorialisation d'un contexte de composition. Dans la suite, on s'intéressera uniquement à la fonctorialisation d'un contexte par rapport à une variable de dimension maximale dans ce contexte, pour des raisons qui seront précisées à la fin de la Section 4.2. Afin de montrer que l'opération de fonctorialisation est compatible avec la règle (OP), nous allons avoir besoin de montrer que la fonctorialisation est compatible avec les contextes de composition :

Proposition 4. *Si* Γ *est un contexte de composition et* x *est une variable de dimension maximale de* Γ*, alors* $\mathrm{Fun}_x(\Gamma)$ *est aussi un contexte de composition. Autrement dit, la règle*

$$\frac{\Gamma \vdash_{\mathsf{ps}}}{\mathrm{Fun}_x(\Gamma) \vdash_{\mathsf{ps}}}$$

est admissible.

Source et but d'un contexte de composition fonctorialisé. Pour les mêmes raison, nous allons aussi avoir besoin de montrer que la fonctorialisation est compatible avec les source et buts :

Proposition 5. *Si* Γ *est un contexte de composition et* x *une variable de dimension maximale dans* Γ*, alors on peut calculer la source et le but de* $\mathrm{Fun}_x(\Gamma)$ *:*

$$\partial^-(\mathrm{Fun}_x(\Gamma)) = \Gamma \qquad\qquad \partial^+(\mathrm{Fun}_x(\Gamma)) = \Gamma[x'/x]$$

4.2 Cohérences fonctorialisées

Nous allons maintenant définir une opération de fonctorialisation pour les cohérences, qui va permettre de générer de nouvelles cohérences à partir de cohérences d'opérations déjà connues. Nous ne définissons cette transformation que pour les cohérences d'opérations, dont les arguments sont en dimension maximale. Nous discuterons de ces restrictions à la fin de cette section.

Termes et types fonctorialisables. Définissons la fonctorialisation des termes comme la fonction (partiellement définie) telle que

$$\mathrm{Fun}_x(y) = \begin{cases} f & \text{si } x = y \\ y & \text{sinon} \end{cases} \qquad \mathrm{Fun}_x(\mathsf{coh}_{\Gamma,B}) = \mathsf{coh}_{\mathrm{Fun}_x(\Gamma), \mathsf{coh}_{\Gamma,B}(\iota) \underset{B}{\longrightarrow} \mathsf{coh}_{\Gamma[x'/x],B}(\lambda)}$$

la dernière égalité n'étant définie que dans le cas où $\mathsf{coh}_{\Gamma,B}$ est une cohérence d'opération, et x une variable de dimension maximale dans Γ : on dit alors que le terme est *fonctorialisable* par rapport à x.

Fonctorialisation des cohérences d'opérations. Supposons fixée une certaine cohérence dérivée par la règle (OP), où l'on suppose tout d'abord pour simplifier que la substitution δ est l'identité et l'on note $B = t \to u$:

$$\frac{\Gamma \vdash_{\mathsf{ps}} \qquad \Gamma \vdash A \qquad \partial^-(\Gamma) \vdash t : A \qquad \partial^+(\Gamma) \vdash u : A}{\Delta \vdash \mathsf{coh}_{\Gamma,B} : B}(\mathrm{OP})$$

Pour x une variable de dimension maximale de Γ, on peut alors facilement vérifier toutes les prémisses de la règle d'introduction des cohérences d'opération suivante, qui correspond à une version « fonctorialisée » de celle ci-dessus :

$$\frac{\mathrm{Fun}_x(\Gamma) \vdash_{\mathsf{ps}} \qquad \mathrm{Fun}_x(\Gamma) \vdash B \qquad \Gamma \vdash \mathsf{coh}_{\Gamma,B} : B \qquad \Gamma[x'/x] \vdash \mathsf{coh}_{\Gamma[x'/x],B} : B}{\Delta \vdash \mathrm{Fun}_x(\mathsf{coh})_{\Gamma,B} : \mathsf{coh}_{\Gamma,B}(\iota) \underset{B}{\longrightarrow} \mathsf{coh}_{\Gamma[x'/x],B}(\lambda)}(\mathrm{OP})$$

En effet, la variable x étant de dimension maximale dans le contexte Γ, elle n'apparaît ni dans $\partial^-(\Gamma)$, ni dans $\partial^+(\Gamma)$. Comme par ailleurs les jugements $\partial^-(\Gamma) \vdash t : A$ et $\partial^+(\Gamma) \vdash u : A$ sont

dérivables, il en découle que ni A, ni t, ni u, ne contiennent la variable x, et donc B non plus. Ceci justifie la dérivabilité du jugement $\Gamma[x'/x] \vdash \mathsf{coh}_{\Gamma[x'/x],B} : B$.

Traitons maintenant le cas général de la dérivation d'une cohérence avec une substitution non triviale $\Delta \vdash \delta : \Gamma$ dans les prémisses de la règle (OP). Choisissons x une variable de dimension maximale de Γ, et notons v le terme associé à x dans σ. La substitution σ étant dérivable, le jugement $\Delta \vdash v : C$ est dérivable pour un certain type C. Supposons que l'on ait également un jugement $\Delta \vdash v' : C$ ainsi qu'un terme $\Delta \vdash w : v \to v'$. À partir de σ, on peut définir deux substitutions

$$\Delta \vdash \sigma' : \Gamma[x'/x] \qquad\qquad \Delta \vdash \tau : \mathrm{Fun}_x(\Gamma)$$

où σ' est définie comme σ excepté qu'elle n'est pas définie sur x et associe v' à x', et τ est définie comme σ sur les variables de Γ, et à x' (resp. f) associe v' (resp. w). Avec ces constructions, on peut alors finalement montrer que la règle (OP) est compatible avec la fonctorialisation.

Théorème 6. *Pour toute cohérence obtenue par la règle* (OP)*, avec les notations précédentes, le jugement*

$$\Delta \vdash \mathrm{Fun}_x(\mathit{coh}_{\Gamma,B})\,(\tau) : \mathit{coh}_{\Gamma,B}(\sigma) \xrightarrow[B[\sigma]]{} \mathit{coh}_{\Gamma[x'/x],B}(\sigma')$$

est dérivable.

Il sera utile de généraliser ce théorème, pour pouvoir l'appliquer dans tous les cas de coupure :

Corollaire 7. *Pour tout terme t fonctorialisable par rapport à une variable x et tout jugement $\Gamma \vdash t : A$ dérivable, il existe un type B tel que le jugement $\Gamma \vdash \mathrm{Fun}_x(t) : B$ soit dérivable*

Vers une fonctorialisation générale. Dans le cas d'une cohérence obtenue avec la règle (EQ), on serait tenté de reproduire la même construction. Expliquons pourquoi on peut espérer une telle construction mais que celle-ci sera sensiblement plus compliquée à définir que la précédente. Considérons par exemple la cohérence d'identité pour les objets, que l'on note ici id, dans le contexte $\Gamma = (x : \star)$:

```
coh id (x:*) : x->x
```

On peut alors calculer $\mathrm{Fun}_x(\Gamma) = (x : \star, x' : \star, f : x \to x')$, et la cohérence identité pour la nouvelle variable x' et l'on cherche alors à définir un terme de type $\mathsf{id}\,x \to \mathsf{id}\,x'$ que l'on pourrait représenter graphiquement par

$$\begin{array}{ccc} x & \xrightarrow{\mathsf{id}\,x} & x \\ & \Downarrow & \\ x' & \xrightarrow[\mathsf{id}\,x']{} & x' \end{array}$$

Une telle 2-cellule ne peut bien sûr exister car nos deux 1-cellules n'ont pas même source et même but. On peut espérer résoudre ce problème en remarquant que l'on a une 1-cellule canonique entre x et x' dans $\mathrm{Fun}_x(\Gamma)$: la cellule f qui a été ajoutée lors de la fonctorialisation. En se servant de cette 1-cellule, on peut compléter le diagramme ci-dessus en

$$\begin{array}{ccc} x & \xrightarrow{\mathsf{id}\,x} & x \\ {\scriptstyle f}\downarrow & \Downarrow & \downarrow{\scriptstyle f} \\ x' & \xrightarrow[\mathsf{id}\,x']{} & x' \end{array}$$

La description d'un tel diagramme peut se faire dans notre théorie à l'aide de la cohérence de composition, et l'on pourrait ainsi espérer automatiquement générer la cohérence

```
coh id' (x:*) (y:*) (f:x->y) : comp (id x) f -> comp f (id y)
```

Cependant, cela nécessite que nous utilisions la cohérence `comp` qui, rappelons-le, n'est pas définie de façon primitive. De plus, lorsque les cohérences d'égalités sur lesquelles on travaille sont plus complexes, il n'est pas aisé de trouver les morphismes verticaux permettant de compléter le diagramme comme ci-dessus. Nous laissons la définition d'un tel algorithme pour des travaux futurs. Les calculs effectués à la main jusqu'à présent semblent suggérer qu'il serait plus naturel, pour exprimer cet algorithme, de développer une version « cubique » de la théorie des types, c'est-à-dire fondée sur la notion de cube au lieu de celle de globe. La situation est similaire pour la fonctorialisation par rapport à une variable qui n'est pas de dimension maximale.

4.3 Application

Implémentation. On formalise l'utilisation de la transformation de fonctorialisation par une nouvelle modification de la règle (CUT) :

$$\frac{\Gamma \vdash t : A \qquad \Delta \vdash \sigma : \mathrm{Fun}_x(\Gamma)}{\Delta \vdash \mathrm{Fun}_x(t)\,[\sigma] : B}(\mathrm{CUT}_x)$$

Il sera utile de remarquer que cette règle n'a de sens que lorsque le terme t est fonctorialisable par rapport à x. Lors de l'application de cette règle, l'utilisateur doit spécifier la variable x à l'aide du positionnement des crochets. Par exemple, en définissant

```
coh comp (x:*) (y:*) (f:x->y) (z:*) (g:y->z) : x->z
```

et en écrivant ensuite `comp [a] h`, le système va automatiquement interpréter qu'il devra fonctorialiser par rapport au premier argument, ici `f`. Par ailleurs, notre système permettant de faire non-seulement de la vérification de typage, mais aussi de l'inférence de types, le Théorème 7 suffit à vérifier la validité de cette construction.

En pratique, l'implémentation permet de fonctorialiser par rapport à plusieurs variables simultanément (les définitions suivent exactement le même principe que pour un seul argument), et de combiner dans une même étape les coupures avec fonctorialisation et suspension.

Utilisation. Reprenons quelques exemples de Section 2.4, en utilisant la transformation de fonctorialisation. On peut ainsi redéfinir la composition horizontale des 2-cellules, ainsi que le moustachement à droite à partir de la composition de 1-cellules par

```
coh comp  (x:*) (y:*) (f:x->y) (z:*) (g:y->z) : x -> z
let hcomp (x:*) (y:*) (f:x->y) (f':x->y) (a:f->f')
                (z:*) (g:y->z) (g':y->z) (b:g->g')
    = comp [a] [b]
let rwhisk (x:*) (y:*) (f:x->y) (f':x->y) (a:f->f') (z:*) (g:y->z)
    = comp [a] g
```

En pratique, on ne définira pas `hcomp` et `rwhisk`, mais on utilisera plutôt directement `comp` avec des arguments entre crochets aux endroits qui le nécessitent. Ceci permet de diminuer substantiellement la taille du fichier de preuve et d'en améliorer la lisibilité.

5 Conclusion

Efficacité en pratique. Pour quantifier l'utilité de nos transformations, nous avons développé en exemple une définition minimale du tressage et de son inverse dans les catégories doublement monoïdales, avec et sans utiliser ces transformations [1]. Sans transformation, le fichier total consiste en 531 lignes de code, pour 93 définitions. Avec la suspension, il se ramène à 476 lignes, (gain de 10%), et 85 définitions. Par ailleurs 58 des précédentes définitions (62%) ont pu être simplifiées. Avec en plus la fonctorialisation, la longueur du fichier est réduite à 422 lignes (gain de 11%), et le nombre de définitions à 76. Notons de plus que l'ajout de ces transformations permet de généraliser beaucoup de définitions ad-hoc, c'est pourquoi on s'attend à un gain relatif bien plus importants lors de la formalisation d'exemples en dimensions variées.

Vers d'autres transformations. Après avoir défini deux transformations qui se sont avérées utiles, il est naturel de se demander s'il y en aurait d'autres. Une bonne approche cette question est de comprendre finement la structure des contextes de compositions. En particuliers, ceux-ci sont semblables aux mots de Dyck, et la suspension et la fonctorialité s'interprètent dans cette structure. Adopter directement ce point de vue peut guider la recherche.

Syllepses. Les développements présentés ici ont été menés en parallèle d'une utilisation de CaTT dans le but de montrer l'existence d'une syllepse pour les ω-catégories triplement monoïdales [1], qui est l'un des problème ouverts sur lesquels l'apport de cet outil est attendu. Cela a permis par la pratique de mieux comprendre quelles transformations de la théorie auraient un impact important sur l'utilisation. Réciproquement, l'implémentation de ces nouvelles transformations a permis de simplifier et de prolonger le développement de cette démonstration.

Références

[1] Thibaut Benjamin. Towards a fully formalized definition of syllepsis in weak higher categories. In *Proceedings of HDRA workshop*, 2018.

[2] Thibaut Benjamin, Eric Finster, and Samuel Mimram. The CaTT proof assistant, 2018. https://github.com/ThiBen/catt.

[3] Eric Finster and Samuel Mimram. A Type-Theoretical Definition of Weak ω-Categories. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017.

[4] Alexander Grothendieck. Pursuing stacks. Unpublished manuscript, 1983.

[5] Georges Maltsiniotis. Grothendieck ∞-groupoids, and still another definition of ∞-categories. Preprint arXiv:1009.2331, 2010.

1. https://github.com/ThiBen/catt/tree/master/examples/eckmann-hilton-versions

SQL à l'Épreuve de Coq
Une Sémantique Formelle pour SQL *

Véronique Benzaken[1] and Évelyne Contejean[2]

[1] Université de Paris Sud - Université Paris Saclay
veronique.benzaken@u-psud.fr
[2] CNRS - Université de Paris Sud - Université Paris Saclay
evelyne.contejean@lri.fr

Résumé

Nous proposons une sémantique formelle, exécutable, mécanisée en Coq pour un fragment réaliste du langage SQL, le standard en terme de bases de données relationnelles. Ce fragment prend en compte des requêtes de la forme `select [distinct] from where group by having` en présence de *valeurs nulles, fonctions, agrégats, quantificateurs* ainsi que de requêtes *imbriquées* potentiellement *corrélées*. Nous relions ce fragment à l'algèbre relationnelle, équivalent pour SQL du λ-calcul pour les langages fonctionnels. Nous obtenons ainsi le premier résultat d'équivalence de ce fragment et de l'algèbre, formellement prouvé. Enfin, grâce au mécanisme d'extraction, nous fournissons un analyseur sémantique du langage SQL certifié en Coq, élément central pour l'obtention d'un compilateur SQL vérifié.

1 Introduction

Définir la sémantique formelle d'un langage est une tâche délicate mais cruciale pour permettre de raisonner rigoureusement sur le comportement des programmes et pour vérifier statiquement la correction des optimisations [13]. La formalisation des langages les plus répandus peut s'avérer ardue puisqu'ils sont fréquemment spécifiés par des documents rédigés en langue naturelle, et de ce fait ambigus. Même lorsqu'il existe une spécification formelle, elle est incomplète, ne couvre qu'un fragment limité du langage et sa correction n'a le plus souvent été vérifiée que par un humain faillible. Dans tous les cas, peu de garanties indiscutables attestent que la sémantique proposée rend fidèlement compte de la sémantique réelle et que les optimisations effectuées sont vraiment correctes. SQL est le langage standard pour les bases de données relationnelles (SGBDR), à ce titre il est largement utilisé et n'échappe pas à ce travers.

Une approche prometteuse pour obtenir des garanties irréfutables est d'utiliser des assistants à la preuve, comme Coq [19] ou Isabelle [20], pour définir des sémantiques exécutables, mécanisées dont la correction est vérifiable par une machine. Le projet CompCert [15] en est une démonstration éclatante. Notre visée à long terme est de développer un compilateur de SQL, certifié en Coq ; dans cet article nous présentons les aspects liés à la sémantique de SQL.

Spécificités de SQL S'il y a eu quelques tentatives pour définir une sémantique formelle pour SQL, elle ne traitent que de fragments limités du langage, ce qui s'explique, en partie, par l'existence de singularités inhérentes au langage. Il est notoire que le bloc `select from where` possède une sémantique *multi-ensembliste*, tandis les opérateurs du langage $\cup$ (`union`), $\cap$ (`intersect`) et $\setminus$ (`except`) ont une sémantique strictement ensembliste. SQL manipule des *valeurs* nulles (`NULL`) pour représenter de l'information inconnue. Dans le domaine des bases

*Financé par le projet ANR DataCert : ANR-15-CE39-0009.

de données, cet aspect est considéré de la plus haute importance. Les requêtes comportent des *fonctions* (`+,-,...`), et des *agrégats* (`sum, max,...`) très utilisés, en particulier par la clause `group by having`. Enfin, SQL permet l'imbrication de sous-requêtes, *i.e.*, des requêtes mettant en jeu différents blocs `select`, et, de façon plus pernicieuse, l'emploi de *requêtes corrélées*, *i.e.*, des sous-requêtes possédant des variables libres comme dans `select a1 from t1 group by a1 having exists (select a2 from t2 group by a2 having sum(1+0*a1) = 0);` où `a1` est libre dans le `select` le plus interne.

Toute sémantique fidèle se doit de rendre compte, avec précision, de ces quatre points qui pris *séparément* sont purement techniques. Les considérer *ensemble*, comme nous le faisons, constitue un défi dû à la gestion étrange des environnements dans le langage. Ceci explique, sans doute, l'absence à ce jour, d'une sémantique formelle pour SQL.

État de l'art La définition d'une sémantique formelle pour SQL a fait et fait encore l'objet de recherches actives dans la communauté des bases de données. L'algèbre relationnelle [8] étant à SQL ce que le λ-calcul est aux langages fonctionnels, c'est tout naturellement que les premiers travaux [5, 17] ont proposé des traductions de SQL vers l'algèbre, mais d'une version restreinte de SQL, sans valeurs nulles ni fonctions, ni requêtes imbriquées ni multi-ensembles. La première sémantique prenant les valeurs nulles et les multi-ensembles en compte est donnée par [12], mais encore une fois sans la clause `group by having` ni les fonctions ni les agrégats. Or, comme nous le montrerons en Sections 2 et 3, le traitement de ces constructions est particulièrement épineux.

Sur le versant des assistants à la preuve, la première tentative de formalisation, en Agda [18], de l'algèbre relationnelle est proposée par [10, 9] tandis que la première formalisation, complète, du modèle relationnel est proposée par [3] où le modèle de données, l'algèbre, les requêtes "tableaux", la procédure de semi-décision "chase" et les contraintes d'intégrité sont formalisés.

La toute première tentative de vérifier, en Coq, un SGBDR est présentée dans [16]. Cependant, le fragment considéré est une reconstruction de SQL dans laquelle les attributs des relations sont représentés par leur position. Ni la clause `group by having`, ni les quantificateurs dans les formules, les valeurs nulles, les agrégats ou les requêtes imbriquées ne sont traités. Un outil permettant de décider de l'équivalence de requêtes SQL est présenté dans [6]. À cette fin, HottSQL, une sémantique basée sur la notion de K-relation [11] est définie. Elle prend en compte le bloc `select from where` avec agrégats. Comme [16], une reconstruction du langage est utilisée, ce qui évite de se pencher sur les aspects les plus délicats relatifs à la gestion des environnements. Enfin, et surtout, cette sémantique n'est pas *exécutable* : il est impossible de vérifier qu'elle est conforme à celle de SQL. La proposition la plus proche de la nôtre en terme de sémantique mécanisée est présentée dans [2]. Les auteurs modélisent en Coq l'algèbre relationnelle imbriquée (NRA [7]) qui sert *directement* de sémantique à SQL. Ils assignent une sémantique par traduction. Cependant, la syntaxe de NRA s'écartant considérablement de celle de SQL, il n'est pas immédiat de se convaincre que la sémantique assignée reflète fidèlement celle du langage. Nous dissocions l'algèbre et sa sémantique de celle de SQL. C'est en cela que notre approche diffère de celle proposée par [2].

Contributions Cet article présente cinq contributions. Contrairement aux approches présentées dans les articles [16, 6], nous définissons (i) SQL_{Coq} (syntaxe et sémantique), une formalisation en Coq de SQL prenant en compte le bloc complet `select [distinct] from where group by having` en présence de *valeurs* `NULL` *fonctions, agrégats, quantificateurs* et de (sous) requêtes *imbriquées* potentiellement *corrélées*. Ce faisant, grâce au mécanisme d'extraction nous obtenons ainsi (ii) un analyseur sémantique pour SQL certifié en Coq. Nous formalisons en Coq (iii) SQL_{Alg} une algèbre relationnelle multi-ensembliste similaire, quoique l'étendant, à celle présentée dans [8]. Nous définissons, en Coq, les traductions de SQL_{Coq} vers SQL_{Alg} et vice versa et démontrons un théorème de préservation des sémantiques. Ceci permet de (iv) recouvrer

toutes les équivalences algébriques sur lesquelles se fondent les optimisations du compilateur et (v) d'obtenir le premier, à notre connaissance, résultat d'équivalence, formellement prouvé, entre ce fragment de SQL et l'algèbre relationnelle. Le développement est disponible à l'url : http://datacert.lri.fr/sqlcert/.

Organisation La Section 2, présente SQL et les spécificités devant être prises en compte afin de fournir une sémantique fidèle pour un fragment réaliste du langage. Nous définissons en Section 3, la syntaxe et la sémantique mécanisées de SQL_{Coq}. La Section 4 se concentre sur la mécanisation de SQL_{Alg}, la définition des traductions et la preuve du théorème d'adéquation. Nous concluons, tirons des enseignements et donnons nos perspectives en Section 5.

2 SQL : simple ... quoique

SQL est un langage déclaratif dédié à la manipulation de données stockées au sein de bases de données relationnelles. À cet égard il est souvent considéré comme simple et intuitif d'utilisation.

2.1 SQL en bref

SQL opère sur des collections de *n-uplets*. Les n-uplets sont des enregistrements étiquetés dont les champs prennent leurs valeurs dans des domaines atomiques (entiers, chaînes etc.). Dans ce cadre les étiquettes, appelées *attributs*, sont dénotables et sont utilisés dans des expressions construites à partir de fonctions et d'accumulateurs appelés *agrégats*. La structure d'une requête ou bloc SQL est la suivante : `select e from q where c1 group by e' having c2`. Par exemple `select a+2 as a', max(c) as mc from t where b>3 group by a having sum(c)=0` est une requête représentative du langage où `a,b,c` sont des attributs, `sum, max` des agrégats, `+` une fonction et `b>3` une condition.

Un bloc s'évalue ainsi : la requête `q` est évaluée, puis filtrée par la condition `c1`. Ce résultat intermédiaire est "groupé" c'est à dire partitionné grâce aux expressions `e'` qui s'évaluent de manière *homogène* sur les éléments de chaque groupe ; les groupes sont maximaux au sens de l'inclusion. Les groupes sont ensuite filtrés par la condition `c2`, et les expressions `e` sont évaluées pour chaque groupe produisant ainsi le résultat. Pour l'exemple précédent, en supposant que l'évaluation de `t` soit

$$\{(\texttt{a=1;b=1;c=4}), (\texttt{a=1;b=5;c=2}), (\texttt{a=1;b=4;c=-2}), (\texttt{a=3;b=5;c=1}), (\texttt{a=3;b=4;c=2})\},$$

l'étape de filtrage avec `b>3` donne

$$\{(\texttt{a=1;b=5;c=2}), (\texttt{a=1;b=4;c=-2}), (\texttt{a=3;b=5;c=1}), (\texttt{a=3;b=4;c=2})\}.$$

Le groupement selon `a` produit deux groupes

$$\{(\texttt{a=1;b=5;c=2}), (\texttt{a=1;b=4;c=-2})\} \text{ et } \{(\texttt{a=3;b=5;c=1}), (\texttt{a=3;b=4;c=2})\}\}.$$

La seconde étape de filtrage par `sum(c)=0` élimine le dernier groupe pour lequel `sum(c)` vaut `3` et seul le premier groupe est conservé. L'évaluation de `a+2, max(c)` donne le résultat final $\{(\texttt{a'=3;mc=2})\}$.

2.2 Au coeur de SQL

Bien que l'évaluation du bloc soit intuitive, la sémantique de SQL est plus complexe qu'il n'y paraît de prime abord. Elle est décrite par le standard ISO [14] dont il n'est pas aisé de tirer parti car :

- il se compose de milliers de pages où un même aspect se trouve éparpillé en maints endroits du document : il est ainsi complexe de reconstituer toute l'information.
- les fonctionnalités sont délayées (la définition d'un "map" s'étale sur plusieurs pages)
- enfin, quoique abondant, le document présente trop souvent des aspects de manière sous-spécifiée.

En aucun cas le standard n'est exploitable, en l'état, pour offrir une sémantique formelle. Ceci explique, d'ailleurs, pourquoi de nombreux constructeurs en implantent certains traits à leur façon [1]. Nous nous en sommes, bien sûr, servi mais, afin de précisément appréhender la sémantique du langage, nous avons systématiquement confronté notre développement à des systèmes tels PostgreSQL et Oracle^{TM} qui comptent parmi les plus robustes et répandus.

r
a
1
NULL

s
a
NULL

t
a
1
NULL
NULL

t_1

a_1	b_1
1	1
1	2
1	3
1	4
1	5

a_1	b_1
1	6
1	7
1	8
1	9
1	10

a_1	b_1
2	1
2	2
2	3
2	4
2	5

a_1	b_1
2	6
2	7
2	8
2	9
2	10

a_1	b_1
3	1
3	2
3	3
3	4
3	5

a_1	b_1
4	6
4	7
4	8
4	9
4	10

t_2	
a_2	b_2
7	7
7	7

```
- Q1
select r.a+2 as b from r;
- Q2
select r.a from r where r.a not in (select s.a from s);
- Q3
select r.a from r where not exists
  (select * from s where s.a = r.a);
- Q4
select r.a from r except select s.a from s;
- Q5
select t.a,count(*) as c from t group by t.a;
- Q6
select a1, max(b1) from t1 group by a1;
- Q7(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a2) = k);
- Q8(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a1) = k);
```

```
- Q9(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having sum(1) = k);
- Q10(k), k =12
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a1)+sum(1+0*a2) = k);
- Q11(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a1+0*a2) = k);
- Q12(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*a1+0*b2) = k);
- Q13(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*b1+0*b2) = k);
- Q14(k)
select a1 from t1 group by a1 having exists
  (select a2 from t2 group by a2 having
                    sum(1+0*b1+0*a2) = k);
```

FIGURE 1: Un bestiaire de requêtes étranges.

Par la suite nous noterons $[\![q]\!]$ le résultat de l'évaluation de la requête q, () le constructeur de n-uplets, [ ] le constructeur de listes, { } celui d'ensembles et {| |} celui de multi ensembles. La Figure 1 contient un florilège de requêtes illustrant les aspects les plus subtils de SQL.

2.2.1 Valeurs nulles : NULL

La requête `Q1` renvoie {|(`b = 3`); (`b = NULL`)|} illustrant le fait que `NULL` est un élément absorbant pour les fonctions (excepté pour les fonctions Booléennes en PostgreSQL).

La prise en compte des valeurs nulles, dans les formules, se fait au moyen d'une logique trivaluée. Les trois requêtes suivantes, tirées de [12], illustrent que `NULL` n'est ni égal à, ni différent de quelque autre valeur (`NULL` compris) : comparer `NULL` à n'importe quelle expression renvoie toujours `unknown`. `Q2` renvoie un résultat vide. En effet, $[\![$`select s.a from s`$]\!]$ = {|(`s.a=NULL`)|}, donc $[\![$`r.a not in select s.a from s`$]\!]$ (`in` est le prédicat d'appartenance) est évalué à `not unknown`, soit `unknown`, pour tous les n-uplets, en particulier pour (`r.a=1`) et (`r.a=NULL`). Finalement, cette condition sera réduite à `false`, ainsi le résultat de `Q2` est vide. `Q3` renvoie {|(`r.a=1`); (`r.a=NULL`)|}. Considérons `subQ3` : `(select * from s where s.a = r.a)`, son évaluation conduit à un résultat vide pour tous les n-uplets (`r.a=x`), par suite $[\![$`exists (subQ3)`$]\!]$, où `exists` est le prédicat de non vacuité, vaut toujours `false` et $[\![$`not exists (subQ3)`$]\!]$ toujours `true`, donc (`r.a=1`) et (`r.a=NULL`)

sont dans le résultat de `Q3`. La requête `Q4` renvoie $\{(\texttt{r.a=1})\}$, car la différence ensembliste s'appuie sur l'égalité syntaxique standard pour tester l'égalité. Ainsi les n-uplets (`r.a=NULL`) et (`s.a=NULL`) sont égaux. L'évaluation des formules s'opère en logique trivaluée lorsque celles ci sont utilisées comme conditions de filtrage (clause `where` et `having`). Puis SQL plonge de la logique trivaluée à la logique Booléenne usuelle qui est finalement utilisée (`unknown` devient `false`). Ceci sera explicité en Figure 7 et 8 de la Section 3.

Enfin, la requête `Q5` retourne $\{\!|(\texttt{t.a=NULL,c=2});(\texttt{t.a=1,c=1})|\!\}$ et illustre le fait que `NULL`, qui est ni égal ni différent de `NULL` en logique trivaluée est égal à `NULL` dans le contexte du groupement. La sémantique que nous présentons en Section 3.2 rendra compte de tels comportements.

2.2.2 Requêtes corrélées

Intéressons nous à la manière dont SQL gère les environnements et l'évaluation d'expressions en présence d'agrégats et de requêtes imbriquées et corrélées. Pour évaluer des expressions simples (sans agrégats) un environnement contenant l'information relative aux attributs liés et leur valeurs associées suffit. Dans ce cas, par exemple `select a1, b1 from t1;`, un tel environnement consiste en l'unique n-uplet (`a1`=x,`b1`=y) où x et y varient dans les domaines de `a1` et `b1` respectivement.

L'évaluation d' expressions avec agrégats est plus subtile car un agrégat opère sur une liste de valeurs, chacune correspondant à un n-uplet. L'aspect le plus délicat est de comprendre comment une telle liste, que nous appellerons désormais *contexte d'évaluation*, est construite. La Section 10.9 du document ISO [14] (< aggregate functions >, *how to retrieve the rows* – page 545) du standard aurait dû nous éclairer. En vain ! Nous nous sommes donc appuyées sur notre développement SQL_{Coq} qui permet d'exécuter des requêtes (contrairement à [6]) afin de confronter notre sémantique mécanisée à celle de systèmes tels que PostgreSQL et Oracle^{TM}. Le recours à une *sémantique mécanisée et exécutable* a été essentiel afin d'examiner tous les sous cas en détail. Cette tâche, quoique chronophage, nous a permis d'obtenir l'ensemble de requêtes pertinentes sur le plan sémantique de la Figure 1. Dans tous les cas nous avons obtenu les même résultats pour les trois systèmes. Détaillons dans ce qui suit le comportement de SQL.

La requête `Q6` retourne $\{\!|(\texttt{a1=1,max=10});(\texttt{a1=2,max=10});(\texttt{a1=3,max=5});(\texttt{a1=4,max=10})|\!\}$. On comprend aisément que `max(b1)` est évalué pour chaque groupe (où `a1` est constant). Par exemple, le groupe T_1 pour lequel `a1=1` contient des n-uplets de la forme (`a1=1,b1=i`), où `i` varie de 1 à 10, ainsi `max(b1)` vaut 10. Le groupe où `a1=3` contient les n-uplets (`a1=3,b1=i`), où `i=1,...,5`, et `max(b1)` vaut 5. Dans ce cas, simple, un groupe correspond naturellement à un contexte d'évaluation. Nous dirons que le groupe a été *scindé* en n-uplets élémentaires.

La situation empire lorsqu'un agrégat est évalué dans une requête imbriquée. Comment SQL construit, dans ce cas, le contexte d'évaluation ? Les dernières requêtes `Q7` à `Q14` et la table `t2` de la Figure 1 ont été conçues afin de clarifier ce point. Notons qu'elles sont toutes corrélées, à l'exception de `Q7` et `Q9`, et ont la forme générale suivante :

```
select a1 from t1 group by a1 having exists (select a2 from t2 group by a2 having e=k);
```

Lorsqu'un aggrégat est présent sous plusieurs niveaux de groupements, dans une requête imbriquée comme c'est le cas pour `e` ci-dessus, plusieurs groupes font partie de *l'environnement*. Pour notre exemple, les groupes homogènes pour `a1` sont :

$$\mathscr{G}_1 = \left\{ \bigcup_{i=1}^{10} \{\!|(\texttt{a1=1;b1=i})|\!\}, \quad \bigcup_{i=1}^{10} \{\!|(\texttt{a1=2;b1=i})|\!\}, \quad \bigcup_{i=1}^{5} \{\!|(\texttt{a1=3;b1=i})|\!\}, \quad \bigcup_{i=6}^{10} \{\!|(\texttt{a1=4;b1=i})|\!\} \right\}$$

alors qu'un seul groupe homogène existe pour `a2`, $T_2 = \{\!|(\texttt{a2=7;b2=7});(\texttt{a2=7;b2=7})|\!\}$. Par conséquent, l'environnement global pour l'expression la plus interne `e`, est constitué de T_2 et d'un élément T_1 de $\mathscr{G}_1$. Nous noterons un tel environnement : $[T_2; T_1]$.

Comment combiner ces groupes pour obtenir le contexte d'évaluation correct ? Quels groupes doivent être scindés ? Considérons `Q7(`k`)`, où la condition de filtrage la plus interne est `sum(1+0*a2)` $= k$. Pour $k \neq 2$, $[\![$`Q7(`k`)`$]\!]$ est vide et pour $k = 2$ le résultat vaut $\bigcup_{i=1}^{i=4} \{|$`(a1=i)`$|\}$. L'expression `sum(1+0*a2)` compte le nombre de n-uplets dans le contexte d'évaluation et ce nombre vaut 2 pour tous les groupes T_1 de $\mathscr{G}_1$. La combinaison de T_1 et T_2 produit un contexte contenant deux (la cardinalité de T_2) n-uplets, quelle que soit la cardinalité de T_1. Nous en tirons la conclusion que lors de l'évaluation de `sum(1+0*a2)` $= k$, T_2 a été scindé tandis que T_1 n'est pas utilisé.

Examinons `Q8(`k`)`, similaire à `Q7(`k`)`, sauf la condition de filtrage qui est `sum(1+0*a1)` $= k$. Pour $k \notin \{5, 10\}$ $[\![$`Q8(`k`)`$]\!]$ est vide alors que $[\![$`Q8(5)`$]\!] = \{|$`(a1=3);(a1=4)`$|\}$ et $[\![$`Q8(10)`$]\!] = \{|$`(a1=1);(a1=2)`$|\}$. L'expression `sum(1+0*a1)` calcule la cardinalité de T_1. Lors de l'évaluation de `sum(1+0*a1)` $= k$, T_1 a été scindé tandis que T_2 n'est pas utilisé.

Constat 1. *Au sein du même environnement, $[T_2; T_1]$, SQL scinde T_1 ou T_2 pour construire son contexte d'évaluation. La façon dont ce contexte est construit est guidée par l'expression à évaluer déterminant quel groupe doit être scindé ou non : T_1 pour* `1+0*a1`*, et T_2 pour* `1+0*a2`*. Un tel choix, en l'espèce, est déterminé par les attributs (*`a1` *ou* `a2`*).*

Il est intéressant, à ce stade, de comprendre le comportement de SQL quand aucun attribut n'est présent dans l'expression sous l'agrégat comme dans `Q9(`k`)`. Quel groupe scinder ? Un tel groupe existe-t-il pour `sum(1)` ? `Q9(`k`)` produit le même résultat que `Q7(`k`)`, le groupe pertinent pour une constante est donc le plus interne : T_2.

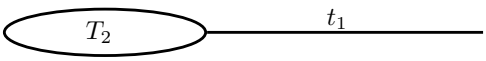
Constat 2. *Dans le même environnement,* `1+0*a2` *est égal à* `1` *et* `1+0*a1` *non puisque ces expressions sont calculées dans des contextes d'évaluations différents : sous un agrégat, en SQL, les égalités arithmétiques usuelles ne sont plus valides !*

Poursuivons notre analyse avec `Q10(`k`)` : qu'en est-il si les expressions `1+0*a1` et `1+0*a2` sont évaluées dans le même environnement où `1+0*a1` et `1+0*a2` apparaissent sous des agrégats *distincts* ? Il n'y a aucun groupe unique et naturellement pertinent. Quand $k \notin \{7, 12\}$ $[\![$`Q10(`k`)`$]\!]$ est vide alors que $[\![$`Q10(7)`$]\!] = \{|$`(a1=3);(a1=4)`$|\}$ et $[\![$`Q10(12)`$]\!] = \{|$`(a1=1);(a1=2)`$|\}$. Les deux expressions `1+0*a1` et `1+0*a2` ont été évaluées *indépendemment*, la première dans un contexte où T_1 a été scindé, la seconde dans un contexte ou c'est T_2 qui a été scindé.

Constat 3. *SQL autorise deux sous expressions d'une* même *expression à être évaluées dans différents contextes ! Ce qui est peu orthodoxe.*

Qu'advient il lorsque `1+0*a1` et `1+0*a2` apparaissent sous le *même* agrégat comme illustré par `Q11(`k`)` ? Pour k=2, $[\![$`Q11(2)`$]\!]$ vaut $\bigcup_{i=1}^{i=4} \{|$`(a1=i)`$|\}$, autrement $[\![$`Q11(`k`)`$]\!]$ est vide.

Constat 4. *T_2, le groupe pertinent le plus interne, a été scindé. T_1 a été aplati en un de ses éléments t_1 puisque seule sa partie homogène,* `a1`*, est utilisée par l'évaluation.*



Le lecteur aura noté que, jusqu'alors, les expressions présentes sous un agrégat mettent uniquement en jeu des attributs groupants. Que fait SQL si tel n'est pas le cas ? `Q12(`k`)` qui contient `sum(1+0*a1+0*b2)` se comporte exactement comme `Q11(`k`)`. `Q13(`k`)` contient `sum(1+0*b1+0*b2)`, conformément au Standard elle est *mal formée* et n'est pas évaluée. La raison réside dans le fait que deux attributs non groupants provenant de deux niveaux d'imbrication différents apparaissent sous l'agrégat. La dernière requête de ce bestiaire, `Q14(`k`)`, contenant l'expression `sum(1+0*b1+0*a2)`, est aussi mal formée et n'est pas évaluée. Cependant, nous aurions pu penser qu'elle eût pu être acceptée et évaluée dans le contexte suivant :

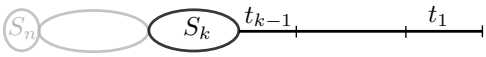
2.3 En résumé

Nous synthétisons au travers des définitions précisées d'environnements complétés et de contexte d'évaluation, les leçons tirées au cours de notre périple au coeur de SQL.

Environnements complétés Un environnement complété, $\mathscr{E} = [S_n; ...; S_1]$, est une pile de *tranches*. Chaque tranche correspond à un *niveau d'imbrication* i, le niveau le plus interne étant en tête de pile et de la forme $S = (A, G, T)$, où A (noté également $A(S)$) contient les attributs relevant de ce niveau d'imbrication, c'est à dire les noms introduits dans la sous requête à ce niveau [1] ; G les expressions de groupement introduites dans la clause `group by` (noté également $G(S)$) ; et T une liste non vide de n-uplets [2] (noté le cas échéant $T(S)$). Quand cela s'avérera utile nous adopterons la notation $\mathscr{E} = (A, G, T) :: \mathscr{E}'$ pour mettre en évidence la tête.

Contextes d'évaluation Lorsque e est une constante, la liste des n-uplets $T(S_n)$ provient de la tranche la plus interne de l'environnement $\mathscr{E} = [S_n; ...; S_1]$. Dans le cas où tous les attributs de e sont introduits au même niveau i, la liste est simplement $T(S_i)$. Lorsque les attributs de e appartiennent à au moins deux niveaux distincts, le plus interne (*i.e.*, celui d'indice plus élevé) étant S_k, nous sommes confrontés à deux cas :

- soit l'expression n'est pas bien formée (*cf* `Q13` et `Q14`), car e contient une expression de $T(S_j), j < k$ qui n'est pas groupée.
- soit l'expression e est exactement construite avec les attributs correspondant au k-ième niveau et les expressions groupantes [3] de niveaux externes $k-1, \ldots, 1$. Dans ce cas, soit t_j le n-uplet représentant pour chaque $T(S_j)$, $j < k$, la liste des n-uplets pertinents est faite de la concaténation $(t; t_{k-1}; \ldots; t_1)$, où t varie dans $T(S_k)$.



Nous présentons dans la section qui suit SQL_{Coq}, l'internalisation en Coq du fragment de SQL considéré, sa syntaxe ainsi que sa sémantique mécanisées.

3 SQL_{Coq} : une mécanisation de SQL en Coq

La syntaxe de SQL_{Coq} est présentée Figure 2, Figure 3 et Figure 4 où la partie gauche représente la syntaxe abstraite la droite son implantation en Coq.

1. Pour un `select from ...` ce sont les noms du `select`.
2. Quand la clause `group by` est présente c'est un groupe homogène autrement c'est un unique n-uplet.
3. Celle du `group by` pour ce niveau ; lorsqu'il n'y a pas de `group by` tous les attributs du niveau sont autorisés.

function ::= + | - | * | / | ... | *user defined fun*
aggregate ::= sum | avg | min | ... | *user defined ag*
value ::= string val | integer val | bool val | NULL
e^f ::= value | attribute | function($\overline{e^f}$)
e^a ::= e^f | aggregate(e^f) | function($\overline{e^a}$)

```
Inductive value : Set :=
  | String : string → value
  | Integer : Z → value
  | Bool : bool → value
  | NULL : value.
Inductive funterm : Type :=
   | F_Constant : value → funterm
   | F_Dot : attribute → funterm
   | F_Expr : symb → list funterm → funterm.
Inductive aggterm : Type :=
   | A_Expr : funterm → aggterm
   | A_agg : aggregate → funterm → aggterm
   | A_fun : symb → list aggterm → aggterm.
```

FIGURE 2: Expressions.

formula ::=
| formula (**and** | **or**) formula
| **not** formula
| **true**
| $p(\overline{e^a})$ $p \in$ *predicate*
| $p(\overline{e^a}, (\textbf{all} \mid \textbf{any})$ dom) $p \in$ *predicate*
| $\overline{e^a}$ **as** attribute **in** dom
| **exists** dom

```
Inductive conjunct : Type := And | Or.
Inductive quantifier : Type := All | Any.
Inductive select : Type :=
    Select_As : aggterm → attribute → select.
Inductive formula (dom : Type): Type :=
  | Conj : conjunct → formula dom → formula dom → formula dom
  | Not : formula dom → formula dom
  | True : formula dom
  | Pred : predicate → list aggterm → formula dom
  | Quant : list aggterm → predicate → quantifier → dom →
            formula dom
  | In : list select → dom → formula dom
  | Exists : dom → formula dom.
```

FIGURE 3: Formules, paramétrées par un domaine fini d'interprétation dom.

3.1 SQL_{Coq} : syntaxe

Nous supposons donnés des attributs, des fonctions, des agrégats. Les valeurs peuvent être des chaînes, des entiers et des booléens ainsi que la valeur spéciale NULL. Les expressions sont usuellement construites à partir des valeurs, tout d'abord sans agrégats e^f, et ensuite avec e^a. Les formules en SQL sont similaires aux formules de la logique du premier ordre à cela près qu'elles sont interprétées dans un domaine fini qui est syntaxiquement référencé par dom sur la Figure 3. Ces mêmes formules seront également utilisées dans le contexte de l'algèbre SQL_{Alg}.

SQL_{Coq} reflète la syntaxe de SQL. Cependant le programmeur SQL averti aura noté quelques différences. Tout d'abord, à des fins d'uniformité, nous imposons d'avoir l'intégralité du bloc `select from where group by having` (les clauses `where` et `group by having` ne sont plus option-

select_item ::= $*$ | $\overline{e^a \textbf{ as attribute}}$
query ::=
| table
| query (**union** | **intersect** | **except**) query
| **select** select_item
from $\overline{\text{from\_item}}$
where formula
group by $\overline{e^f}$
having formula
from_item ::= query($\overline{\text{attribute as attribute}}$)

```
Inductive select_item : Type :=
  Select_Star | Select_List : list select → select_item.
Inductive att_renaming : Type :=
   | Att_As : attribute → attribute → att_renaming.
Inductive att_renaming_item : Type :=
   | Att_Ren_Star
   | Att_Ren_List : list att_renaming → att_renaming_item.
Inductive group_by : Type :=
   Group_Fine | Group_By : list funterm → group_by.
Inductive set_op : Type := Union | Intersect | Except.
Inductive query : Type :=
  | Table : relname → query
  | Set : set_op → query → query → query
  | Select : (* select *) select_item  →
             (* from *) list from_item →
             (* where *) formula query →
             (* group by *) group_by →
             (* having *) formula query → query
with from_item : Type :=
 | From_Item : query → att_renaming_item → sql_from_item.
```

FIGURE 4: Syntaxe de SQL_{Coq}

nelles). Lorsque la clause `where` est absente en SQL, elle est forcée à `true`. La table ci dessous résume la manière dont le parseur prend en compte les différents cas pour traduire l'absence ou la présence de `group by` en une construction de groupement en SQL_{Coq}.

SQL			SQL_{Coq}
`aggregate` (in `select`)	`group by`	`having`	
?	`g`	?	`Group_By g`
✓	✗	?	`Group_By nil`
?	✗	✓	`Group_By nil`
✗	✗	✗	`Group_Fine`

La construction `Group_Fine` correspond à la partition la plus fine [4] et diffère de `Group_By` $[a_1, \ldots, a_n]$ où $[a_1, \ldots, a_n]$ est la liste de labels de la requête courante. `Group_By` $[a_1, \ldots, a_n]$ est utilisé en SQL_{Coq} pour encoder le `distinct` de SQL. `Group_By nil` correspond à la partition grossière. Nous imposons également un renommage obligatoire et explicite des attributs quand $*$ n'est pas employé. Ainsi, dans notre syntaxe, `select a, b from t;` s'exprime par `select a as a, b as b from (table t[*]) where true group by Group_Fine having true`. Un dernier point, plus subtil, qu'il convient de mentionner est la distinction faite entre e^f et e^a. Toutes deux sont des expressions mais les premières sont construites uniquement avec des fonctions ($\mathfrak{fn}$) et sont évaluées sur des *n-uplets* tandis que les secondes utilisent des agrégats sans imbrication [5] ($\mathfrak{ag}$) et sont dans ce cas évaluées sur des *collections de n-uplets*. Nous utilisons le même langage pour les formules qu'elles apparaissent dans la clause `where` (portant sur un n-uplet) ou `having` (portant sur des collections de n-uplets) simplement en identifiant chaque n-uplet avec le singleton lui correspondant. Enfin, nous n'autorisons pas les alias pour les requêtes ce cas étant pris en compte par un renommage des attributs.

$$
\begin{array}{lll}
[\![c]\!]^{\text{f}}_{\mathscr{E}} & = \text{c} & \text{si } c \text{ est une value} \\
[\![a]\!]^{\text{f}}_{[]} & = \textbf{default} & \text{si } a \text{ est un attribut} \\
[\![a]\!]^{\text{f}}_{(A,G,[])::\mathscr{E}} & = [\![a]\!]^{\text{f}}_{\mathscr{E}} & \\
[\![a]\!]^{\text{f}}_{(A,G,t::T):::\mathscr{E}} & = t.\text{a} & \text{si } a \in \ell(t) \\
[\![a]\!]^{\text{f}}_{(A,G,t::T):::\mathscr{E}} & = [\![a]\!]^{\text{f}}_{\mathscr{E}} & \text{si } a \notin \ell(t) \\
[\![\mathfrak{fn}(\overline{e})]\!]^{\text{f}}_{\mathscr{E}} & = [\![\mathfrak{fn}]\!]_f(\overline{[\![e]\!]^{\text{f}}_{\mathscr{E}}}) & \\
 & & \text{si } \mathfrak{fn} \text{ est une function,} \\
 & & \text{et } \overline{e} \text{ est une liste d'expressions simples}
\end{array}
$$

```
Definition env_type :=
   list (list attribute * group_by * list tuple).
Fixpoint interp_dot env (a : attribute) :=
 match env with
  | nil ⇒ default_value a
  | (sa, gb, nil) :: env' ⇒ interp_dot env' a
  | (sa, gb, t :: l) :: env' ⇒
    if a inS? labels t then (dot t a) else interp_dot env' a
  end.
Fixpoint interp_funterm  env t :=
 match t with
  | F_Constant c ⇒ c
  | F_Dot a ⇒ interp_dot env a
  | F_Expr f l ⇒
    interp_symb f (map (fun x ⇒ interp_funterm env x) l)
  end.
```

FIGURE 5: Sémantique des expressions simples.

3.2 SQL_{Coq} : sémantique

Cette section est le reflet en Coq des constats établis en Section 2. Étant donné un n-uplet t nous notons $\ell(t)$ les attributs présents dans t. Nous supposons l'existence d'une instance $[\![\_]\!]_{db}$ de la base de données définie comme une fonction des noms de relations vers des *multi-ensembles* de n-uplets [6] ainsi que d'interprétations prédéfinies, fixées $[\![\_]\!]_p$, pour les prédicats $\mathfrak{pr}$ [7], *i.e.*, une fonction de vecteurs de valeurs vers les Booléens, $[\![\_]\!]_a$ et $[\![\_]\!]_f$ pour les agrégats

4. La partition consistant de collections de singletons, un singleton pour chaque n-uplet.
5. e^a est de la forme : `avg(a)` ; `sum(a+b)+3` ; `sum(a+b)+avg(c+3)` mais pas de la forme `avg(sum(c)+a)`
6. Ces multi-ensembles sont munis d'opérateurs similaires à ceux des listes, tels que empty, map, filter, etc.
7. $\mathfrak{pr}$ correspond à `<, in` etc.

$$\frac{c \in \mathscr{V}}{\mathbb{B}_u(G, c)} \qquad \frac{e \in G}{\mathbb{B}_u(G, e)} \qquad \frac{\bigwedge_{\overline{e}} \mathbb{B}_u(G, e)}{\mathbb{B}_u(G, \mathfrak{fn}(\overline{e}))}$$

$$\frac{\mathbb{B}_u((A \cup \bigcup_{(A',G,T)\in\mathscr{E}} G), e)}{\mathbb{S}_e(A, \mathscr{E}, e)}$$

$$\frac{c \in \mathscr{V}}{\mathbb{F}_e(\mathscr{E}, c) = \mathscr{E}} \qquad \frac{e \notin \mathscr{V}}{\mathbb{F}_e([], e) = \mathtt{undefined}}$$

$$\frac{e \notin \mathscr{V} \quad \mathbb{F}_e(\mathscr{E}, e) = \mathscr{E}'}{\mathbb{F}_e(((A, G, T) :: \mathscr{E}), e) = \mathscr{E}'}$$

$$\frac{\mathbb{F}_e(\mathscr{E}, e) = \mathtt{undefined} \quad \mathbb{S}_e(A, \mathscr{E}, e)}{\mathbb{F}_e(((A, G, T) :: \mathscr{E}), e) = (A, G, T) :: \mathscr{E}}$$

$$[\![\mathfrak{fn}(\overline{e})]\!]^{\mathrm{a}}_{\mathscr{E}} = [\![\mathfrak{fn}]\!]_f(\overline{[\![e]\!]^{\mathrm{a}}_{\mathscr{E}}})$$

$$[\![\mathfrak{ag}(e)]\!]^{\mathrm{a}}_{\mathscr{E}} = [\![\mathfrak{ag}]\!]_a \left(\overline{[\![e]\!]^{\mathrm{f}}_{((A,G,[t])::\mathscr{E}')}}\right)_{t\in T} \text{ ssi } \mathbb{F}_e(\mathscr{E}, e) = (A, G, T) :: \mathscr{E}'$$

```
Fixpoint (* (𝔹_u(G, f)) *) is_built_upon G f :=
 match f with
  | F_Constant _ ⇒ true
  | F_Dot _ ⇒ f inS? g
  | F_Expr s l ⇒ (f inS? G) || forallb (is_built_upon G) l
 end.
Definition (* (𝕊_e(la, env, f)) *) is_a_suitable_env la env f :=
  is_built_upon
   (map (fun a ⇒ F_Dot a) la ++
   flat_map (fun slc ⇒ match slc with (_, G, _) ⇒ G end) env)
   f.
Fixpoint (* (𝔽_e(env, f)) *) find_eval_env env f :=
 match env with
  | nil ⇒ if is_built_upon nil f then Some nil else None
  | (la1, g1, l1) :: env' ⇒
    match find_eval_env env' f with
     | Some _ as e ⇒ e
     | None ⇒
        if is_a_suitable_env la1 env' f then Some env else None
    end
 end.
Fixpoint interp_aggterm env (ag : aggterm) :=
  match ag with
  | A_Expr ft ⇒ (* simple expression without aggregate *)
     interp_funterm env ft
  | A_fun f lag ⇒
    (* simple recursive call in order to evaluate independently
     the sub-expressions when the top symbol is a function *)
    interp_symb f (List.map (fun x ⇒ interp_aggterm env x) lag)
  | A_agg ag ft ⇒
    let env' :=
        if is_empty (att_of_funterm ft)
        then (* expression under the aggregate is a constant *)
             Some env
        else (* find the outermost suitable level *)
             find_eval_env env ft in
    let lenv :=
        match env' with
          | None | Some nil ⇒ nil
          | Some ((la1, g1, l1) :: env'') ⇒
              (* outermost group is split into *)
               map (fun t1 ⇒ (la1, g1, t1 :: nil) :: env'') l1
        end in
    interp_aggregate ag
       (List.map (fun e ⇒ interp_funterm e ft) lenv)
  end.
```

FIGURE 6: Sémantique des expressions complexes (avec agrégats).

$\mathfrak{ag}$ et fonctions $\mathfrak{fn}$ respectivement [8]. Comme nous l'avons montré en Section 2, les expressions (complexes) apparaissant au sein de sous-requêtes potentiellement corrélées sont évaluées dans un environnement découpé en tranches, $\mathscr{E} = [S_n; ...; S_1]$ (ou $\mathscr{E} = (A, G, T) :: \mathscr{E}'$), le niveau le plus interne, n, correspondant à la première tranche. L'évaluation d'une entité syntaxique e de type x dans l'environnement $\mathscr{E}$ sera notée $[\![e]\!]^{\mathrm{x}}_{\mathscr{E}}$ (où x est f pour les expressions seulement construites avec des fonctions, a pour les expressions construites également avec des agrégats, b pour les formules et q pour les requêtes).

La sémantique des expressions simples est donnée Figure 5. La sémantique des expressions complexes, présentée Figure 6, mérite que l'on s'y attarde. Lorsque l'expression est gardée par un symbole de fonction, $\mathfrak{fn}(\overline{e})$, une simple descente récursive suffit. Lorsque l'expression est de la forme $\mathfrak{ag}(e)$, conformément à ce qui a été décrit en Section 2, il faut trouver le niveau d'imbrication adéquat afin de construire le groupe devant être scindé. Puis construire la liste de valeurs en évaluant e, et enfin évaluer $\mathfrak{ag}$ sur cette liste. Pour l'environnement $\mathscr{E}$=$[S_n; \ldots S_1]$, i est un niveau adéquat, exprimé par $\mathbb{S}_e(A(S_i), [S_{i-1}; \ldots; S_1], e)$ sur la Figure 6 dès lors que e est

8. a correspond à `sum`, `count` etc. et f : `+`,`*`,`-` etc.

$$
\begin{array}{lcl}
[\![f_1 \textbf{ and } f_2]\!]^{\mathrm{b}}_{\mathscr{E}} & = & [\![f_1]\!]^{\mathrm{b}}_{\mathscr{E}} \wedge [\![f_2]\!]^{\mathrm{b}}_{\mathscr{E}} \\
[\![f_1 \textbf{ or } f_2]\!]^{\mathrm{b}}_{\mathscr{E}} & = & [\![f_1]\!]^{\mathrm{b}}_{\mathscr{E}} \vee [\![f_2]\!]^{\mathrm{b}}_{\mathscr{E}} \\
[\![\textbf{not } f]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \neg [\![f]\!]^{\mathrm{b}}_{\mathscr{E}} \\
[\![\textbf{true}]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
[\![\mathfrak{pr}(\overline{e_i})]\!]^{\mathrm{b}}_{\mathscr{E}} & = & [\![\mathfrak{pr}]\!]_p(\overline{[\![e_i]\!]^{\mathrm{a}}_{\mathscr{E}}}) \\
[\![\mathfrak{pr}(\overline{e_i}, \textbf{ all } q)]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{l}{\text{ssi } [\![\mathfrak{pr}(\overline{e_i}, t)]\!]^{\mathrm{b}}_{\mathscr{E}} = \top^{\dagger} \text{ pour tout } t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}} \\
[\![\mathfrak{pr}(\overline{e_i}, \textbf{ any } q)]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{l}{\text{ssi } [\![\mathfrak{pr}(\overline{e_i}, t)]\!]^{\mathrm{b}}_{\mathscr{E}} = \top^{\dagger} \text{ pour un } t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}} \\
[\![\overline{e_i \textbf{ as } a_i} \textbf{ in } q]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{l}{\text{si } (\overline{a_i = [\![e_i]\!]^{\mathrm{a}}_{\mathscr{E}}}) \text{ appartient à}^{\dagger} \ [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}} \\
[\![\textbf{exists } q]\!]^{\mathrm{b}}_{\mathscr{E}} & = & \top \\
\multicolumn{3}{l}{\text{ssi } [\![q]\!]^{\mathrm{q}}_{\mathscr{E}} \text{ est non vide}}
\end{array}
$$

[†]Voir le paragraphe sur les `NULL`' en Section 3.2.

```
Hypothesis I : env_type → dom → bagT.
Fixpoint eval_formula env (f : formula) : Bool.b B :=
 match f with
 | Sql_Conj a f1 f2 ⇒
     (interp_conj B a)
       (eval_formula env f1) (eval_formula env f2)
 | Sql_Not f ⇒ Bool.negb B (eval_formula env f)
 | Sql_True ⇒ Bool.true B
 | Sql_Pred p l ⇒ interp_pred p (map (interp_aggterm env) l)
 | Sql_Quant qtf p l sq ⇒
   let lt := map (interp_aggterm env) l in
   interp_quant B qtf
     (fun x ⇒ let la := Fset.elements _ (labels T x) in
           interp_pred p (lt ++ map (dot T x) la))
     (Febag.elements _ (I env sq))
 | Sql_In s sq ⇒
   let p := (projection env (Select_List s)) in
   interp_quant B Exists_F
    (fun x ⇒ match Oeset.compare (OTuple T) p x with
              | Eq ⇒ if contains_null p
                     then unknown else Bool.true B
              | _ ⇒ if (contains_null p || contains_null x)
                    then unknown else Bool.false B
              end)
       (Febag.elements _ (I env sq))
  | Sql_Exists sq ⇒
     if Febag.is_empty _ (I env sq)
     then Bool.false B else Bool.true B
 end.
```

FIGURE 7: Sémantique des formules.

construite avec $G = A(S_i) \cup \bigcup_{j<i} G(S_j)$ ce qui est exprimé par $\mathbb{B}_u(G, e)$ sur la Figure 6. Lorsque e est une constante, le niveau le plus interne est choisi (ici n), sinon le candidat pertinent le plus externe est choisi $\mathbb{F}_e(\mathscr{E}, e)$. La sémantique des formules donnée en Figure 7, s'appuie sur la sémantique des expressions. La syntaxe étant paramétrée par un domaine `dom`, de la même façon la sémantique est paramétrée par l'évaluation de ce domaine. Ce qui s'exprime dans notre développement Coq par : `Hypothesis I : env_type → dom → bagT.`, et par, $[\![\_]\!]^{\mathrm{q}}_{\_}$, dans la définition formelle.

La sémantique des requêtes, $[\![\_]\!]^{\mathrm{q}}_{\_}$ est détaillée sur la Figure 8. Les opérateurs ensemblistes sont munis, par défaut, d'une sémantique multi-ensembles (ce qui correspond à `union all`, `intersect all` etc en SQL). La sémantique strictement ensembliste pour `sq` $=$ `q`$_1$ `op` `q`$_2$, est obtenue en appliquant l'opérateur d'élimination des doublons δ`(sq)` $=$ `select * from sq` $\overline{(a_i \texttt{ as } a_i)_{a_i \in \ell(sq)}}$ `group by` ℓ`(sq)`. Le cas le plus complexe est celui du `select from where group by having`. Informellement, une première étape consiste à évaluer la partie `from` puis à filtrer le résultat au moyen du `where`. Plus précisément, vérifier qu'un n-uplet t satisfait la condition `where` w dans le contexte $\mathscr{E}$ s'opère ainsi : w est évaluée relativement à un unique environnement. Ceci implique que t et $\mathscr{E}$ doivent être combinés au sein d'un unique environnement, $\mathscr{E}'$, tel que $[\![w]\!]^{\mathrm{f}}_{\mathscr{E}'}$ corresponde à l'évaluation de w, où les attributs a éléments de $\ell(t)$ sont lié à $t.a$, et où les attributs a éléments de $\bigcup_{S \in \mathscr{E}} A(S)$ sont liés grâce à $\bigcup_{S \in \mathscr{E}} A(T)$. C'est exactement ce qui est fait lorsque $\mathscr{E}' = (\ell(t), [], [t]) :: \mathscr{E}$.

La collection (intermédiaire) de n-uplets obtenue est ensuite partitionnée au moyen des expressions de groupement du `group by` G, conduisant à l'obtention d'une collection de collections de n-uplets : les groupes. Pour rendre compte du caractère optionnel du `group by`, la partition la plus fine est utilisée ce qui est noté par `Group_Fine` dans le développement Coq.

La manière de filtrer les groupes relativement au `having` h est similaire à ce que nous avons décrit pour le `where`, mis à part que des expressions complexes peuvent être présentes dans h. Lors de l'évaluation d'une expression de la forme `ag`(e) pour un groupe T, tous les n-uplets du groupe sont indispensables ; lors de l'évaluation d'une expression simple, n'importe quel n-

$$
\begin{array}{lcl}
[\![tbl]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![tbl]\!]_{db} \\
\quad \text{si } tbl \text{ est une table} & & \\
[\![q_1 \ \mathbf{union} \ q_2]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathscr{E}} \cup [\![q_2]\!]^{\mathrm{q}}_{\mathscr{E}} \\
[\![q_1 \ \mathbf{intersect} \ q_2]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathscr{E}} \cap [\![q_2]\!]^{\mathrm{q}}_{\mathscr{E}} \\
[\![q_1 \ \mathbf{except} \ q_2]\!]^{\mathrm{q}}_{\mathscr{E}} & = & [\![q_1]\!]^{\mathrm{q}}_{\mathscr{E}} \setminus [\![q_2]\!]^{\mathrm{q}}_{\mathscr{E}} \\
[\![\mathbf{select} \ \overline{e_i \ \mathbf{as} \ a_i} \ \mathbf{from} \ \overline{f_i} \ \mathbf{where} \ w & & \\
\qquad \mathbf{group\ by} \ G \ \mathbf{having} \ h]\!]^{\mathrm{q}}_{\mathscr{E}} = & & \\
\quad \left\{ \left| \overline{\left(a_i = [\![e_i]\!]^{\mathrm{a}}_{(\ell(T),G,T)::\mathscr{E}} \right)} \right| T \in \mathbb{F}_3 \right\} & & \\
\quad \text{si } \ F = \bowtie_i [\![f_i]\!]^{\mathrm{from}}_{\mathscr{E}} & & \\
\quad \text{et } F_1 = \left\{ \left| t \in F \ \middle| \ [\![w]\!]^{\mathrm{b}}_{(\ell(t),[],[t])::\mathscr{E}} = \top \right| \right\} & & \\
\quad \text{et } \mathbb{F}_2 \text{ est une partition}^{\dagger} \text{ de } F_1 \text{ selon } G & & \\
\quad \text{et } \mathbb{F}_3 = \left\{ \left| T \in \mathbb{F}_2 \ \middle| \ [\![h]\!]^{\mathrm{b}}_{(\ell(T),G,T)::\mathscr{E}} = \top \right| \right\} & & \\
[\![q(\overline{a_i \ \ \mathbf{as} \ \ b_i})]\!]^{\mathrm{from}}_{\mathscr{E}} & = & \\
\qquad \{\!| \overline{(b_i = c_i)} \mid \overline{(a_i = c_i)} \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}} |\!\} & &
\end{array}
$$

[†]Voir le paragraphe sur les NULL's en Section 3.2.

```
Fixpoint eval_sql_query env sq {struct sq} :=
match sq with
 | Sql_Table tbl ⇒ instance tbl
 | Sql_Set o sq1 sq2 ⇒
   if sql_sort sq1 =S?= sql_sort sq2
   then Febag.interp_set_op _ o
          (eval_sql_query env sq1) (eval_sql_query env sq2)
   else Febag.empty _
 | Sql_Select s lsq f1 gby f2  ⇒
  let elsq :=
  (* evaluation of the from part *)
        List.map (eval_sql_from_item env) lsq in
  let cc :=
  (* selection of the from part by formula f1, with old names *)
     Febag.filter _
       (fun t ⇒
         Bool.is_true _
           (eval_sql_formula eval_sql_query (env_t env t) f1))
             (N_product_bag elsq) in
  (* computation of the groups grouped according to gby *)
  let lg1 := make_groups env cc gby in
  (* discarding groups according the having clause f2 *)
  let lg2 :=
    List.filter
     (fun g ⇒
        Bool.is_true _
          (eval_sql_formula eval_sql_query (env_g env gby g) f2))
      lg1 in
  (* applying outermost projection w.r.t. the select part s *)
   Febag.mk_bag BTupleT
     (List.map (fun g ⇒ projection (env_g env gby g) s) lg2)
  end
(* evaluation of the from part *)
with eval_sql_from_item env x :=
  match x with
  | From_Item sqj sj ⇒
    Febag.map BTupleT BTupleT
    (fun t ⇒ projection (env_t env t)
                        (att_renaming_item_to_from_item sj))
    (eval_sql_query env sqj)
    end.
```

FIGURE 8: Sémantique des requêtes.

uplet de T suffit car T est homogène relativement au critère de groupement G. Par conséquent, l'environnement correct pour filtrer le groupe T par h dans $\mathscr{E}$ est $(\ell(T), G, T) :: \mathscr{E}$. Enfin, le `select` est appliqué qui conduit à une collection de n-uplets : le résultat final.

À propos des NULL's Au niveau des expressions,, les NULL's se comportent comme des éléments absorbants relativement aux fonctions et sont ignorés pour les agrégats (à l'exception du `count(*)` où ils contribuent pour 1. Ceci est exprimé dans notre formalisation en imposant des contraintes sur $[\![\_]\!]_a$ et $[\![\_]\!]_f$. Pour les formules, une logique trivaluée est utilisée. L'évaluation de $\mathtt{pr}(\overline{e})$ dans l'environnement $\mathscr{E}$ vaut `unknown` ssi il existe e_i dans $\overline{e}$ telle que $[\![e_i]\!]^{\mathrm{a}}_{\mathscr{E}} = \mathtt{NULL}$. La valeur `unknown` se distribue conformément aux règles classiques de la logique trivaluée. Les quantificateurs `all` et `any` sont vu respectivement comme la conjonction ou disjonction finie de la logique trivaluée. Enfin, $\overline{e \ \mathbf{as} \ a}$ **in** q est évaluée comme la conjonction finie $\bigwedge \overline{e = t.a}$ où t varie dans $[\![q]\!]^{\mathrm{q}}$, ce qui induit que dès lors que e ou $t.a$ est évalué à NULL, $\bigwedge \overline{e = t.a}$ vaut `unknown`. Tôt ou tard, lors de l'évaluation des requêtes, l'évaluation des formules ayant conduit à `unknown` est convertie à false. Notons également que bien que NULL ne soit ni égal ni différent de NULL ou de quelqu'autre valeur dans le contexte de formules, NULL est bien égal à NULL pour le groupement. Ceci est pris en compte sur la Figure 8 par une définition minutieuse de partition et de `make_groups` dans le développement Coq.

4 $\mathbf{SQL_{Alg}}$: une algèbre relationnelle mécanisée en Coq

4.1 L'algèbre relationnelle en bref

L'algèbre relationnelle (étendue) telle qu'elle est présentée dans les ouvrages de référence [8], est constituée des opérateurs σ (sélection), π (projection) and $\bowtie$ (jointure) étendue avec l'opérateur γ (groupement) complétée avec les opérateurs ensemblistes, intersection, union et différence. Nous nous concentrons sur les quatre premiers opérateurs dont nous rappelons la sémantique dans ce qui suit : $q := r \mid \sigma_f(q) \mid \pi_S(q) \mid q \bowtie q \mid \gamma_{g,ag}(q)$.

Les relations de base, r sont des expressions. L'opérateur de sélection permet de filtrer des collections de n-uplets, ne retenant que ceux qui satisfont la condition f. La sémantique de cet opérateur est $[\![\sigma_f(q)]\!] = \{t \mid t \in [\![q]\!] \wedge [\![f]\!]\{x \to t\}\}$ où $[\![f]\!]\{x \to t\}$ signifie "t satisfait la formule $[\![f]\!]$", x étant l'unique variable libre de $[\![\mathrm{f}]\!]$.

L'opérateur de projection, de la forme π_S, opère sur toutes les expressions, q, dont la sorte contient le sous ensemble d'attributs S. Sa sémantique est donnée par $[\![\pi_S(q)]\!] = \{t|_S \mid t \in [\![q]\!]\}$ où la notation $t|_S$ représente la restriction du n-uplet t aux seuls attributs de S.

L'opérateur de jointure, noté $\bowtie$, prend en argument deux expressions q_1 et q_2 dont les sortes respectives sont V et W, et permet de combiner les n-uplets provenant de chaque opérande. Sa sémantique est donnée par $[\![q_1 \bowtie q_2]\!] = \{t \mid \exists v \in [\![q_1]\!], \exists w \in [\![q_2]\!], t|_V = v \wedge t|_W = w\}$.

Le dernier opérateur, $\gamma_{g,ag}$, est défini comme suit dans [8], « *operator $\gamma_{g,ag}$ partitions the tuples of q into groups. Each group consists of all tuples having one particular assignment of values to the grouping attributes in g. If there are no grouping attributes, the entire relation q is one group. For each group, one tuple consisting of the grouping attributes' values for that group and the aggregations, over all tuples of that group, for the aggregated attributes in ag is produced* ». Sa définition formelle est donnée en Figure 9 où f vaut true.

4.2 $\mathbf{SQL_{Alg}}$ syntaxe et sémantique

L'algèbre présentée dans [8], ne rend compte ni des formules de la clause `having` ni des expressions complexes (le groupement s'opère seulement avec des attributs et les agrégats ne sont appliqués qu'à un seul attribut) ni des environnements. Afin d'accueillir SQL, notre modélisation est plus expressive autorisant le groupement à partir d'expressions simples et permettant l'utilisation d'expressions complexes e^a dans les projections. Enfin, de sorte à considérer les conditions du `having`, qui opèrent directement sur des groupes, $\mathrm{SQL_{Alg}}$ étend l'algèbre décrite dans [8] en ajoutant un paramètre supplémentaire à γ : la condition de la clause `having`.

$$\begin{aligned} \mathrm{Q} ::= {} & \mathit{table} \mid \mathrm{Q}\ (\textbf{union} \mid \textbf{intersect} \mid \textbf{except})\ \mathrm{Q} \mid \mathrm{Q} \bowtie \mathrm{Q} \\ & \mid \pi_{(\overline{e^a\ \textbf{as}\ \mathrm{attribute}})}(\mathrm{Q}) \mid \sigma_{\mathsf{formula}}(\mathrm{Q}) \mid \gamma_{(\overline{e^a\ \textbf{as}\ \mathrm{attribute}}, \overline{e^f}, \mathsf{formula})}(\mathrm{Q}) \end{aligned}$$

Nous donnons en Figure 9 la sémantique de $\mathrm{SQL_{Alg}}$. Les expressions (simples et complexes) ainsi que les formules (dont paramètre de domaine `dom` est ici celui des requêtes algébriques) sont partagées avec $\mathrm{SQL_{Coq}}$. Pour définir la sémantique des expressions, la notion d'environnement est nécessaire pour les mêmes raisons qu'en $\mathrm{SQL_{Coq}}$: prendre en compte la corrélation. Ainsi, les environnements de $\mathrm{SQL_{Alg}}$ sont les mêmes qu'en $\mathrm{SQL_{Coq}}$. Il faut noter que $\bowtie$ est le véritable opérateur de jointure naturelle et que γ peut être vu comme une version dégénérée de `select from where group by having`, dans laquelle la condition `where` est absente (ou vaut `true`). Nous sommes, à présent, en mesure de relier, formellement, $\mathrm{SQL_{Coq}}$ et $\mathrm{SQL_{Alg}}$.

4.3 $\mathbf{SQL_{Coq}} \equiv \mathbf{SQL_{Alg}}$

La Figure 10, présente $\mathbb{T}^{\mathrm{q}}(\_)$ une traduction de $\mathrm{SQL_{Coq}}$ vers $\mathrm{SQL_{Alg}}$, ainsi que la traduction inverse $\mathbb{T}^{\mathrm{Q}}(\_)$. Toutes deux s'appuient sur les traductions auxiliaires ($\mathbb{T}^{\mathrm{f}}(\_)$, resp. $\mathbb{T}^{\mathrm{F}}(\_)$) qui effectuent une simple traversée des formules en traduisant les requêtes qu'elles contiennent.

$$[\![tbl]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![tbl]\!]_{db} \qquad \text{si } tbl \text{ est une table}$$
$$[\![q_1 \ \mathbf{union} \ q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \cup [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}}$$
$$[\![q_1 \ \mathbf{intersect} \ q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \cap [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}}$$
$$[\![q_1 \ \mathbf{except} \ q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \setminus [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}}$$

$$[\![q_1 \bowtie q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} = \left\{ \left| \left(\overline{a_i = c_i}, \overline{b_j = d_j}\right) \right| \begin{array}{l} (\overline{a_i = c_i}) \in [\![q_1]\!]^{\mathrm{Q}}_{\mathscr{E}} \wedge \\ (\overline{b_j = d_j}) \in [\![q_2]\!]^{\mathrm{Q}}_{\mathscr{E}} \wedge \\ (\forall\ i, j,\ \ a_i = b_j \implies c_i = d_j) \end{array} \right| \right\}$$

$$[\![\pi_{(\overline{e_i \ \mathbf{as} \ a_i})}(q)]\!]^{\mathrm{Q}}_{\mathscr{E}} = \{\!|(\overline{a_i = [\![e_i]\!]^{\mathrm{a}}_{(\ell(t),[],[t])::\mathscr{E}}}) \mid t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}}|\!\}$$
$$[\![\sigma_f(q)]\!]^{\mathrm{Q}}_{\mathscr{E}} = \{\!|t \in [\![q]\!]^{\mathrm{q}}_{\mathscr{E}} \mid [\![f]\!]^{\mathrm{b}}_{(\ell(t),[],[t])::\mathscr{E}} = \top|\!\}$$

$$[\![\gamma_{(\overline{e_j \ \mathbf{as} \ a_j}, \overline{e_i}, f)}(q)]\!]^{\mathrm{q}}_{\mathscr{E}} = \left\{ \left| \overline{\left(a_j = [\![e_j]\!]^{\mathrm{a}}_{(\ell(T),\overline{e_i},T)::\mathscr{E}}\right)} \right| T \in \mathbb{F}_3 \right\}$$
$$\text{si } \mathbb{F}_2 \text{ est une partition de } [\![q]\!]^{\mathrm{Q}}_{\mathscr{E}} \text{ selon } \overline{e_i} \text{ et } \mathbb{F}_3 = \left\{ \left| T \in \mathbb{F}_2 \ \right| [\![f]\!]^{\mathrm{b}}_{(\ell(T),\overline{e_i},T)::\mathscr{E}} = \top \ \right| \right\}$$

FIGURE 9: Sémantique de $\mathrm{SQL}_{\mathrm{Alg}}$

$$\mathbb{T}^{\mathrm{q}}(tbl) = tbl$$
$$\mathbb{T}^{\mathrm{q}}(q_1 \ \mathbf{op} \ q_2) = \mathbb{T}^{\mathrm{q}}(q_1) \ \mathbf{op} \ \mathbb{T}^{\mathrm{q}}(q_2) \qquad \text{où } \mathbf{op} \in \{ \mathbf{union}, \mathbf{intersect}, \mathbf{except} \}$$
$$\mathbb{T}^{\mathrm{q}}(\mathbf{select} \ \overline{e_i \ \mathbf{as} \ a_i} \ \mathbf{from} \ \overline{f_i} \ \mathbf{where} \ w) = \pi_{(\overline{e_i \ \mathbf{as} \ a_i})}(\sigma_{\mathbb{T}^{\mathrm{f}}(w)}(\bowtie_i \overline{\mathbb{T}^{\mathrm{from}}(f_i)}))$$
$$\mathbb{T}^{\mathrm{q}}(\mathbf{select} \ \overline{e_i \ \mathbf{as} \ a_i} \ \mathbf{from} \ \overline{f_i} \ \mathbf{where} \ w \ \mathbf{group\ by} \ G \ \mathbf{having} \ h) = \gamma_{(\overline{e_i \ \mathbf{as} \ a_i}, G, \mathbb{T}^{\mathrm{f}}(h))}(\sigma_{\mathbb{T}^{\mathrm{f}}(w)}(\bowtie_i \overline{\mathbb{T}^{\mathrm{from}}(f_i)}))$$

$$\mathbb{T}^{\mathrm{from}}(q\overline{(a_i \ \mathbf{as} \ b_i)}) = \pi_{\overline{(a_i \ \mathbf{as} \ b_i)}}(\mathbb{T}^{\mathrm{q}}(q))$$

$$\mathbb{T}^{\mathrm{Q}}(tbl) = tbl$$
$$\mathbb{T}^{\mathrm{Q}}(q_1 \ \mathbf{op} \ q_2) = \mathbb{T}^{\mathrm{Q}}(q_1) \ \mathbf{op} \ \mathbb{T}^{\mathrm{Q}}(q_2) \qquad \text{où } \mathbf{op} \in \{ \mathbf{union}, \mathbf{intersect}, \mathbf{except} \}$$
$$\mathbb{T}^{\mathrm{Q}}(q_1 \bowtie q_2) = \mathbf{select} \ (\overline{a'_1 \ \mathbf{as} \ a_1}_{a_1 \in \ell(q_1)}, \overline{a'_2 \ \mathbf{as} \ a_2}_{a_2 \in \ell(q_2) \setminus \ell(q_1)}) \mathbf{from} \ [\mathbb{T}^{\mathrm{Q}}(q_1)(\overline{a_1 \ \mathbf{as} \ a'_1}); \mathbb{T}^{\mathrm{Q}}(q_2)(\overline{a_2 \ \mathbf{as} \ a'_2})]$$
$$\mathbf{where} \ (\overline{a'_1 = a'_2})_{a_1 \in \ell(q_1), a_2 \in \ell(q_2), a_1 = a_2} \qquad \text{où } \overline{a'_1} \text{ et } \overline{a'_2} \text{ sont des noms frais}$$
$$\mathbb{T}^{\mathrm{Q}}(\pi_{(\overline{e \ \mathbf{as} \ a})}(q)) = \mathbf{select} \ (\overline{e \ \mathbf{as} \ a}) \ \mathbf{from} \ [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \ \mathbf{as} \ a})]$$
$$\mathbb{T}^{\mathrm{Q}}(\sigma_f(q)) = \mathbf{select} \ * \ \mathbf{from} \ [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \ \mathbf{as} \ a})] \mathbf{where} \ \mathbb{T}^{\mathrm{F}}(f)$$
$$\mathbb{T}^{\mathrm{Q}}(\gamma_{(\overline{e \ \mathbf{as} \ a}, G, f)}(q)) = \mathbf{select} \ (\overline{e \ \mathbf{as} \ a}) \ \mathbf{from} \ [\mathbb{T}^{\mathrm{Q}}(q)(\overline{a \ \mathbf{as} \ a})] \ \mathbf{group\ by} \ G \ \mathbf{having} \ \mathbb{T}^{\mathrm{F}}(f)$$

FIGURE 10: Traductions entre $\mathrm{SQL}_{\mathrm{Coq}}$ et $\mathrm{SQL}_{\mathrm{Alg}}$.

Les expressions, partagées par $\mathrm{SQL}_{\mathrm{Alg}}$ et $\mathrm{SQL}_{\mathrm{Coq}}$, sont inchangées par les traductions. Ces traductions sont correctes dès lors qu'elles sont appliquées sur des instances de bases de données et des requêtes « raisonnables » :

Définition 1. *Une instance de bases de données* $[\![\_]\!]_{db}$ *est* bien sortée *si et seulement si tous les n-uplets d'une même relation possèdent les mêmes labels :*
$\forall r, t_1, t_2, \quad t_1 \in [\![r]\!]_{db} \wedge t_2 \in [\![r]\!]_{db} \implies \ell(t_1) = \ell(t_2)$.

Définition 2. *Une requête SQL*$_{Coq}$ *sq est* bien formée *si et seulement si tous les labels introduits dans la clause* `from` *sont disjoints deux à deux et si toutes ses sous-requêtes sont récursivement bien formées.*

$$\frac{}{\mathbb{W}^q(\overline{tbl})} \ \textit{si tbl est une table} \qquad \frac{\mathbb{W}^q(q_1) \quad \mathbb{W}^q(q_2)}{\mathbb{W}^q(q_1 \ \mathbf{union} \ q_2)} \qquad \frac{\mathbb{W}^q(q_1) \quad \mathbb{W}^q(q_2)}{\mathbb{W}^q(q_1 \ \mathbf{intersect} \ q_2)} \qquad \frac{\mathbb{W}^q(q_1) \quad \mathbb{W}^q(q_2)}{\mathbb{W}^q(q_1 \ \mathbf{except} \ q_2)}$$

$$\frac{\mathbf{disjoint}\overline{\{b_i\}_i} \quad \bigwedge_i \mathbb{W}^q(q_i) \quad \mathbb{W}^f(w) \quad \mathbb{W}^f(h)}{\mathbb{W}^q(\mathbf{select} \ s \ \mathbf{from} \ \overline{q_i(\overline{a_i \ \mathbf{as} \ b_i})} \ \mathbf{where} \ w \ \mathbf{group\ by} \ G \ \mathbf{having} \ h)}$$

$$\frac{\mathbb{W}^f(f_1) \quad \mathbb{W}^f(f_2)}{\mathbb{W}^f(f_1 \textbf{ and } f_2)} \qquad \frac{\mathbb{W}^f(f_1) \quad \mathbb{W}^f(f_2)}{\mathbb{W}^f(f_1 \textbf{ or } f_2)} \qquad \frac{\mathbb{W}^f(f)}{\mathbb{W}^f(\textbf{not } f)} \qquad \frac{}{\mathbb{W}^f(\textbf{true})} \qquad \frac{}{\mathbb{W}^f(\mathfrak{pr}(\overline{e}_i))}$$

$$\frac{\mathbb{W}^q(q)}{\mathbb{W}^f(\textbf{exists } q)} \qquad \frac{\mathbb{W}^q(q)}{\mathbb{W}^f(\mathfrak{pr}(\overline{e}_i, \textbf{all } q))} \qquad \frac{\mathbb{W}^q(q)}{\mathbb{W}^f(\mathfrak{pr}(\overline{e}_i, \textbf{any } q))} \qquad \frac{\mathbb{W}^q(q)}{\overline{e_i \textbf{ as } a_i} \textbf{ in } q}$$

Quand ces conditions sont remplies, SQL_{Coq} et SQL_{Alg} sont sémantiquement équivalents :

Théorème 1. *Soit* $[\![\_]\!]_{db}$ *une instance* bien sortée*, sq une requête* SQL_{Coq} *et aq une requête* SQL_{Alg} *:*

$$\forall \mathscr{E}, sq, \mathbb{W}^q(sq) \implies [\![\mathbb{T}^q(sq)]\!]^Q_{\mathscr{E}} = [\![sq]\!]^q_{\mathscr{E}}$$
$$\forall \mathscr{E}, aq, \quad [\![\mathbb{T}^Q(aq)]\!]^q_{\mathscr{E}} = [\![aq]\!]^Q_{\mathscr{E}}$$

La preuve procède par induction structurelle mutuelle sur les requêtes et formules en utilisant leurs tailles respectives comme mesure. Dans la preuve de correction de $\mathbb{T}^q(\_)$, l'hypothèse $\mathbb{W}^q(sq)$ assure que le produit Cartésien et la jointure naturelle coïncident. Cette hypothèse de bonne formation est essentielle pour assurer le résultat ce qui donne un éclairage intéressant au fait qu'en réalité la clause `from` de SQL se comporte vraiment comme un produit. L'hypothèse de bonne sorte assure une forme de typage faible : les n-uplets d'une même relation ont les mêmes attributs que ceux déclarés lors de la définition de la relation. Ceci est toujours le cas dans la réalité. Cette hypothèse permet de raisonner sur les attributs de manière globale, en calculant statiquement les labels de la requête pour évaluer celle-ci.

5 Conclusions

De longue date, la communauté bases de données s'est employée à définir une sémantique formelle pour SQL. Cet article propose une sémantique *exécutable, mécanisée* en Coq pour un fragment réaliste du langage. *La mécanisation* au sein d'un assistant à la preuve ouvre d'intéressantes perspectives. Les fonctions récursives, en Coq, devant être totales, une fois fixée la syntaxe, la sémantique doit être « totalement » définie : aucun détail ne peut être occulté. Tous les cas devant être considérés, ceci nous a permis de proposer le florilège de requêtes étranges de la Figure 1 qui pourraient servir de base à un « benchmark » sémantique pour tester différentes implantations du langage. Produire une sémantique *exécutable* permet de se convaincre de sa correction puisqu'il est alors possible de la confronter aux systèmes existants. Ce que nous avons fait en testant systématiquement le comportement de SQL_{Coq} vis à vis de celui de PostgreSQL et Oracle™.

Définir formellement la sémantique de SQL et la relier rigoureusement à l'algèbre relationnelle, nous a permis d'établir le premier résultat, à notre connaissance, d'équivalence entre les deux langages, recouvrant ainsi les équivalences algébriques qui fondent les optimisations opérées par le compilateur SQL. Ces équivalences sont formellement prouvées dans [3]. Quoique le sachant, nous avons eu la confirmation que, SQL ayant été initialement conçu comme un langage dédié n'ayant pas vocation à être Turing-complet, l'ajout, au cours du processus de standardisation, de traits nouveaux nécessaires, l'a inexorablement écarté de ses fondement élégants. En reliant formellement SQL et l'algèbre relationnelle nous souhaitions, humblement, rendre hommage aux pères fondateurs qui jetèrent les bases des SGBDR.

Dans une version préliminaire de ce travail nous avions proposé une sémantique purement ensembliste. Puis nous nous sommes attelées à la version multi-ensemble et avons été agréablement surprises de découvrir que cette évolution se passe sans heurts. En parfaite contradiction avec la croyance, répandue dans le monde des bases de données, que *la* difficulté consiste à doter SQL d'une sémantique multi-ensemble. La prise en compte des `NULL`'s est souvent considérée comme difficile. Ceci est dû au fait que SQL ne les traite pas uniformément selon le contexte. La

logique trivaluée nous a suffi à en rendre compte. Le véritable défi a consisté analyser scrupuleusement la gestion des environnements opérée par SQL en présence de requêtes imbriquées et corrélées et de combiner les quatre aspects afin de fournir une sémantique fidèle. Le document de standardisation ISO ne nous a guère aidées, Coq, en revanche, a été un maître intraitable dont l'aide nous fût des plus précieuses.

Notre objectif à long terme est l'obtention d'un compilateur de SQL vérifié. Le travail présenté dans cet article fourni un analyseur sémantique certifié, étape cruciale de la chaîne de compilation. Nous envisageons d'étendre le fragment considéré à la construction `order by`. Les aspects relatifs aux couches basses d'un moteur d'exécution de SQL sont présentés dans [4] qui propose une spécification et une implantation certifiée des opérateurs physiques. Il nous reste à traiter des aspects liés à l'optimiseur du compilateur.

Références

[1] T. Arvin. *Comparison of different SQL's implementations*, 2017.

[2] J. S. Auerbach, M. Hirzel, L. Mandel, A. Shinnar, and J. Siméon. Handling environments in a nested relational algebra with combinators and an implementation in a verified query compiler. In *SIGMOD Conference, Chicago, USA*, 2017.

[3] V. Benzaken, E. Contejean, and S. Dumbrava. A Coq Formalization of the Relational Data Model. In *23rd European Symposium on Programming (ESOP)*, 2014.

[4] V. Benzaken, É. Contejean, C. Keller, and E. Martins. A Coq formalisation of SQL's execution engines. In *Int. Conf. on Interactive Theorem Proving (ITP 2018)*, Oxford, UK, July 2018.

[5] S. Ceri and G. Gotlob. Translating SQL into relational algebra : Optimisation, semantics, and equivalence of SQL queries. *IEEE Trans., on Software Engineering*, SE-11 :324–345, April 1985.

[6] S. Chu, K. Weitz, A. Cheung, and D. Suciu. HoTTSQL : Proving query rewrites with univalent SQL semantics. In *PLDI*. ACM, 2017.

[7] S. Cluet and G. Moerkotte. Nested queries in object bases. In *Database Programming Languages, New York City, USA*, 1993.

[8] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[9] C. Gonzalia. Towards a formalisation of relational database theory in constructive type theory. In *RelMiCS*, 2003.

[10] C. Gonzalia. *Relations in Dependent Type Theory*. PhD thesis, Chalmers Göteborg Univ., 2006.

[11] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc., of the 26th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Beijing, China*, 2007.

[12] P. Guagliardo and L. Libkin. A formal semantics of SQL queries, its validation, and applications. *PVLDB*, 11(1) :27–39, 2017.

[13] R. Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2016.

[14] ISO/IEC. Information technology - database languages - SQL - part 2 : Foundation (SQL/foundation), 2006. Final Commitee Draft.

[15] X. Leroy. A formally verified compiler back-end. *J. Autom. Reasoning*, 43(4) :363–446, 2009.

[16] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *ACM Int. Conf. POPL*, 2010.

[17] M. Negri, G. Pelagatti, and L. Sbattella. Formal semantics of SQL queries. *ACM Trans. Database Syst.*, 16(3) :513–534, 1991.

[18] The Agda Development Team. *The Agda Proof Assistant Reference Manual*, 2010.

[19] The Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2010.

[20] The Isabelle Development Team. *The Isabelle Interactive Theorem Prover*, 2010.

SMTCoq: Automatisation Expressive et Extensible dans Coq

Valentin Blot[1,2,3] *†, Amina Bousalem[1], Quentin Garchery[1,2,4], et Chantal Keller[1]

[1] LRI, Univ. Paris-Sud, CNRS UMR8623, Orsay, Université Paris-Saclay
Prénom.Nom@lri.fr
[2] IRIF, Univ. Paris-Diderot, CNRS UMR8243, Paris
[3] LSV, ENS Paris-Saclay, CNRS UMR8643, Cachan, Université Paris-Saclay
[4] Inria, Palaiseau, Université Paris-Saclay

Résumé

Les assistants de preuve basés sur la Théorie des Types, tels que Coq, permettent l'implantation de tactiques automatiques efficaces reposant sur la réflexion calculatoire (ex : `lia`, `ring`). Malheureusement, celles-ci sont généralement limitées à un domaine mathématique particulier (ex : l'arithmétique linéaire entière, la théorie des anneaux). *A contrario*, SMTCoq est un outil modulaire et extensible, faisant appel à des prouveurs externes, qui généralise ces approches calculatoires pour combiner les raisonnements issus de multiples domaines. Pour cela, il repose sur une interface de haut niveau, qui offre une plus grande expressivité, au prix d'une automatisation plus complexe.

Dans cet article, nous détaillons deux améliorations : la possibilité de faire appel à des lemmes quantifiés, et celle d'utiliser plusieurs représentations d'une même structure de données. Elles permettent de construire une tactique automatique basée sur SMTCoq qui soit expressive sans remettre en cause la modularité ni l'efficacité de ce dernier. Une telle tactique permet ainsi une automatisation extensible, à faible coût, à de nouveaux domaines supportés par les prouveurs automatiques de l'état de l'art.

1 Introduction

Les méthodes formelles regroupent un ensemble de techniques et outils permettant de concevoir, développer et vérifier des systèmes informatiques avec un haut niveau de confiance. L'intérêt des méthodes formelles est mis en avant dans le cas des systèmes critiques. Plus généralement, un certain nombre de méthodes nécessitent des outils capables de formuler et de prouver des propriétés mathématiques de manière formelle. Parmi ces outils, on distingue deux grandes familles : les assistants de preuve et les prouveurs automatiques. Les premiers sont expressifs mais nécessitent une intervention importante de l'utilisateur lors de la contruction des preuves. Les prouveurs automatiques, quant à eux, définissent des heuristiques de recherche de preuve ne nécessitant aucune intervention de l'utilisateur, au prix d'une expressivité plus restreinte. SMTCoq (disponible à l'adresse `https://smtcoq.github.io`) fournit un pont entre ces deux familles d'outils en permettant l'appel, au moyen de tactiques dédiées, de prouveurs automatiques au sein de l'assistant de preuve Coq. Pour cela, afin de conserver la cohérence de Coq, les prouveurs automatiques utilisés doivent renvoyer un fichier de certificat justifiant la preuve d'un problème donné.

*This research was supported by the Labex DigiCosme (project ANR11LABEX0045DIGICOSME) operated by ANR as part of the program "Investissements d'Avenir" Idex ParisSaclay (ANR11IDEX000302).

† This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 799557.

Initialement, le développement de SMTCoq visait à permettre la vérification de tels certificats SAT ou SMT importants [1] : un programme écrit en Coq, le vérificateur, rejoue les étapes du certificat et vérifie la correction de chacune d'elles, afin de déterminer la validité du problème d'entrée. Ce programme est certifié : un lemme de correction énonce que si le vérificateur termine avec succès, alors le problème est prouvé. Ainsi, la confiance dans les résultats des prouveurs automatiques est accrue par la vérification supplémentaire effectuée en Coq.

Les développements récents de SMTCoq [10, 11] se sont attachés à améliorer son expressivité et sa facilité d'utilisation. En effet, SMTCoq possède une interface de haut niveau reposant sur une représentation modulaire des certificats. Ainsi il peut facilement être étendu à de nouveaux prouveurs ainsi qu'à de nouvelles théories supportées par ces prouveurs. SMTCoq a ainsi été étendu au prouveur CVC4 [2] et aux théories des vecteurs de bits et des tableaux supportées par celui-ci. Cela requiert l'extension du format de certificats de SMTCoq, la définition du vérificateur sur ces nouvelles règles et la preuve des lemmes de correction correspondant. Cette preuve requiert la définition des théories supportées par le prouveur au sein de Coq, ou l'utilisation de théories déjà existantes. L'expressivité de SMTCoq peut également être améliorée en permettant d'effectuer des conversions entre les diverses implantions en Coq d'une théorie donnée et la théorie dans laquelle a été implanté le lemme de correction. Pour la théorie des formules booléennes, le lemme de correction utilise le type `bool` des booléens. Le développement d'une tactique de conversion entre ce type et le type `Prop` des propositions de Coq [11] a ainsi permis d'utiliser la machinerie de SMTCoq pour prouver des théorèmes formulés dans le type `Prop`.

Dans cet article, nous exploitons ces techniques dans un cadre général pour étendre encore l'expressivité de SMTCoq. Nos contributions sont les suivantes. D'une part, nous montrons comment étendre le vérificateur à une théorie d'une autre nature : un fragment de la logique du premier ordre. Pour cela nous étendons le format de certificats de SMTCoq, le vérificateur, et le lemme de correction, pour permettre l'instanciation de lemmes universellement quantifiés. Nous appliquons cette extension aux certificats générés par le prouveur veriT. D'autre part, nous définissons de nouvelles tactiques de conversion entre différents types d'entiers dans Coq. Ainsi, bien que la théorie arithmétique entière de SMTCoq utilise le type de données `Z` de Coq, il est maintenant également possible d'utiliser des représentations alternatives telles que `N`, `nat` ou `positive`. Ces améliorations font bénéficier les utilisateurs de Coq d'une meilleure automatisation au sein de l'assistant de preuve tout en rendant l'outil plus facile à utiliser.

Cette automatisation peut être appliquée à de nombreux cas du premier ordre. Par exemple, en axiomatisant la théorie des groupes avec un élément neutre à gauche `e` et un inverse à gauche, la tactique `verit` fournie par SMTCoq (faisant appel au prouveur SMT veriT) permet de prouver automatiquement que `e` est neutre à droite, et que l'inverse est un inverse à droite. En axiomatisant également l'itéré de la multiplication `power`, on peut également simplement prouver

```
Goal forall (n:nat), power e n = e.
```

en réalisant une induction sur `n`, puis en appliquant la tactique `verit`.

Dans la section suivante nous donnons un aperçu du fonctionnement général de SMTCoq ainsi que les éléments nécessaires à la présentation de notre travail. Dans la section 3 nous présentons l'extension de SMTCoq pour la gestion de lemmes quantifiés puis, dans la section 4, nous présentons les tactiques de conversion entre représentations d'entiers en Coq. Nous finissons par une discussion sur les travaux connexes avant de conclure et donner nos perspectives de recherche futures.

2 SMTCoq et son environnement

Afin de mieux comprendre comment SMTCoq établit une communication entre l'assistant de preuve Coq et différents prouveurs automatiques, nous commençons par détailler le fonctionnement de ces logiciels formels. On s'intéresse ensuite plus précisément à SMTCoq en donnant sa décomposition en modules de programmation.

2.1 L'assistant de preuve Coq

Les assistants de preuve permettent d'exprimer des théorèmes complexes puis de les vérifier de manière interactive. Un assistant de preuve est en ce sens un outil permettant de vérifier les étapes d'une preuve fournie de manière rigoureuse et exhaustive par l'utilisateur.

L'assistant de preuve Coq s'appuie sur l'isomorphisme de Curry-Howard et implante la vérification de preuves au moyen d'un algorithme de typage pour le calcul des constructions inductives [16]. Celui-ci constitue le *noyau* de Coq, le composant critique sur lequel repose la correction de l'assistant de preuve dans son ensemble. Afin de maximiser la confiance des utilisateurs dans ce noyau, celui-ci est implanté en OCaml, un langage de haut niveau proche de sa logique. L'accent est mis sur la concision et la clarté du code.

Les structures de données en Coq sont définies au moyen de types inductifs. Le type simplifié des formules de SMTCoq est donné par :

```
Inductive formula :=
    | Bool (b : bool)
    | Neg (f : formula)
    | And (f1 : formula) (f2 : formula)
    | ...
.
```

Un élément du type `formula` représente un arbre avec des booléens aux feuilles, les nœuds étant des connecteurs logiques. Dans SMTCoq, en plus des booléens, le type des formules inclut les symboles d'arithmétique linéaire sur les entiers, les symboles de la théorie des tableaux et de la théorie des vecteurs de bits mais ne contient pas de quantificateurs.

L'interprétation est une fonction qui calcule la valeur d'une formule de SMTCoq et est définie par induction :

```
Fixpoint interp t := match t with
    | Bool b => b
    | Neg f => negb (interp f)
    | And f1 f2 => andb (interp f1) (interp f2)
    | ...
end.
```

Les fonctions Coq `negb` et `andb` implantent respectivement la négation booléenne et la conjonction booléenne. Enfin, il est possible de prouver des propriétés reposant sur ces définitions :

```
Lemma andproj1 f1 f2 :
  interp (And f1 f2) = true -> interp f1 = true.
```

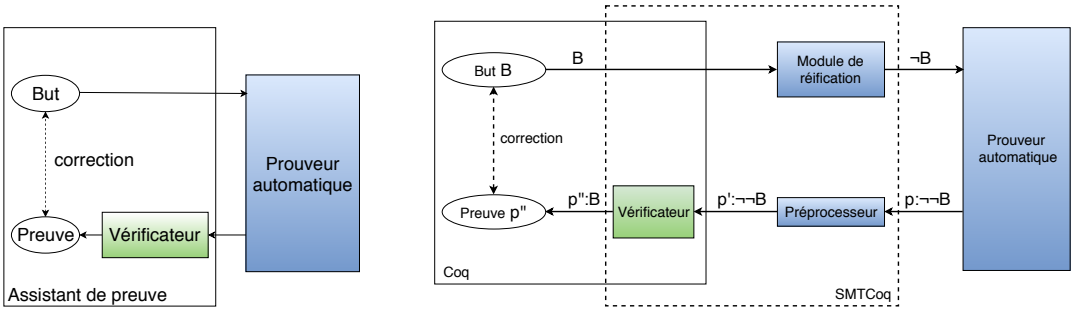


FIGURE 1 – Approche sceptique : cas général (gauche) et application à SMTCoq (droite)

2.2 Les prouveurs automatiques

Les prouveurs automatiques définissent des heuristiques de recherche de preuve et ne reposent donc pas sur l'utilisateur pour la construction de la preuve : l'effort de certification se réduit à la formalisation du problème. En contrepartie, la logique d'un prouveur automatique est plus limitée et/ou la réponse en temps fini n'est pas garantie. L'efficacité de la recherche de preuve étant primordiale, les prouveurs automatiques sont fréquemment écrits dans des langages bas niveau tels que C ou C++, font usage de structures mutables complexes et contiennent généralement de nombreuses optimisations.

Parmi les prouveurs automatiques, les prouveurs SAT (*satisfiabilité* propositionnelle) et SMT (*satisfiabilité modulo théories*) prennent en entrée des formules contenant des variables et tentent de résoudre le problème de satisfiabilité de ces formules : il s'agit de savoir s'il existe une affectation des variables rendant toutes les formules vraies. Dans le cas où une telle affectation existe, la réponse consiste en l'affectation en question. Dans le cas contraire, certains prouveurs fournissent une preuve qu'aucune affectation des variables ne rend les formules vraies. Cette preuve se présente sous la forme d'un fichier de certificat dont le format peut varier en fonction du prouveur automatique considéré.

2.3 Approche sceptique

L'utilisation de prouveurs automatiques au sein d'un assistant de preuve peut se faire selon deux méthodes appelées *sceptique* et *autarcique*. Alors que l'approche autarcique repose sur une certification de l'ensemble du code du prouveur automatique, l'approche sceptique se contente d'obtenir et vérifier un certificat de la part du prouveur automatique.

L'approche sceptique, utilisée par SMTCoq, nécessite seulement la certification d'un vérificateur de certificats plutôt que de l'ensemble du prouveur automatique, comme représenté sur la FIGURE 1 (gauche). À chaque appel du prouveur automatique, le certificat éventuellement généré par celui-ci est vérifié puis transformé en une preuve du but initial. Cette approche ne permet pas de garantir la complétude du système : certains buts valides ne sont pas démontrés, notamment si le prouveur automatique renvoie un certificat erroné ou ne renvoie pas de certificat. En revanche, le développement du prouveur automatique reste indépendant de son utilisation dans l'assistant de preuve puisque seuls les certificats qu'il produit sont vérifiés.

2.4 Fonctionnement de SMTCoq

L'utilisation de l'approche sceptique permet à SMTCoq d'être modulaire vis-à-vis des prouveurs automatiques : SMTCoq supporte aujourd'hui les prouveurs zChaff, veriT et CVC4 et peut être étendu à d'autres prouveurs simplement en écrivant l'analyseur syntaxique pour le format des certificats fournis par celui-ci. Ainsi, la version actuelle de SMTCoq fournit les tactiques `zchaff`, `verit` et `cvc4` qui permettent de faire appel au prouveur correspondant pour résoudre le but courant et ainsi profiter de l'automatisation du prouveur au sein de l'assistant de preuve.

Décrivons maintenant les différentes étapes effectuées par SMTCoq lorsque l'utilisateur lance l'une des tactiques fournies et que le but courant est, par exemple, $\forall x\, B(x)$ (où B est une formule sans quantificateur). SMTCoq envoie tout d'abord la négation de la proposition B au prouveur automatique. Si ce dernier renvoie un certificat indiquant la non-satisfiabilité du problème, on en déduit une preuve du lemme. En effet, puisque les prouveurs automatiques résolvent des problèmes de satisfiabilité (quantificateurs existentiels), pour prouver un lemme en forme prénexe (quantificateurs universels) tel que $\forall x\, B(x)$, on se ramène à $\neg(\exists x\, \neg B(x))$. Ces deux formules sont équivalentes dans notre cas car nous requérons que tous les prédicats apparaissant dans B soient décidables[1].

La construction d'une preuve du but initial par SMTCoq passe par plusieurs étapes (Figure 1 droite). L'énoncé du lemme est d'abord traduit par le module de réification (§ 2.6) dans un type de données OCaml. Le terme obtenu peut alors être écrit dans un fichier servant d'entrée au prouveur automatique. En cas de succès de ce prouveur, on obtient un fichier de certificat qui est ensuite traduit par le préprocesseur (§ 2.7) dans un format adapté au vérificateur de SMTCoq. Ce dernier rejoue la preuve en Coq (§ 2.5).

2.5 Vérificateur certifié

Le vérificateur est une fonction `checker` (§ 2.5.1) écrite et certifiée en Coq qui reproduit le fonctionnement des certificats des prouveurs automatiques. La preuve du but initial repose sur le théorème de correction de cette fonction (§ 2.5.2).

2.5.1 La fonction `checker`

La fonction `checker` a pour type `formula -> list rule -> bool` où le premier argument de cette fonction est la formule d'entrée du problème. Le deuxième argument est un certificat formalisé en Coq, défini comme une liste de règles. La valeur de retour de `checker` indique s'il est possible d'obtenir une contradiction à partir de la formule et du certificat comme précisé dans le paragraphe suivant. Le certificat Coq est obtenu à partir d'un fichier de certificat fourni par un prouveur automatique, chaque règle de ce dernier étant traduite en une règle Coq, dont le type est donné par :

```
Inductive rule :=
    | AndProj (pos_prem : int) (ind_proj : int)
    | Resolution (pos_param : int list)
    | ...
.
```

Afin de calculer sa valeur de retour, la fonction `checker` maintient un ensemble de formules appelé état (que nous représentons ici, pour plus de clarté, à l'aide d'une liste) qui, initialement,

1. Nous n'ajoutons donc pas d'axiome classique en Coq, $\neg(\exists x\, \neg B(x)) \Rightarrow \forall x\, B(x)$ étant alors prouvable.

contient la formule d'entrée. Prenons pour exemple le problème ayant pour formule d'entrée `in` définie par

```
in := And (Bool x) (Neg (Bool x))
```

où `x` est un booléen. L'état est alors initialisé à `[in]`. Pour chaque règle du certificat Coq, la fonction `checker` enrichit l'état d'une nouvelle formule définie par la règle en question.

En appelant un prouveur automatique sur l'exemple, on obtient un fichier de certificat que l'on peut traduire en un certificat Coq :

```
c := [AndProj 1 1; AndProj 1 2; Resolution [3; 2]]
```

Dans le cas de la règle `AndProj`, le paramètre `pos_prem` indique la position dans l'état de la prémisse. La règle est correctement appliquée si cette formule est de la forme `And t1 t2`. Dans ce cas, l'indice de projection `ind_proj` est utilisé pour savoir quelle projection appliquer pour obtenir la formule à rajouter à l'état. La première règle du certificat donné en exemple a pour prémisse la formule en première position de l'état, c'est-à-dire `And (Bool x) (Neg (Bool x))`. En faisant la projection sur la première composante de la conjonction, on sait que `checker` ajoute la formule `Bool x` à l'état. Après cette règle, l'état est donc `[in; Bool x]`. De même, la deuxième règle ajoute `Neg (Bool x)` à l'état.

La règle de résolution prend une liste de paramètres en argument qui représente les emplacements de l'état à prendre en compte. Retenons simplement que si les formules présentes à ces positions contiennent une formule et sa négation alors `checker` ajoute `Bool false` à l'état. Ainsi, après la dernière règle du certificat, l'état devient `[in; Bool x; Neg (Bool x); Bool false]`.

Enfin, la valeur de retour de `checker` est un booléen qui indique si, après avoir pris en compte le certificat, l'état contient la formule `Bool false`. C'est le cas de notre exemple, on a donc `checker in c = true`.

2.5.2 Le théorème de correction

Pour obtenir une preuve du but initial, on applique le théorème de correction qui, pour une formule `in` et un certificat `c`, nous donne :

```
checker in c = true -> interp in = false
```

On notera que ce théorème donne bien la négation de la formule d'entrée (§ 2.4).

La preuve de ce théorème repose sur le *lemme de correction d'une règle* qui nous assure que `checker` ajoute une nouvelle formule valide dans l'état courant (on dit qu'une formule est valide lorsque son interprétation est vraie). Ainsi, lorsque l'on veut modifier une règle ou rajouter un nouveau type de règle, il suffit de modifier ou de compléter la preuve de ce lemme. Ce fonctionnement facilite le développement incrémental de SMTCoq, qui va nous permettre d'ajouter le support pour des lemmes quantifiés (section 3).

Donnons les grandes étapes de la preuve du théorème de correction. Étant donnés un certificat `c` et une formule `in`, on suppose à la fois que `checker in c = true` et que `in` est valide et on veut aboutir à une contradiction. Puisque l'état est initialisé avec la formule d'entrée `in`, toutes les formules de l'état sont valides. On sait, grâce au lemme de correction d'une règle, que cette propriété est conservée par `checker`. Comme on a par ailleurs `checker in c = true`, c'est-à-dire que l'état final contient la formule `Bool false`, l'interprétation de cette formule fournit la contradiction voulue.

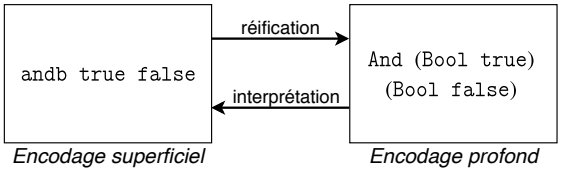
La preuve de l'hypothèse `checker in c = true` est obtenue par calcul de la fonction `checker`. La preuve du but initial repose donc sur un calcul, c'est ce qu'on appelle la réflexion calculatoire.

2.6 Réification

La réification est le fait de passer d'un encodage superficiel (*shallow-embedding*) à un encodage profond (*deep-embedding*). Dans le cas d'un encodage profond, un terme est un élément d'un type de données (comme le type `formula`) ce qui met en évidence sa structure. À l'inverse, l'encodage superficiel du même terme est donné directement par sa valeur dans le langage (ici, Coq).

Donnons l'exemple de la formule $true \ \& \ false$ où & désigne la conjonction booléenne. On peut donner son encodage superficiel à l'aide de la fonction Coq `andb` qui implante la conjonction booléenne : `andb true false`. L'encodage profond de la formule peut être donné dans le type inductif `formula` : `And (Bool true) (Bool false)`.

Pour pouvoir utiliser la fonction de réification, il faut que son résultat soit lié au terme initial : c'est le rôle de la fonction d'interprétation (§ 2.1). Cette dernière doit inverser la réification de sorte que, pour un terme Coq `t`, on ait `t = interp u` où `u` est la réification de `t`.



La réification, dans le cas de SMTCoq, sert à mettre en évidence la structure des formules afin de pouvoir les écrire dans un fichier au format reconnu par un des prouveurs automatiques disponibles et de permettre à la fonction `checker` de les manipuler.

2.7 Préprocesseur

Les certificats passent par une étape intermédiaire assurée par le préprocesseur avant d'être interprétés en des certificats SMTCoq. Cette étape présente plusieurs avantages.

Premièrement, le préprocesseur permet d'assurer la modularité de SMTCoq vis-à-vis des prouveurs automatiques. Bien que les certificats puissent avoir différentes formes en fonction du prouveur automatique dont ils proviennent, il sont tous traduits dans le format commun du paragraphe 2.5.1. Actuellement, le préprocesseur prend en charge trois prouveurs automatiques, à savoir : zChaff, veriT et CVC4.

Une fois cette traduction dans un format commun faite, le préprocesseur optimise le certificat obtenu. Par exemple, afin d'avoir un certificat de petite taille, le préprocesseur élimine les règles redondantes. Il calcule également l'allocation des formules dans l'état du vérificateur (un emplacement contenant une formule qui n'est plus utile peut être réutilisé).

Enfin, il n'est pas nécessaire de prouver la correction du préprocesseur. En effet, on utilise l'approche sceptique qui consiste à vérifier que le certificat fourni est correct et non à verifier que le procédé de génération du certificat est correct.

3 Ajout de lemmes quantifiés

3.1 Motivations

Initialement, SMTCoq permettait uniquement la démonstration automatique de théorèmes en forme prénexe : il n'était pas possible de faire appel à un théorème quantifié déjà démontré.

Par exemple, en supposant que l'on a réussi à montrer les deux lemmes suivants :

$$\forall h.homme(h) \Rightarrow mortel(h) \qquad\qquad homme(Socrate)$$

alors on ne peut pas montrer automatiquement (en utilisant SMTCoq) que :

$$mortel(Socrate)$$

Cette partie détaille une amélioration de la modularité de SMTCoq qui consiste à permettre la transmission de lemmes (potentiellement) quantifiés au prouveur automatique.

3.2 Ajout de lemmes au contexte

L'ajout de lemmes quantifiés au contexte demande d'étendre les étapes de construction d'une preuve Coq à partir du but initial (Figure 2).

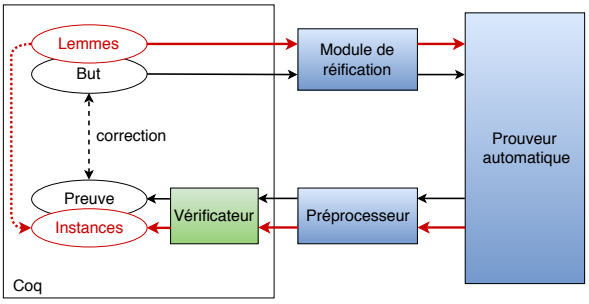


Figure 2 – Ajout de lemmes

En plus du but initial, le module de réification envoie au prouveur automatique les lemmes que l'utilisateur a rajoutés au contexte. Le préprocesseur traduit la réponse du prouveur en un certificat dont le format aura été étendu pour prendre en compte les instanciations de lemmes (§ 3.4.2). Le vérificateur en déduit une preuve qui dépend d'instances de ces lemmes (§ 3.3). Enfin, il faut trouver des preuves de ces instances à partir des lemmes (§ 3.4.3).

Cet ajout a l'avantage de ne pas étendre les formules de SMTCoq avec des quantificateurs, et ne demande donc que de légères modifications au niveau du vérificateur.

3.3 Vérificateur pour l'ajout de lemmes quantifiés

On se propose ici de compléter le type `rule` en laissant inchangé le type des formules. Cette modification vise à prendre en charge les lemmes en forme prénexe ajoutés au contexte par l'utilisateur. Le cas général où les quantificateurs peuvent être dans n'importe quelle position dans la formule ne peut pas être traité avec cette approche : il faudrait étendre le type `formula` avec des quantificateurs, ce qui nous obligerait à étendre la fonction d'interprétation et à adapter les preuves la concernant.

Le nouveau constructeur du type `rule` s'écrit :

```
ForallInst (lemma : Prop) (plemma : lemma)
           (inst : formula) (pinst : lemma -> interp inst = true)
```

Lorsque `checker` rencontre cette règle, l'instance `inst` est ajoutée à l'état. Les autres champs de cette règle sont nécessaires pour la preuve du lemme de correction d'une règle. Ainsi `lemma` devra être un lemme en rapport avec cette instance :

— ce lemme doit être prouvé, la preuve est le champ `plemma`

— ce lemme doit nous assurer que l'instance est valide, c'est le champ `pinst`

En effet, on peut prouver que `inst` est valide, et même indépendamment des formules contenues dans l'état. Il suffit pour cela d'appliquer `pinst` à `plemma`. On a donc rétabli la preuve du théorème de correction.

Cette règle permet de traiter les instanciations des prouveurs automatiques mais demande, pour une instance donnée, d'identifier un lemme, sa preuve et de donner une preuve d'instanciation (champ `pinst`). Cet aspect de la construction d'une règle `ForallInst` est spécifique au prouveur automatique considéré, le cas d'étude présenté résout le cas de veriT (§ 3.4.3). La preuve d'instanciation est donnée à l'aide d'une coupure (tactique `assert` en Coq), ce qui nous permet de l'expliciter dans un deuxième temps.

La règle `ForallInst` diffère des règles utilisées dans SMTCoq jusqu'ici : elle utilise l'encodage superficiel pour le lemme Coq et l'encodage profond pour l'instance. D'un côté, l'utilisation de lemmes Coq déjà existants ne nécessite pas de les réifier, ce qui justifie l'utilisation d'un encodage superficiel. D'un autre côté, l'encodage profond des formules est déjà implanté pour les termes en provenance du fichier de certificat.

3.4 Cas d'étude : application au prouveur veriT

Le format commun des termes de SMTCoq permet d'encoder les certificats des différents prouveurs automatiques sous un même type. L'extension du vérificateur qui est proposée ici est donc utilisable pour n'importe quel prouveur automatique à condition d'étendre le préprocesseur pour celui-ci.

Dans cette partie, nous allons voir comment la règle d'instanciation $forall\_inst$ de veriT est encodée dans le format commun. Cet encodage nécessite de transformer le certificat veriT, de retrouver à quel lemme correspond une instance et de montrer que ce lemme implique l'instance.

3.4.1 Instanciation d'un lemme par veriT

Les certificats de veriT sont constitués d'une suite de règles de la forme :

$$id : (typ\ res\ prem)$$

où id est un entier qui identifie la règle, typ est le type de la règle, la formule res est la conclusion de la règle et $prem$ est la liste des identifiants des prémisses de la règle.

Prenons en exemple le fichier de certificat suivant :

$1 : (input\ (\neg((3 + a) * b = 3 * b + a * b))$

$2 : (input\ (\forall x\, y\, z,\ (x + y) * z = x * z + y * z))$

$3 : (forall\_inst\ (\neg(\forall x\, y\, z,\ (x + y) * z = x * z + y * z) \vee (3 + a) * b = 3 * b + a * b))$

$4 : (resolution\ ()\ 3\ 2\ 1)$

Les règles *input* correspondent aux formules données en entrée. La règle 4 est une règle de résolution qui a pour prémisses les conclusions des règles 1, 2 et 3 et qui en déduit l'absurde représenté par (). La règle 3 est la règle utilisée par veriT pour instancier un lemme. On remarquera que cette règle n'a pas de prémisse et que l'utilisation de son résultat se fait au moyen de la règle de résolution.

3.4.2 Préprocesseur pour la règle *forall_inst*

Le résultat de la règle *forall_inst* est de la forme $\neg(\forall x, P\ x) \vee (P\ c)$ alors que lorsque `checker` rencontre la règle `ForallInst` c'est seulement l'instance `P c` qui est enregistrée dans l'état. Puisqu'on veut encoder la règle *forall_inst* de veriT par la règle Coq `ForallInst`, il faut modifier la suite du certificat qui dépend de cette règle. La forme de la conclusion de la règle *forall_inst* indique qu'elle ne sera utilisée dans la suite du certificat que dans une ou plusieurs règles de résolution (ce qui se vérifie en pratique). On se ramène donc à modifier seulement les règles de résolution.

Par exemple, le fichier certificat de l'exemple donné dans le paragraphe 3.4.1 est traduit dans un certificat Coq de la forme :

```
[ForallInst _ _ inst _ ; Resolution 4 [1; 3]]
```

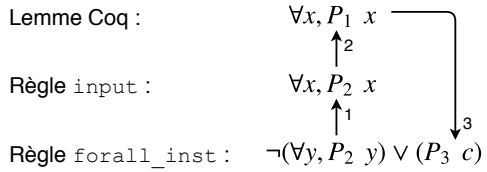
où les `_` sont à compléter (voir paragraphe suivant) et où l'instance `inst` est donnée par :

```
Eq (Mult (Plus 3 a) b) (Plus (Mult 3 b) (Mult a b))
```

On remarque que la règle `Resolution` n'a plus de dépendance à la formule $\forall x\, y\, z,\ (x+y)*z = x*z+y*z$ donnée en entrée.

3.4.3 Construction d'une règle `ForallInst`

La construction d'une règle `ForallInst` demande de reconnaître à quel lemme correspond une instance (champs `lemma` et `plemma`) et de montrer que ce lemme implique bien l'instance (champ `pinst`).



Reconnaissance du lemme correspondant à l'instance. La reconnaissance du lemme passe par une étape intermédiaire : on commence par chercher de quelle règle *input* vient l'instanciation (flèche `1` dans la figure). L'avantage de cette étape est que la formule dans la règle *input* est plus proche du lemme Coq, en particulier les variables liées n'ont pas été renommées.

On cherche ensuite à établir un lien entre une formule du certificat de veriT et un lemme rajouté par l'utilisateur de SMTCoq (flèche `2` dans la figure). La difficulté vient du fait que les lemmes additionnels peuvent apparaître modifiés dans les certificats de veriT. Par exemple, si un lemme Coq a une sous-formule de la forme $a = b$, alors le résultat de la règle *input* correspondant à ce lemme peut avoir à la place $b = a$ comme sous-formule. Rétablir la forme initiale est une solution trop coûteuse car il faudrait modifier la structure de toutes les formules suivantes qui en dépendent. Pour résoudre ce problème, on utilise une table de hachage, initialisée avec toutes les sous-formules des lemmes Coq, qui nous sert pour reconnaître des formules aux modifications de veriT près. Pour traiter l'exemple de la symétrie de l'égalité, à chaque fois que l'on rencontre une sous-formule de la forme $a = b$, en plus de vérifier si cette formule est déjà contenue dans la table, on regarde aussi si la formule $b = a$ est dans cette même table. Ainsi, on peut reconnaître

efficacement des formules identiques modulo symétrie de l'égalité et retrouver à quel lemme se rapporte une formule.

La recherche du lemme correspondant à l'instance à été implantée en OCaml et nous permet de compléter les champs `lemma` et `plemma` lors de la construction d'une règle `ForallInst`.

Preuve automatique d'une instanciation. Pour compléter la construction de la règle `ForallInst`, il faut donner la preuve d'instanciation (flèche `3` dans la figure ci-dessus). On définit donc une tactique Coq qui permet d'automatiser la recherche de ces preuves.

Le lemme Coq suivant correspond directement à l'instanciation du lemme donné :

$$(\forall x, P\ x) \Rightarrow P\ c$$

Dans ce cas simple, la tactique `auto` résout automatiquement le but. Cependant, ce n'est plus le cas lorsque l'instance est légèrement modifiée. Par exemple, le lemme suivant ne peut pas être prouvé directement par la tactique `auto` :

$$(\forall x, f\ x = f\ c) \Rightarrow f\ c = f\ 3$$

Il faut donc remettre le but dans une forme qui correspond à celle du lemme. Dans le cas ci-dessus il faut utiliser la symétrie de l'égalité avant d'appliquer la tactique `auto`.

Deux sources de modifications sont à prendre en compte ici (voir figure). D'une part le lemme en entrée dans le fichier de certificat peut différer du lemme Coq initial (on peut avoir $P_1 \neq P_2$ comme expliqué précédemment). D'autre part, certaines modifications de veriT sont propres à la règle d'instanciation (on peut aussi avoir $P_2 \neq P_3$).

La solution proposée se présente sous la forme d'une tactique Coq et prend en compte toutes ces modifications afin de trouver automatiquement les preuves d'instanciation.

3.5 Résultats et perspectives

Nous avons présenté une extension de la tactique `verit` à l'utilisation de lemmes déjà démontrés. Ceux-ci sont ajoutés au contexte au moyen d'une instruction dédiée `Add_lemmas` ou passés en paramètre à la tactique. Lorsqu'aucun lemme n'est ajouté, son comportement reste le même que précédemment. Dans le cas où des lemmes sont ajoutés au contexte et que le prouveur automatique instancie ces lemmes, les instanciations sont automatiquement prouvées comme décrit ci-dessus.

L'ajout de lemmes s'est traduit par une extension de la logique utilisée et se retrouve dans toutes les étapes intermédiaires de SMTCoq, étapes qu'il a fallu étendre en conséquence. Cette extension a nécessité un développement original : la technique d'encodage des instanciations des lemmes utilisée simplifie l'extension du vérificateur (§ 3.3). Enfin, le procédé de vérification final a été automatisé (§ 3.4.3) afin de préserver la facilité d'utilisation de SMTCoq. Grâce à la modularité de SMTCoq, appliquer l'approche à d'autres prouveurs (capable d'instancier des lemmes et de fournir les instances) consiste simplement à compléter le pré-processeur comme cela a été fait pour `verit`, sans avoir besoin de modifier le code Coq.

Pour valider notre approche, nous avons automatiquement prouvé des propriétés plus complexes grâce à SMTCoq, portant sur : la théorie des groupes, une théorie formalisant les listes d'entiers, des fonctions définies récursivement, etc.

Une extension serait de pouvoir ajouter au contexte des lemmes qui ne sont pas nécessairement en forme prénexe. Cette extension de la logique de SMTCoq ne peut pas être réalisée directement avec la méthode proposée ici, qui requiert que les instances des lemmes soient sans quantificateurs, et est donc laissée pour une perspective à plus long terme.

Afin que l'utilisateur n'ait pas à fournir les lemmes à utiliser, tout en évitant de surcharger le prouveur avec tous les lemmes présents dans le contexte, on pourrait aussi utiliser des méthodes de *machine learning* pour sélectionner automatiquement un sous-ensemble des lemmes, comme c'est fait dans CoqHammer et SledgeHammer [4, 9].

4 Traductions entre représentations des données

4.1 Motivations

En Coq, comme dans la plupart des assistants de preuve, plusieurs représentations cohabitent pour une même structure de données. Le problème est bien connu des utilisateurs de Coq pour la représentation des entiers : les bibliothèques usuelles définissent notamment les entiers naturels unaires (type `nat`) et binaires (type `N`), les entiers strictement positifs binaires (type `positive`), les entiers relatifs binaires (type `Z`), les grands entiers en base 2^{31} (types `bigN` et `bigZ`), ...etc. Ce problème se pose pour la plupart des types de données : par exemple, les vecteurs de bits peuvent être représentés avec des types dépendants de la longueur ou non.

Pour utiliser la pleine potentialité des prouveurs sous-jacents, SMTCoq doit injecter toutes les représentations d'un même type de données dans le type SMT correspondant. Plusieurs approches ont été proposées pour effectuer automatiquement de telles conversions (voir la section 5 pour une présentation détaillée), mais un problème nouveau se pose dans le cas de SMT : la conversion doit pouvoir être effectuée dans des termes combinant plusieurs théories, et non uniquement la théorie correspondant à la structure de données considérée.

Par exemple, le but suivant, mêlant la théorie de l'arithmétique (sur les entiers strictement positifs) avec un symbole de fonction non interprété `f` :

```
x, y : positive                          f : positive -> positive
============================================================
((x + 3) =? y) -> ((3 <? y) && ((f (x + 3)) <=? (f y)))
```

doit être converti [2] en un but similaire sur les entiers relatifs :

```
x', y' : Z          Hx' : 0 <? x'       Hf'x' : 0 <? f' (x' + 3)
f' : Z -> Z         Hy' : 0 <? y'       Hf'y' : 0 <? f' y'
============================================================
((x' + 3) =? y') -> ((3 <? y') && (f' (x' + 3) <=? f' y'))))
```

où, cette fois, les symboles `+`, `=?`, `<=?` et `<?` et la constante `3` sont ceux du type `Z`, type que la plupart des prouveurs SMT ainsi que SMTCoq supportent déjà [1].

Dans cette partie, nous détaillons une nouvelle approche tenant compte de cette particularité, et là encore peu intrusive vis-à-vis de SMTCoq. Pour des raisons de clarté, nous la présentons ici sur le cas de l'injection du type `positive` dans le type `Z`, mais l'implantation est générique pour toute injection dans `Z` (réalisée à l'aide d'un foncteur, qui est instancié pour les types `nat`, `N` et `positive`) et peut se généraliser similairement à d'autres structures de données.

4.2 Principe

L'approche retenue consiste à transformer le but avant tout appel aux prouveurs automatiques. Cette transformation est effectuée par une tactique écrite en Ltac, le langage de tactiques

2. Plus généralement, on pourrait souhaiter l'ajout de l'hypothèse `forall z, 0 <? z -> 0 <? f' z`. Nous laissons cela en perspective.

de Coq, ce qui permet de profiter du moteur de réécriture de Coq. Dans l'architecture de SMTCoq (FIGURE 1 droite), cela se positionne donc avant le module de réification. La transformation se déroule en trois étapes :

1. l'application d'une double conversion à un sous-ensemble des objets de type `positive` ;
2. l'application de réécritures permettant de convertir les symboles de la théorie des entiers du type `positive` vers le type `Z` ;
3. le renommage des sous-termes résiduels, avec l'ajout des propriétés liées au fait que l'injection de `positive` dans `Z` n'est pas surjective.

Nous allons détailler le fonctionnement sur l'exemple du but indiqué ci-dessus.

1. Double conversion La première étape applique récursivement une double conversion à un sous-ensemble des sous-termes du but de type `positive`, en vue d'obtenir le but suivant :

```
(Z2Pos (Pos2Z x) + Z2Pos (Pos2Z 3) =? Z2Pos (Pos2Z y)) ->
  ((Z2Pos (Pos2Z 3) <? Z2Pos (Pos2Z y)) &&
   (Z2Pos (Pos2Z (f (Z2Pos (Pos2Z x) + Z2Pos (Pos2Z 3))))
                      <=? Z2Pos (Pos2Z (f (Z2Pos (Pos2Z y)))))).
```

L'injection `Pos2Z : positive -> Z` injecte simplement les entiers strictement positifs dans les entiers relatifs. Son inverse partiel `Z2Pos : Z -> positive` injecte les entiers relatifs strictement positifs dans le type `positive`, et renvoie une valeur quelconque dans les autres cas. En effet, ces cas ne sont pas utilisés car la propriété essentielle est que la double conversion de type `positive -> positive` soit l'identité :

```
Lemma Pos2Z_id : forall (p : positive), p = Z2Pos (Pos2Z p).
```

Afin de pouvoir appliquer cette réécriture, cette étape recherche, de manière répétée, tous les contextes du but contenant un terme de type `positive`, et applique la double conversion à chaque sous-terme voulu :

```
repeat
  match goal with
  | |- appcontext C[?p]  =>
    lazymatch type of p with
    | positive => if forme_voulue C p
                  then rewrite (Pos2Z_id p) else fail
    end
  end
```

où `lazymatch` est une variante de `match` permettant ici plus d'efficacité.

L'originalité de cette tactique réside dans la manière dont la tactique teste si le contexte `C` et le terme `p` sont de la forme voulue (noté abusivement `forme_voulue C p` ci-dessus). En effet, on ne souhaite pas appliquer la double conversion si :

1. le symbole de tête de `p` est un symbole arithmétique (car celui-ci sera transformé lors de la deuxième étape), ou l'injection `Z2Pos` (pour que la procédure termine) ; ou
2. le terme n'est pas déjà dans un contexte de la forme `Z2Pos (Pos2Z _)` (même raison).

Pour le cas 1, il suffit de regarder le symbole de tête de `p`. Pour le cas 2, il faut regarder si le contexte `C[]` est de la forme `C'[Pos2Z (ZPos [])]`. Pour cela, on introduit une variable fraîche `var` qui sert à inspecter le contexte sans interférer avec `p` :

```
lazymatch context C[var] with
| context [Z2Pos (Pos2Z var)] => fail
| _ => idtac
end
```

2. Réécriture et 3. Renommage Une fois le but mis sous cette forme, les deux étapes suivantes sont directes.

La deuxième étape réécrit, de manière répétée, les théorèmes établissant les liens entre les opérateurs et les prédicats arithmétiques sur le type `positive` et leurs contreparties sur le type `Z`, afin d'obtenir le but suivant :

```
((Pos2Z x + 3) =? (Pos2Z y)) -> ((3 <? Pos2Z y) &&
   (Pos2Z (f (Z2Pos (Pos2Z x + 3))) <=? Pos2Z (f (Z2Pos (Pos2Z y)))))
```

Cette étape déplace donc la conversion `Z2Pos` le plus à l'extérieur possible, c'est-à-dire sous les appels de fonctions non interprétées (comme `f` dans l'exemple).

Enfin, la troisième étape renomme les sous-termes résiduels : `Pos2Z x`, `Pos2Z y` et `fun a => Pos2Z (f (Z2Pos a))`, en conservant la propriété que, les variables étant injectées de `positive` dans `Z`, elles sont strictement positives. On obtient ainsi le but voulu.

4.3 Résultats et perspectives

Cette méthode permet de traduire des buts entre deux types de données représentant une même structure ou sous-structure, avec les avantages suivants :

— elle agit en présence de symboles de fonctions ne faisant pas partie de la structure considérée (ceux-ci sont alors convertis en symboles non interprétés) ;
— les types n'ont pas besoin d'être isomorphes : il suffit d'une injection ;
— l'effort de démonstration est assez faible, et porte essentiellement sur les preuves d'équivalence entre les symboles interprétés.

Elle a été appliquée avec succès dans SMTCoq pour traduire la plupart des types d'entiers dans le type `Z`. Une perspective à court terme est de l'appliquer aux autres théories, notamment les vecteurs de bits et les tableaux, pour lesquels plusieurs représentation cohabitent également. Cela nécessite l'abstraction ces théories dans les tactiques de conversion. Une perspective à plus long terme est de s'appuyer sur les mécanismes des classes de type, comme présenté dans [18].

5 Travaux connexes

CoqHammer [9] est un plug-in pour Coq proposant une tactique automatique qui, à partir d'un but, sélectionne un ensemble de lemmes, appelle plusieurs prouveurs externes, et vérifie leurs résultats grâce à un prouveur du premier ordre certifié en Coq. Cette approche a l'avantage d'être robuste et a montré de bons résultats sur la bibliothèque standard de Coq. Cela repose sur une axiomatisation des théories (utilisation de lemmes lors de la vérification) qui est moins efficace que la combinaison de procédures de décision et la réflexion calculatoire proposées par SMTCoq. Par ailleurs, CoqHammer s'applique essentiellement à des buts peu combinatoires, alors que SMTCoq permet également de prouver très efficacement des problèmes demandant un raisonnement combinatoire important [1, 13].

CoqHammer tire ses idées de SledgeHammer [17], un outil similaire pour l'assistant de preuve Isabelle/HOL. La première version de SledgeHammer était basée sur la vérification de traces SMT [5], de manière similaire à SMTCoq.

Une alternative est l'approche autarcique, consistant à prouver la correction d'un prouveur dans son ensemble. Cette approche a été menée avec succès par exemple en Coq avec ergo [14] ou dans les prouveurs de type HOL avec metis [12]. Bien qu'ayant l'avantage de prouver la correction des algorithmes sous-jacents, cette approche fige une implantation qui est ainsi difficilement améliorable. Comme indiqué ci-dessus, de tels prouveurs certifiés sont utilisés au sein de SledgeHammer et CoqHammer.

S. Boulmé et A. Maréchal ont proposé une nouvelle approche, appelée la *certification défensive* [8, 7], pour générer un prouveur correct par construction à partir d'un vérificateur de preuves. Cette approche très prometteuse est en cours d'exploration pour SMTCoq.

Plusieurs tactiques proposent des plongements entre types de données, appliqués notamment aux entiers. Dans Coq, les plus utilisées sont `zify` [15], `ppsimpl` [3] et les *transfer tactics* [18]. À notre connaissance, les tactiques `zify` et `ppsimpl` ne sont pas compatibles avec le fait de combiner des théories ; notamment, elles n'opèrent pas sous les symboles de fonctions non interprétés car venant d'autres théories. Elles ne sont donc pas utilisables dans notre cadre. Les *transfer tactics* pourraient opérer sous les symboles de fonctions, mais au prix d'un effort de démonstration important. Par ailleurs, elles ne s'appliquent qu'à des types isomorphes. Cela nous a conduit au développement de nouvelles tactiques (section 4) qui pourront être utilisées dans d'autres cadres que SMTCoq. L'approche par classes de types proposée dans `ppsimpl` et les *transfer tactics* est néanmoins très intéressante pour l'extensibilité et nous souhaitons la porter à notre cadre.

6 Conclusion et perspectives

La combinaison des deux méthodes présentées dans cet article permet d'étendre SMTCoq avec des tactiques de plus en plus expressives, sans remettre en cause la simplicité et l'efficacité du noyau : il n'a pas besoin de gérer des variables liées ou de multiples représentations des données. Nous pensons que cette approche permet de programmer des tactiques automatiques performantes (en efficacité et en expressivité) avec une interface de haut niveau facilement extensible. Cela fait de SMTCoq un combinateur de tactiques.

En complément de l'implantation de tactiques automatiques, certaines idées présentées dans cet article sont à notre connaissance originales et utilisables dans d'autres contextes. Le fait d'utiliser simultanément des encodages profonds avec des encodages superficiels dans les certificats (§ 3) permet de combiner, dans un même vérificateur certifié, la manipulation de termes par leur structure ou par leur représentation en Coq. Le fait d'appliquer une fonction d'injection aux termes d'un type donné dans tout contexte, grâce à l'introduction d'une variable fraîche (§ 4), permet d'avoir une tactique opérant sous les symboles de fonctions non interprétés.

Les perspectives propres à chaque fonctionnalité ont été présentées tout au long de l'article. De nombreuses extensions peuvent être mises en place pour continuer à améliorer les tactiques automatiques de SMTCoq : l'intégration de nouvelles théories comme les types de données algébriques, les chaînes de caractères ou les ensembles finis ; ou encore l'encodage des aspects d'ordre supérieur. Un de nos objectifs à plus long terme est d'offrir une interface de très haut niveau pour ajouter à Coq de nouvelles procédures de décision dédiées supportées par SMT, se combinant aux procédures de décision existantes de manière agnostique, en se basant par exemple sur les idées de [6].

Remerciements Les auteurs remercient Guillaume Melquiond pour ses conseils sur la définition des tactiques de conversion de la section 4, Claude Marché pour la relecture de cet article, ainsi que les relecteurs pour leurs remarques.

Références

[1] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *Certified Programs and Proofs*, pages 135–150, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[3] Frédéric Besson. ppsimpl : a reflexive Coq tactic for canonising goals. In *CoqPL*, 2017.

[4] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. A Learning-Based Fact Selector for Isabelle/HOL. *J. Autom. Reasoning*, 57(3) :219–244, 2016.

[5] S. Böhme and T. Weber. Fast LCF-Style Proof Reconstruction for Z3. In *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2010.

[6] Maria Paola Bonacina, Stéphane Graham-Lengrand, and Natarajan Shankar. Proofs in conflict-driven theory combination. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 186–200. ACM, 2018.

[7] Sylvain Boulmé. Defensive Certification in Coq with ML Type-Safe Oracles. http://www-verimag.imag.fr/~boulme/dc201604, 2016.

[8] Sylvain Boulmé and Alexandre Maréchal. Toward Certification for Free ! working paper or preprint, July 2017.

[9] Lukasz Czajka and Cezary Kaliszyk. Hammer for Coq : Automation for Dependent Type Theory. *J. Autom. Reasoning*, 61(1-4) :423–453, 2018.

[10] Burak Ekici, Guy Katz, Chantal Keller, Alain Mebsout, Andrew J. Reynolds, and Cesare Tinelli. Extending SMTCoq, a Certified Checker for SMT (Extended Abstract). In *Proceedings First International Workshop on Hammers for Type Theories, HaTT@IJCAR 2016, Coimbra, Portugal, July 1, 2016.*, pages 21–29, 2016.

[11] 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 *Computer Aided Verification - 29th International Conference*, Heidelberg, Germany, July 2017.

[12] J. Hurd. System Description : The Metis Proof Tactic. *Empirically Successful Automated Reasoning in Higher-Order Logic (ESHOL)*, pages 103–104, 2005.

[13] Chantal Keller. *Proof Technology in Mathematics Research and Teaching*, chapter SMTCoq : Mixing automatic and interactive proof technologies. Springer, 2018. À paraître.

[14] Stéphane Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq. (Formalisation et developpement d'une tactique reflexive pour la demonstration automatique en coq)*. PhD thesis, University of Paris-Sud, Orsay, France, 2011.

[15] Pierre Letouzey. The zify tactic. https://coq.inria.fr/library/Coq.omega.PreOmega.html.

[16] Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. In *All about Proofs, Proofs for All*, volume 55 of *Studies in Logic (Mathematical logic and foundations)*. College Publications, January 2015.

[17] Lawrence C. Paulson and Jasmin Christian Blanchette. Three years of experience with Sledgehammer, a Practical Link Between Automatic and Interactive Theorem Provers. In *The 8th International Workshop on the Implementation of Logics, IWIL 2010, Yogyakarta, Indonesia, October 9, 2011*, volume 2 of *EPiC Series in Computing*, pages 1–11. EasyChair, 2010.

[18] Théo Zimmermann and Hugo Herbelin. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. *CoRR*, abs/1505.05028, 2015.

CAMLroot: Revisiting the OCaml FFI

Frédéric Bour

Abstract

The OCaml language comes with a facility for interfacing with C code – the Foreign Function Interface or FFI. The primitives for working with the OCaml runtime – and, in particular, with the garbage collector (GC) – strive for a minimal overhead: they avoid unnecessary work and allow for calls to C code to be very cheap. But they are also hard to use properly. Satisfying the GC invariants leads to counter-intuitive C code and there are hardly any safety checks to warn the developer.

In this work, we explore two complementary approaches to mitigate these issues. First, simply adding an indirection to the API manipulating OCaml values lets us write safer code amenable to optional runtime tests that assert proper use of the API. Second, a notion of region for tracking lifetimes of OCaml values on the C side lets us trade some performance for simpler code.

1 Introduction

Writing code to bridge C libraries to OCaml code is a difficult task. While writing Cuite[1], an OCaml library that interfaces the Qt tool-kit, we discovered a few idioms that help to keep this plumbing simple to reason about.

Qt is a C++ framework that enables writing portable user interfaces. User interfaces are challenging to write because they involve complex lifetimes and control flow: data is described as a dynamically changing graph of components, control can jump back-and-forth between user code and library code, different tasks can run concurrently, etc.

Interfacing with OCaml means exporting all these features while abiding by OCaml & Qt rules about memory management. By revisiting a few assumptions of the OCaml GC interface, we believe that the CAMLroot approach makes interface work more tractable and easier to debug.

1.1 Our approach: roots before values

The less natural part of OCaml interaction from C code, in our opinion, is the management of roots and the lifetime of OCaml values. The C programmer is accustomed to manual memory management: by explicitly creating and destroying pieces of memory or by tying the variable lifetime to the scope. While syntactically the management of OCaml memory seems to fall into these cases, it is actually much more subtle.

If the OCaml garbage collector triggers at the wrong time, (1) a value can be moved, (2) a piece of OCaml memory that is locally referenced but not registered as a root can be collected.

If (1) happens in the middle of a sequence point where the same value has been read, this results in an undefined behavior of the C language. Referencing a variable is an effectful operation: the value is copied and its lifetime is disconnected from the lifetime of the variable that holds it: referencing a variable is an effectful operation. This behavior completely contradicts the intuition of the C developer who is not used to distinguishing a variable from its contents. Fortunately that can be addressed by a slight, almost mechanical, change to the C API of the

[1] https://github.com/let-def/cuite

garbage collector. By making most FFI functions take roots (in the form of pointer to values) rather than direct values as arguments, this class of error can be ruled-out.

We also propose an alternative root management strategy that results in simpler code. Both changes are amenable to dynamic checks that can detect incorrect uses.

1.2 Contributions

We claim the following three contributions:

- a general design principle for OCaml FFI functions, working with value pointers rather than plain values, that prevents a class of FFI bugs and integrates well with existing FFI code
- an alternative interface for managing local roots that trades some performance for ease of use and safety by batching roots in *regions*
- camlroot[2] a reusable and open-source library that implements both of those via the `mlroot.h` and `mlregion.h` files.

While these were developed in the context of Cuite, the Qt binding, this paper solely focuses on the management of OCaml memory from C code. However to illustrate its applicability to realistic code, we describe how our proposed API behaves in complex situations – involving callbacks, exceptions and multi-threaded code.

2 The original OCaml-C FFI

In this section we describe the existing solution for writing bindings to external library. The OCaml FFI lets the developer manipulate OCaml values from another programming language. A C library provided by the OCaml distribution exposes the primitive operations to achieve that (Leroy et al.).

This library helps accomplish two main tasks:

1. constructing and deconstructing OCaml values, interpreting them in a meaningful way from the C language (for instance by mapping back-and-forth between OCaml and C representations of integers, of strings, etc);
2. cooperating with the OCaml garbage collector, or GC, the runtime service that takes care of managing OCaml memory.

Even though both tasks are much more difficult than when working from within OCaml code (the typechecker will not provide help in foreign code), it is at least possible to reason locally, in a compositional way, about OCaml values – task (1). For instance, building nested tuples just involves repeatedly building flat tuples. The same cannot be said about task (2). The GC needs to know about all OCaml values that are manipulated from C code, and can look at them at almost any moment. These restrictions are not natural while programming in C and can lead to subtle bugs that are hard to discover.

Most of the time the GC will not do any work, preferring to wait for a batch of work that is big enough to amortize its overhead. As such an improper use of the OCaml API can go unnoticed for a long time. But even once harm has been done, it might just lead to a corruption

[2] https://github.com/let-def/camlroot

of the OCaml heap that affects an unrelated piece of code and fails much later in the program. GC bugs combine two nasty properties: they cannot be studied in isolation and they trigger depending on a complex set of conditions that cannot be inferred by just looking at the buggy code.

On the other hand, the OCaml FFI API enjoys a remarkably low overhead: the restrictions are difficult to adhere to but lead to a cheap and portable interface with the OCaml runtime. This makes OCaml applicable to domains where connecting to a foreign programming language is generally considered too expensive (Bourke et al., 2016).

We propose to explore a different trade-off in the design space of FFI API: providing a safer and more convenient API by giving up some of the performance. Many mainstream languages have adopted heavier FFIs by default (Lua, Java JNI, Go), optionally allowing to resort to a lower-level one for performance critical code (ctypes from LuaJit); thus, a relatively expensive FFI can still be relevant.

But before trying to build an alternative interface to the FFI, let us take a closer look at the restrictions imposed by the GC.

2.1 Value representation

In C code, all OCaml values are represented by the `value` type. It is a signed integer of the same size as a pointer of the host system (in practice, 32 or 64 bits). Values of this size are called "words".

The least significant bit is reserved to help the GC traverse the OCaml heap:

- if it is set, the value is said to be immediate and the remaining bits are directly interpreted: as a 31-bit or 63-bit integer for the `int` OCaml type, or as a unique pattern of bits for constant variants and polymorphic variant constructors

- if it is not set, the value is interpreted as a pointer to a "block"; it is a piece of memory provided by the runtime that is guaranteed to be aligned on a word boundary

Blocks are preceded by a header containing their "tag" and their size. The tag determines how to interpret the contents of the block. For common OCaml values, such as algebraic data, records, tuples or arrays, blocks are made of other values.

2.2 Traversal and compaction

Under certain conditions, the OCaml GC might need to traverse the heap. The basic operation is to find which blocks are reachable from a value.

Depending on the tag, the OCaml GC decides whether a block is made of other `value`s (which, in turn, can be immediate or pointer to blocks) or just of an opaque chunk of memory that does not need to be scanned.

By repeating this operation, the GC can traverse the whole heap. The traversal starts from the roots, a distinguished set of values.

If necessary the GC might also decide to move some blocks. Moving blocks is more demanding than mere traversal: the GC not only needs to know all values referenced from C code, it also needs to be able to update them. The C compiler needs to be aware that all OCaml values have to be reloaded when such an operation happens, as previous values might have been invalidated.

2.3 The memory management macros

A few C macros are provided by the OCaml runtime to implement foreign features. As a guiding example, here is a simple function that takes two values and builds a pair out of them:

```
(* The OCaml version *)
let mk_pair_ocaml x y = (x,y)
(* An external function, implemented in C *)
external mk_pair_c : 'a -> 'b -> 'a * 'b = "mk_pair_c_impl"
```

The string after the external declaration is the name of the C function that implements the functionality. The corresponding C code looks like this:

```
CAMLprim
value mk_pair_c_impl(value a, value b)
{
  CAMLparam2(a, b);
  CAMLlocal1(pair);
  pair = caml_alloc(2, 0);
  Store_field(pair, 0, a);
  Store_field(pair, 1, b);
  CAMLreturn(pair);
}
```

The first macro `CAMLprim` ensures that the symbol is visible from OCaml code.

CAMLparam The `CAMLparam2(a,b)` call expands to two other macros:

- `CAMLparam0()` saves the previous set of local roots
- `CAMLxparam2(a,b)` setups a new block of roots with the addresses of `a` and `b`.

The local roots are in a linked list of pointers to OCaml values, implemented by the `struct caml__roots_block` type, and stored in the `caml_local_roots` global variable.

The job of the memory management macros is to make it as easy as possible to register all local variables of type `value` in this linked list and to remove them when returning from the function.

There should be only one `CAMLparam0()` in a function, but there can be as many calls to `CAMLxparam` as needed. The variants from `CAML(x)param1` to `CAML(x)param5` are available as well as `CAML(x)paramN(array, array_size)` for registering array of values.

CAMLlocal The next macro call of interest is `CAMLlocal1(pair)` that expands to `value pair = Val_unit; CAMLxparam1(pair)`:

- it declares and initializes a local variable named `pair`,
- it adds its address to the set of local roots.

The next lines, the calls to `caml_alloc` and `Store_field`, are not directly related to the management of roots. They deal with the construction of OCaml values – assuming that all variables have been registered properly.

```
// Allocating values
value caml_alloc(int size, int tag);
value caml_copy_string(const char *string);
...

// Deconstructing and mutating values (actually implemented by
   macros)
long Long_val(value v);
value Val_long(long x);
value Field(value v, int offset);
void Store_field(value v, int offset, value x);
...
```

Figure 1: Some OCaml FFI functions for manipulating values

CAMLreturn This last macro restores the previous set of local roots. It sets the variable `caml_local_roots` to the state that was saved by `CAMLparam0()`.

The code above desugars to the following equivalent code:

```
CAMLprim
value mk_pair_c_impl(value a, value b)
{
  // CAMLparam2(a, b);
  CAMLparam0();         // 1) save the state of local roots
  CAMLxparam2(a, b);    // 2a) add &a and &b to local roots

  // CAMLlocal1(pair);
  value pair = Val_unit;
  CAMLxparam1(pair);    // 2b) add &pair to local roots

  ...

  // CAMLreturn(pair);
  CAMLdrop;             // 3) restore the state of local roots
                        //    (forgetting &a, &b and &pair)
  return(pair);
}
```

The three fundamental operations of root managements are saving local roots, registering new ones, and restoring the saved ones when leaving a scope.

The OCaml FFI provides macros to automate most of this work but has no way to enforce their proper use. The functions provided `mlroot.h` can detect large classes of possible misuses while `mlregion.h` introduces an alternative approach to the management of roots.

2.3.1 Carefully dealing with intermediate results

Here is an example that shows how easy it is to misuse this API, taken from (Dolan, 2018, caml-oxide). Lets imagine one wants to make a triplet as two nested pairs:

```
let triplet x y z = (x,(y,z))
```

Armed with `mk_pair_c_impl` and the rules above, one might be tempted to write:

```
CAMLprim
value c_triplet(value x, value y, value z)
{
  CAMLparam3(x,y,z);
  CAMLlocal1(triplet);

  triplet = mk_pair_c_impl(x, mk_pair_c_impl(y, z));

  CAMLreturn(triplet);
}
```

But a bug lies in this implementation: the C compiler might have already loaded the value of `x` (for instance, by copying it on the stack) before the nested call to `mk_pair_c_impl(y, z)`.

If this call triggers a compaction and `x` is moved, the old, and wrong, value of `x` will be passed to the outer call.

The correct version uses an intermediate variable for the temporary value:

```
CAMLprim
value c_triplet(value x, value y, value z)
{
  CAMLparam3(x,y,z);
  CAMLlocal2(intermediate, triplet);

  intermediate = mk_pair_c_impl(y, z);
  triplet = mk_pair_c_impl(x, intermediate);

  CAMLreturn(triplet);
}
```

To avoid bugs, calls to functions manipulating the OCaml memory should be linearized and temporary results should be stored in local roots.

3 mlroot: solving problems with one level of indirection

The first change we propose is to replace the type of arguments of type `value` to the type `value*`, representing roots rather than direct values. Similarly, return values of type `value` are replaced by an extra argument of type `value*` that is used to store the results.

This rewriting is only necessary for functions that allocate, but for the sake of uniformity we offer alternatives in this style for most GC functions. Figure 1 shows some functions from the original OCaml FFI while figure 2 shows the equivalent functions provided by mlroot API. This minor change brings many benefits.

No risk of unexpected copy. A tricky source of bug that we highlighted in the previous section was that OCaml values can be unexpectedly copied in the middle of a call. With the

added indirection only the pointer is copied. If the GC kicks in and rewrites the roots, the pointer will not be invalidated.

Looking at the operations involved in terms of lifetime, reading a value from a root makes it ephemeral: the value is valid only until the next OCaml allocation – or simply undefined if an allocation can happen in the same sequence-point.

When directly working with `value`, this operation is implicit. Working with `value*`, the operation becomes explicit and forces the developer to think about its effects – they choose when to dereference the pointer. In practice this is almost never needed outside the implementation of `mlroot`, relieving the user of the API from this burden.

```
// Allocating values
void mlroot_alloc(value *root, mlsize_t size, tag_t tag);
void mlroot_string_copy(value *root, const char *string);
...

// Deconstructing and mutating values
long mlroot_get_long(value *root);
void mlroot_set_long(value *root, long x);
void mlroot_get_field(value *root, const value *src, int index);
void mlroot_set_field(const value *root, int index, const value *
    src);
...
```

Figure 2: Functions from 1 following mlroot conventions

Preventing nested calls. Now that all functions that interact with the garbage collector take pointers and return void, offending code patterns become much harder. Calls cannot be nested anymore.

Here is what the `triplet` would look like with this approach:

```
static void mk_pair(value *result, value *a, value *b)
{
  mlroot_assert(result != a && result != b);
  mlroot_alloc(result, 2, 0);
  mlroot_set_field(result, 0, a);
  mlroot_set_field(result, 1, b);
}

CAMLprim
value caml_triplet(value x, value y, value z)
{
  CAMLparam3(x, y, z);
  CAMLlocal2(pair, result);
  mk_pair(&pair, &y, &z);
  mk_pair(&result, &x, &pair);
  CAMLreturn(result);
}
```

No need to repeat roots. Callees no longer have the responsibility of registering roots for their arguments.

With the existing OCaml API, any function receiving an argument of type `value` has to register a corresponding root. There are as many roots for the same value as sub-routines calls that received it as argument. With the indirect approach only places that have their address taken need to be registered as root. Since this operation is explicit, we believe the risk of mistake is reduced.

Dereferencing also require some care, but only the implementation of mlroot functions should have to do that and not the user code.

Dealing with immediate values. A reader familiar with OCaml binding code might be worried that working with immediate values (an integer directly stored in a `value`) becomes less convenient with our approach than with the normal API.

Immediate values enjoy a lot of nice properties in the OCaml FFI. Since they do not interact with the memory graph of OCaml – they don't reference blocks, they cannot be moved – the rules for dealing with them are relaxed: they don't have to be put in roots, they can be created without triggering a garbage collection, etc.

We argue these properties should not be exploited. That some values can be represented without interacting with the GC is an implementation detail. Being prepared for the "worst case" allow to present a uniform interface.

3.1 Safety of this indirect API

Moving everything to pointers opens a new opportunity for incorrect uses: aliasing. In the `triplet` case it would mean using the `result` variable both as input and output in the same call:

```
CAMLprim
value caml_triplet(value x, value y, value z)
{
  CAMLparam3(x, y, z);
  CAMLlocal1(result);
  mk_pair(&result, &y, &z);
  mk_pair(&result, &x, &result); // result is aliased!
  CAMLreturn(result);
}
```

While problematic indeed, this case is actually less worrying. The code that dereferences roots can be instrumented to deal with that:

- by properly handling aliasing, for instance by ensuring that all arguments are read before any are written,
- by checking for this case and failing or emitting a warning, as illustrated by the assertions in the implementation of `mk_pair` primitive.

A debugging workflow. Actually thanks to the indirection we can go further than that. The observation is that any well-formed argument of type `value*` should point to a root.

The native OCaml FFI is `value`-centric: functions directly take and produce values. Our rewriting make it `root`-centric: functions receive roots and manipulate values though them.

With native OCaml FFI the connection between a root and its value is lost: when a value argument is passed, a copy is made and it is not possible to know which root it got copied from. However `mlroot` functions can check that these roots have been properly registered by plugging into the GC infrastructure. This comes at a moderate speed cost in the form of a "defensive" mode that can be switched on during development.

Where to add the indirection? Having made explicit the distinction between values (of type `value`) and roots (of type `value*`) in the API, one could wonder why our API makes use of roots in places where values would be fine. The results of `mlroot_get_field` or `mlroot_long_val` for instance. We are not totally decided on this issue and might revisit this design in the future. However the ability to dynamically check for correct use and the more explicit, safer-looking nature of the resulting code makes us favor the root arguments.

Beside the slight increase in verbosity, we did not find any drawback to this approach. The indirection does not increase memory use because the root has to be registered in the GC anyway. The impact on execution time is not significant either: the only change is that the pointer dereferencement (a memory load) is done eagerly with the value-centric API while it is delayed with the root-centric approach.

4 mlregion: dynamic allocation of roots

To further simplify the API described above we propose to make allocation of roots simpler.

The `CAMLparam` and `CAMLlocal` macros declare OCaml roots with a static lifetime, known at compile-time. This is nice for performance but puts more burden on the developer.

The semantics of these macros is hard to understand and some use cases are not easily covered. As we already saw, returning values is tricky, but storing temporary values in code controlled by an external framework is even more problematic.

The need for side-channel allocation. For the sake of the example, let's imagine that we need to sort some C structures containing OCaml values. To achieve this the `qsort_r` function from the C standard library seems appropriate. It takes an array of user-defined structures and a custom comparison operator in the form of a function pointer.

```
struct item {
  my_c_type x;
  value v;
};

static int
c_comparator(const void *item1, const void *item2, void *
    comparator)
{
    value v1 = ((const struct item *)item1)->v;
    value v2 = ((const struct item *)item2)->v;
    value ml_comparator = *(value*)comparator;
    return Val_int(caml_callback2(ml_comparator,v1,v2));
}
```

```
void sort_ocaml_items(struct item *items, size_t count, value *
    comparator)
{
     qsort_r(items, count, sizeof(struct item),
             c_comparator, comparator);
}
```

`caml_callback2` is a primitive function of native OCaml FFI that allows to invoke an OCaml closure from C code[3].

Because of this callback, the garbage collector can be called in the middle of the sorting. Even if we registered roots for all the values in this array, the implementation of `qsort_r` might have made copies that will not be updated by the GC. More generally, rewriting the array in the middle of the sort can lead to unexpected behaviors.

Since we know all the OCaml values that will be reached prior to calling `qsort_r`, a solution is to work with pointer to values. One first allocates an array of roots and passes pointers into this array.

However there exist situations where the set of roots cannot be pre-determined. Regions appeared as a solution to this problem, and proved to be convenient in simpler cases too.

4.1 Region-based management

To let the developer dynamically manage the set of roots, we propose a simple API that over-approximates the lifetime of local roots:

```
typedef struct ... region_t;
void mlregion_enter(region_t *region);
void mlregion_leave(region_t *region);
value *mlregion_new_root(void);
#define CAMLregion(...) ...
#define CAMLregion_return(p) ...
```

In this approach, we distinguish between external and helper functions:

- external functions are the ones that can be directly called from OCaml,
- helper functions implement useful routines for binding foreign code.

The external functions are responsible for setting up the region while helper functions assume that a region has already been set up. Mimicking `CAMLparam...` macros, we provide some helpers for registering parameters while setting up the region:

```
value *pair_helper(value *a, value *b)
{
    value *v = mlregion_new_root();
    mlroot_alloc(v, 2, 0);
    mlroot_set_field(v, 0, a);
    mlroot_set_field(v, 0, b);
    return v;
}
```

[3]For the sake of simplicity we do not deal with the case where the callback raises an exception

```
CAMLprim
value mk_pair(value a, value b)
{
    CAMLregion(&a, &b);
    CAMLregion_return(pair_helper(&a, &b));
}

CAMLprim
value mk_triplet(value x, value y, value z)
{
    CAMLregion(&x, &y, &z);
    CAMLregion_return(pair_helper(&x, pair_helper(&y, &z)));
}
```

Setting up a region introduces a new set of local roots that can grow dynamically as new roots are requested. Leaving a region releases all the roots at once.

Dynamic scoping of regions. A point that might surprise users of this API is that the current region is not explicitly passed to functions, instead it is accessed by some external means.

This design choice was made to simplify integration with Qt code: OCaml code can get called from a method deep in the object hierarchy, whose interface is imposed by the framework. As there is no easy way to thread a region handle to that point, dynamic scoping comes naturally as a solution. We might revisit this decision later. For instance regions could be threaded explicitly by default, and auxiliary functions could allow to set and retrieve the current region for situations where threading is not possible.

4.2 Sub-regions

Assuming that all roots have the same life-time as the external entrypoint works well if a fixed amount of work has to be done. However, for long-running function (for instance, an event loop driven by C-code), the over-approximation of lifetimes can be problematic. For these cases, we allow the introduction of sub-regions, valid in a local scope.

These sub-regions follow a stack discipline: they can be nested and are released in the reverse order of their allocation.

```
void mlregion_subenter(region_t *region);
void mlregion_subleave(region_t *region);
```

For instance, the following code avoids leaking roots while transforming all the elements of an array:

```
void process_item(value *acc, value*item);

void fold_array(value *acc, value *array)
{
    region_t region;
    size_t count = mlroot_get_size(array);
    for (size_t i = 0; i < count; ++i)
    {
```

```
        mlregion_subenter(&region);
        value *item = mlregion_new_root();
        *item = mlroot_get_field(array, i);
        process_item(*acc, *item);
        mlregion_subleave(&region);
    }
}
```

Macros can be used to automate some of the boilerplate.

4.3 Releasing the lock in a region

So far we have demonstrated the use of regions to allocate and manage OCaml memory. The concept can also be applied to the converse: preventing allocation and manipulation of OCaml memory in a given scope.

Although a multi-core runtime is being developed (Dolan et al., 2014), the vanilla OCaml runtime can only execute on a single thread of execution. When multiple C threads are in use, a lock is used by the OCaml runtime to ensure that only one of them executes OCaml code at any given time.

The C FFI provides an API for releasing the OCaml runtime lock in a given scope of code.

```
// Existing API
void caml_release_runtime_system(void);
void caml_acquire_runtime_system(void);
```

These APIs can be wrapped in corresponding `mlregion_{acquire,release}_runtime_system` functions that does additional bookkeeping to ensure proper use of regions while the runtime is released:

- new roots cannot be allocated,
- dereferencing a value is forbidden, most helper functions won't work,
- setting up normal regions is forbidden, but a special kind of region allows reacquiring the runtime.

All these restrictions can be tested at a moderate cost. While no checks are done at compile-time, misuse of the API can be reliably detected during execution.

```
// Wrappers for releasing the runtime
void mlregion_release_runtime_system(void);
void mlregion_acquire_runtime_system(void);

// Wrappers for locally reacquiring the runtime
void mlregion_reacquire_runtime_system(void);
void mlregion_rerelease_runtime_system(void);
```

4.4 Calling OCaml from region-managed code

The last feature that needs some special care from the region API is the ability to call OCaml closures from C code. When switching back to OCaml code, the runtime marks a region as disabled: the roots it contains are still reachable, but no new roots can be added to the region.

This helps detect and handle a few unfortunate cases:

- When re-entering C code from OCaml deeper in the call stack, an entrypoint that forgot to setup a region could allocate from the outer region by mistake.
- If we are unlucky, the OCaml thread scheduler could preempt the current thread and the re-entry would happen from another thread, damaging the internal datastructures of the regions library. By wrapping calls with custom code, we can rely on the OCaml runtime lock to also protect region sections.
- The OCaml closure could raise an exception. The native FFI deals with this situation by simply dropping roots from the local roots linked list: since the nodes allocated by `CAMLparam/local` macros are stored on the stack, when an exception is raised the local root and stack pointers are simply reset to their value before entering the C code. A workaround for regions is discussed below.

Handling exceptions. The OCaml native FFI provides two means for calling OCaml closures:

- the `CAMLcallback()` variants, that do not intercept exceptions. The C code will be aborted by directly jumping to the OCaml code that called an external function.
- the `CAMLcallback_exn()` variants, that tag the return value to distinguish exceptional case.

The return value of `CAMLcallback_exn()` should be tested for the exceptional case with `Is_exception_result` before resuming normal execution. Because our region management code needs to execute cleanup code when leaving a scope, we forbid the former case. The user-code has to handle the exceptional case without resorting to non-local control flow.

While it would have been possible to provide support for non-local jumps, it did not made much sense in the Qt case: the binding is implemented in C++, which allow arbitrary code to be executed when leaving a scope. C++ exceptions are expressive enough to handle all our requirements (non-local control flow, proper interaction with the regions and with OCaml GC), but the bindings themselves did not need that feature.

5 Future work

`mlroot` and `mlregion` emerged during the design of the Cuite library and are extracted from its core code. As the project is still in its infancy, it is evolving rapidly and the libraries have been properly tested only for the use cases stressed by Cuite. We still have to cover the rest of the FFI API.

Similarly, the support for runtime checks was only used for a few ad-hoc cases. Devising and implementing a robust suite of dynamic checks that are useful beyond Cuite is on our roadmap. Thanks to the transparent integration with the original FFI, this would help to debug existing bindings.

As Cuite is developped in C++, we have already developped a C++ layer on top of the C API to make it more idiomatic and remove some of the boilerplate – using references instead of pointers for dealing with roots, using RAII-idiom (Stroustrup, 1994) to ensure well-bracketed use of regions, etc. Extracting and generalizing this part would extend the usefulness of our library to other C++ bindings.

Finally, the only part of the OCaml runtime that we rely on and that is not already in the public interface is the representation of local roots. This part has already been stable for years. We are quite confident that it will be possible to get guarantees from the upstream developers that this API will not break in future releases of OCaml.

6 Related Work

The safety and simplicity of foreign function interfaces for OCaml has been approached from many angles.

O-Saffire (Furr and Foster, 2005) is a static analysis that works on the official OCaml FFI. It goes beyond checking the registration of roots and also checks that the value representation on the C and OCaml sides is compatible.

Unfortunately, O-Saffire has not received much changes since 2005 and we could not get it to work on a recent distribution of OCaml.

Ctypes (Yallop et al., 2018) proposes an alternative way to bind libraries. Rather than writing C code, a specification of the library is described in OCaml code. From this specification bridging code will be generated. The code can be instrumented to check for different safety properties.

Ctypes is very convenient for binding simple C functions. Since most of the code is described using OCaml combinators, it is easy to get bindings that are type-safe by construction. However it falls short on two fronts: there is limited support for calling C++ code and for manipulating objects with complex lifetimes or custom memory management rules. In these cases one has to write low-level code that follows the requirement of the library being bound. Achieving that through Ctypes combinators can prove more cumbersome and limited than directly writing the corresponding C code.

Caml-oxide (Dolan, 2018) is a proof of concept implementation of an OCaml FFI for Rust. In particular, it demonstrates that the restrictions applying to GC roots can be tracked by Rust type system.

This is the most promising approach for getting bindings that are safe by construction. It does so by leveraging the type systems of OCaml and of Rust. As such it cannot help with C/C++ libraries. The actual implementation is also too limited for most practical applications: it only covers a minimal part of the GC API, just enough to demonstrate the viability of the approach.

References

Timothy Bourke, Jun Inoue, and Marc Pouzet. Sundials/ML: interfacing with numerical solvers. ACM Workshop on ML, September 2016. URL https://hal.inria.fr/hal-01408230.

Stephen Dolan. caml-oxide: Safely mixing ocaml and rust. ML Family Workshop 2018, https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxtbHdvcmtzaG9wcGV8Z3g6NDNmNDlmNTcxMDk1YTRmNg, 2018.

Stephen Dolan, Leo White, and Anil Madhavapeddy. Multicore ocaml. OCaml Workshop (2014)., 2014.

Michael Furr and Jeffrey S. Foster. Checking type safety of foreign function calls. *SIGPLAN Not.*, 40(6):62–72, June 2005. ISSN 0362-1340. doi: 10.1145/1064978.1065019. URL http://doi.acm.org/10.1145/1064978.1065019.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. Living in harmony with the garbage collector. OCaml manual, §20.5.

Bjarne Stroustrup. *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994. ISBN 0-201-54330-3.

Jeremy Yallop, David Sheets, and Anil Madhavapeddy. A modular foreign function interface. *Sci. Comput. Program.*, 164:82–97, 2018. doi: 10.1016/j.scico.2017.04.002. URL https://doi.org/10.1016/j.scico.2017.04.002.

Arguments Cadencés dans un Compilateur Lustre Vérifié

Timothy Bourke[1,2] et Marc Pouzet[3,2,1]

[1] Inria Paris
[2] École normale supérieure, PSL University
[3] Sorbonne Universités, UPMC Univ. Paris 06

Résumé

Lustre est un langage synchrone pour programmer des systèmes avec des schémas-blocs desquels un code impératif de bas niveau est généré automatiquement. Des travaux récents utilisent l'assistant de preuve Coq pour spécifier un compilateur d'un noyau de Lustre vers le langage Clight de CompCert pour ensuite générer du code assembleur. La preuve de correction de l'ensemble relie la sémantique de flots de Lustre avec la sémantique impérative du code assembleur.

Chaque flot dans un programme Lustre est associé avec une « horloge » statique qui représente ses instants d'activation. La compilation transforme les horloges en des instructions conditionnelles qui déterminent quand les valeurs associées sont calculées. Les travaux précédents faisaient l'hypothèse simplificatrice que toutes les entrées et sorties d'un bloc partagent la même horloge. Cet article décrit une façon de supprimer cette restriction. Elle exige d'abord d'enrichir les règles de typage des horloges et le modèle sémantique. Ensuite, pour satisfaire le modèle sémantique de Clight, on ajoute une étape de compilation pour assurer que chaque variable passée directement à un appel de fonction a été initialisée.

1 Introduction

Les langages basés sur les schémas-blocs sont couramment utilisés dans la programmation des logiciels contrôle-commande critiques. Le langage Scade 6 [9] en est un exemple emblématique ainsi que son prédécesseur académique le langage synchrone flot de données Lustre [11]. Les compilateurs des langages schéma-blocs traduisent des modèles abstraits en du code impératif de bas niveau, typiquement en C ou en Ada. Le compilateur Scade est doté de plusieurs qualifications aux normes industrielles. L'avantage majeur de ces qualifications est que dans un processus de développement certifié où Scade est utilisé, les exigences ne doivent être tracées qu'aux modèles de conception de haut niveau plutôt qu'au code de bas niveau. L'inconvénient est que la qualification d'un compilateur est pénible et coûteuse.

Les assistants de preuve permettent d'encoder les spécifications formelles et les algorithmes et de vérifier leur cohérence et leurs propriétés sur ordinateur. Des travaux récents montrent comment appliquer de tels outils pour spécifier et raisonner sur les compilateurs de langages impératifs de bas niveau comme C, dans le compilateur CompCert [14], et de langages avec gestion de mémoire automatique comme Standard ML, dans le compilateur CakeML [13]. Dans le projet Vélus [6, 5], nous développons un prototype d'un compilateur pour le noyau d'un langage synchrone flot de données en utilisant l'assistant de preuve Coq [15]. Le but ultime est de fournir un schéma directeur pour appliquer un assistant de preuve à la spécification et à la vérification de langages schéma-bloc et d'avoir un aperçu de leur utilité pour faciliter la qualification d'outils industriels.

Les deux éléments principaux d'un langage schéma-bloc sont les « blocs », pour représenter les unités fonctionnelles comme des compteurs ou des filtres, et les « signaux », pour lier les entrées et les sorties des blocs. En Lustre, un signal est modélisé par un flot de valeurs et un bloc, appelé un *nœud*, est modélisé par une fonction de flots en flots. Les flots d'un programme Lustre

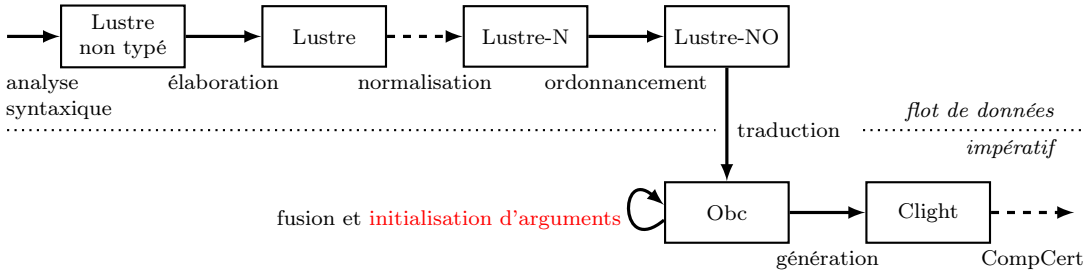


FIGURE 1 – Architecture d'un compilateur du code « modulaire et dirigé par les horloges ».

sont synchronisés les uns aux autres par rapport à une *horloge de base* qui est finalement réalisée par l'exécution répétée du code généré. Lustre fournit des opérateurs pour échantillonner ou sur-échantillonner un flot, ce qui permet l'activation conditionnelle de sous-parties d'un programme et, en particulier, la compilation efficace des constructions de haut niveau comme les machines à état hiérarchiques. L'utilisation des opérateurs d'échantillonnage est limitée par un système de typage [10] qui assure que les programmes acceptés peuvent être exécutés en mémoire bornée [7].

Les opérateurs d'échantillonnage et le typage par horloges statiques sont implémentés dans la première version du compilateur Vélus [6, 5] avec la restriction que les entrées et sorties d'un même nœud doivent toutes avoir la même horloge statique. Dans cet article, nous expliquons nos travaux récents pour enlever cette restriction et traiter les arguments cadencés.

L'architecture globale du compilateur Vélus est illustrée à la figure 1. Elle suit l'approche dite « modulaire et dirigée par les horloges » [1]. Le résultat de l'analyse syntaxique d'un fichier source est un arbre de syntaxe abstrait auquel l'*élaboration* ajoute des annotations validées pour les types et les horloges statiques. Une étape de *normalisation* est appliquée pour simplifier la forme des expressions et équations pour produire des programmes Lustre-N. La normalisation n'est pas encore implémenté, nous rejetons donc les programmes qui ne sont pas déjà en forme normale. Dans cet article, nous nous concentrons sur le langage Lustre-N. Les modifications qu'il a fallu apportées au langage Lustre-N et à ses règles d'horloge sont décrites dans la section 2. Les équations dans chaque nœud sont ordonnées par l'étape d'*ordonnancement* pour produire des programmes Lustre-NO, c'est-à-dire, Lustre-N et un prédicat sur l'ordre d'équations. Le prédicat est indispensable à l'étape suivante de *traduction* qui transforme les programmes Lustre-NO en un langage intermédiaire qui s'appelle Obc. Une optimisation de *fusion* factorise les instructions conditionnelles dans le code intermédiaire avant que l'étape de *génération* traduise un programme Obc en Clight pour ensuite être compilé par CompCert. La section 3 présent une vue d'ensemble de la génération du code impératif et décrit les obligations de preuves supplémentaires engendrées par notre introduction des arguments cadencés et l'utilisation de Clight. En particulier, il est nécessaire d'adapter Obc pour permettre l'utilisation de variables non initialisées dans les appels de méthode, comme le décrit la section 4, et puis d'assurer l'initialisation de ces variables en ajoutant une étape de compilation, indiquée en rouge dans la figure 1 et décrit dans la section 5, avant la génération de Clight.

2 Lustre normalisé et ses horloges

Un programme Lustre-N est une liste de nœuds. Chaque nœud définit une fonction entre une liste de variables d'entrée et une liste de variables de sortie. Un exemple est présenté dans la figure 2. Les définitions dans l'exemple ont pour seul but de présenter les caractéristiques du

```
node n1(c : bool; x : bool when c)
returns (v : int; o : int when c); (* ... *)

node n2(c1       : bool;
        c2, w, y : bool when c1;
        x        : bool when c2;
        z        : bool when not c1)
returns (o : int); (* ... *)

node f(m : bool; h : bool when m; x : bool)
returns (o1 : int; o2 : int when m);
var e0, e3 : bool when h;
    w      : int  when m;
    y      : int  when h;
    r1, e1 : bool when m;
    r2, e2 : bool when not m;
let
  e0 = (not (x when m)) when h;
  e1 = r1 and (x when m);
  e2 = r2 and (x when not m);
  e3 = e0 or ((x when m) when h);
  r1 = false fby not r1;
  r2 = true fby not r2;
  (w, y) = n1(h, e3);
  o1 = n2(m, h, (not x) when m, e1, e0, e2);
  o2 = merge h y (0 when not h);
tel
```

FIGURE 2 – Exemple : source Lustre

```
class f {
  instance i1 : n1;
  instance i2 : n2;
  memory r1, r2 : bool;

  method reset() { ··· }

  method step(m, h, x : bool)
  returns (o1, o2 : int32)
  var e0, e1, e2, e3 : bool; w, y : int32
  {
    e0 := 0;
    if (m) {
      e2 := 0;
      e1 := state(r1) & x;
      state(r1) := ! state(r1);
      if (h) {
        e0 := ! x;
        e3 := e0 | x
      } else {
        e3 := 0;
        skip
      };
      y := n1(i1).step(⟨h⟩, ⟨e3⟩);
      if (h) {
        o2 := y
      } else {
        o2 := 0
      }
    } else {
      o2 := 0;
      e1 := 0;
      e2 := state(r2) & x;
      state(r2) := ! state(r2)
    };
    o1 := n2(i2).step(⟨m⟩, ⟨h⟩, ! x,
                      ⟨e1⟩, ⟨e0⟩, ⟨e2⟩)
  }
}
```

FIGURE 3 – Exemple : Obc généré

langage et de sa compilation. Le programme est composé de trois nœuds : `n1`, `n2` et `f`. Nous nous concentrons sur le dernier nœud `f` dans lequel on instancie les autres nœuds, `n1` et `n2`, dont les définitions ne sont pas données.

Le nœud `f` a trois variables d'entrée de type `bool` : `m`, `h` et `x`. L'annotation « `bool when m` » à droite de `h` indique que cette variable représente un flot de valeurs booléennes qui ne sont *présentes* que quand `m` est `true`. Considérons, par exemple, les trois flots d'entrée suivants.

m	false	false	true	false	···	false	true	false	true	true	···
h			false		···		true		false	false	···
x	false	true	false	true	···	true	true	false	false	false	···

Les valeurs des flots `m` et `x` sont toujours présentes, mais celle du flot `h` est *absente* dans les colonnes où `m` est `false`. L'alignement des valeurs dans les colonnes se justifie par le modèle synchrone du temps. L'absence d'une valeur est indiquée par un espace vide.

Le corps d'un nœud est une liste d'équations. Il y en a une pour chaque variable de sortie, celles déclarées après `returns`, et chaque variable locale, celles déclarées après `var`. La signifi-

cation d'un nœud est insensible à l'ordre de ses équations. Les expressions qui définissent les équations sont construites à partir de constantes, variables, l'opérateur `when`, les opérateurs point par point unaires (p. ex., `not`) et binaires (p. ex., `and` et `or`) et l'opérateur `merge`[1]. L'opérateur `when` échantillonne un flot. Dans les flots d'entrée donnés en exemple ci-dessus, `h` égale « `x when m` ». Il est également possible de spécifier le flot complémentaire, par exemple, « `x when not m` », et de fusionner deux flots complémentaires : « `merge m h (x when not m)` » spécifie un flot identique à celui d'`x`.

La construction des expressions et équations est régie par un système de typage basé sur les horloges statiques. Une *horloge* est définie par la grammaire suivante que nous encodons comme le type inductif *clock* dans notre formalisation Coq.

$$ck ::= \mathsf{base} \mid ck \text{ on } (x = b)$$
$$b ::= \mathsf{true} \mid \mathsf{false}$$

Le constructeur `on` est associatif de gauche à droite. Dans l'exemple, l'horloge des variables `m` et `x` est base — elles sont toujours présentes dans `f` — et celle de `h` est base on $(m = \mathsf{true})$. Dans la définition de `e0`, l'horloge de « `x when m` » est base on $(m = \mathsf{true})$, tout comme celle de « `not (x when m)` » ; l'horloge de « `(not (x when m)) when h` » et donc aussi celle d'`e0` est base on $(m = \mathsf{true})$ on $(h = \mathsf{true})$. Les horloges servent à rejeter les expressions incorrectes. Par exemple, « `(x when m) + (x when not m)` » ne peut pas être calculée sans une mémoire tampon potentiellement sans borne puisque l'opérateur d'addition exige une valeur de chacun de ses flots d'entrée pour en produire une sur son flot de sortie. En Lustre, les arguments d'un opérateur binaire doivent avoir la même horloge statique. Les horloges jouent aussi un rôle important dans la génération du code impératif tel que décrit dans la section suivante.

Il y a trois formes d'équation en Lustre-N : (*a*) les équations simples, dans l'exemple, celles qui définissent `e0`, `e1`, `e2`, `e3` et `o2`, (*b*) les équations définies par un seul `fby`, comme celles de `r1` et `r2`, et (*c*) les équations définies par l'instanciation d'un autre nœud, comme celles de `w`/`y` et `o1`. Une équation `fby` spécifie un délai initialisé. Pour les entrées dans l'exemple, `r1` et `r2` prennent les valeurs suivantes.

r1			false		···		true		false	true	···
r2	true	false		true	···	false		true			···

La première valeur est définie par la constante à la gauche du `fby` et la suite égale au flot à sa droite (avec un délai unitaire).

L'exemple de la figure 2 n'est pas accepté par les versions précédentes de Vélus [6, 5], car les interfaces des nœuds contiennent une ou plusieurs variables sur des horloges différentes. En `n1`, par exemple, `c` et `v` ont l'horloge base alors que `x` et `o` ont l'horloge base on $(c = \mathsf{true})$.

Plusieurs modifications aux parties flot de données du compilateur — les langages et algorithmes au-dessus de la ligne en pointillé de la figure 1 — sont nécessaires pour permettre les interfaces plus riches. Les modifications de l'analyseur syntaxique et de l'arbre de syntaxe abstrait étaient mineures. L'élaboration a été adaptée pour déduire les horloges des constantes — dans l'exemple, à la ligne 26, le « `when not h` » autour de la constante `0` est ajouté automatiquement — et pour contrôler les horloges dans les équations où un nœud est instancié. L'étape d'ordonnancement et le système de typage ne changent pas. La plupart des modifications s'appliquent au système d'horloges et aux modèles sémantiques.

Dans notre compilateur, un nœud est représenté par un enregistrement qui rassemble son nom, les listes des déclarations de variables d'entrée (nins), des variables locales (nvars) et des

1. L'utilisation de `merge` est restreinte dans un programme normalisé, mais les règles précises ne sont pas importantes ici.

$$\textsc{SameVar-None}\quad \mathsf{SameVar\ None}\ e \qquad \textsc{SameVar-Some}\quad \mathsf{SameVar\ (Some}\ z)\ z \qquad \textsc{inst-base}\quad inst_{sub}\ nck\ \mathsf{base} = \mathsf{Some}\ nck$$

$$\textsc{inst-on}\quad \frac{inst_{sub}\ nck\ ck = \mathsf{Some}\ ck' \qquad sub\ x = \mathsf{Some}\ x'}{inst_{sub}\ nck\ (ck\ \mathsf{on}\ (x = b)) = \mathsf{Some}\ \big(ck'\ \mathsf{on}\ (x' = b)\big)}$$

$$\textsc{Clk-EqApp}\quad \frac{\mathsf{find\text{-}node}\ f\ G = \mathsf{Some}\ n \qquad (\forall x,\ \neg \mathsf{In}\ x\ (\mathsf{map\ fst}\ n.(\mathsf{nouts})) \implies osub\ x = \mathsf{None}) \qquad \mathsf{Forall2}\left(\lambda(x,(ty,ck))\ e,\ \mathsf{SameVar}\ (isub\ x)\ e \wedge \begin{array}{c}\exists ck',\ inst_{isub}\ nck\ ck = \mathsf{Some}\ ck' \\ \wedge \qquad \mathsf{wc\_exp}\ e\ ck'\end{array}\right) n.(\mathsf{nins})\ es \qquad \mathsf{Forall2}\left(\begin{array}{l}\lambda(x,(ty,ck))\ y, \\ \qquad \wedge \exists ck', \\ \qquad\qquad \wedge\ inst_{isub \lhd\!+ osub}\ nck\ ck = \mathsf{Some}\ ck'\end{array}\begin{array}{c}(isub \lhd\!+ osub)\ x = \mathsf{Some}\ y \\ \mathsf{In}\ (y, ck')\ vars\end{array}\right) n.(\mathsf{nouts})\ xs}{\mathsf{wc\_equation}(xs =^{nck} f(es))}$$

FIGURE 4 – La règle d'horloges statiques d'instanciation de nœud dans Lustre-N.

variables de sortie (nouts), la liste d'équations et quelques prédicats de bonne formation. Les déclarations de variables sont du type list (ident × (type × clock)).

Nous introduisons le prédicat wc_env pour affirmer la bonne formation des horloges dans une déclaration. Ce prédicat est utilisé trois fois : wc_env nins, wc_env (nins ++ nouts) et wc_env (nins ++ nvars ++ nouts). De plus, elle exige que chaque équation satisfasse un prédicat wc_equationqui affirme la bonne formation des horloges dans les équations.

Les cas de wc_equation pour les équations simples et `fby`s suivent simplement la structure des termes. Les définitions associées aux instanciations des nœuds sont décrites dans la figure 4. Étant donné un programme G et une liste de déclarations de variables $vars$, la règle Clk-EqApp définit le prédicat pour une équation « $xs =^{nck} f(es)$ » où xs est une liste de noms de variable, nck est l'horloge de base de l'instanciation, f est le nom du nœud et es est une liste d'expressions. La règle exige l'existence d'un nœud n associé à f dans G et deux fonctions partielles $isub$ et $osub$ qui associent respectivement les variables d'entrée et les variables de sortie dans la déclaration du nœud aux variables dans le contexte de son instanciation. Il faut que $osub$ n'associe que les variables de sorties déclarées pour le nœud.

La première proposition Forall2 [2] dans Clk-EqApp fait le lien entre chaque déclaration d'entrée dans $n.(\mathsf{nins})$ et l'expression correspondante dans es. Pour une variable d'entrée x d'horloge ck et l'expression correspondante e, il y a trois conditions : (a) si e est une variable z alors $isub\ x = \mathsf{Some}\ z$, c'est-à-dire, les variables qui peuvent apparaître dans les horloges déclarées dans l'interface d'un nœud sont cohérentes avec celles qui peuvent apparaître dans les horloges de es et de xs, (b) l'instanciation de la horloge ck en substituant ses variables selon $isub$ et son horloge de base avec nck est une horloge ck', pour laquelle (c) l'expression e est bien cadencée.

La deuxième proposition Forall2 dans Clk-EqApp relie chacune des variables de sortie dans $n.(\mathsf{nouts})$ à la variable correspondante dans xs. Pour une variable de sortie x d'horloge ck et la variable correspondante y, il y a trois conditions : (a) x est remplacé, soit par $isub$ soit par

2. Forall2 P xs ys est satisfait seulement si $xs = ys = []$ ou si P (hd xs) (hd ys) et Forall2 P (tl xs) (tl ys).

osub, par y[3], (*b*) y est déclarée dans *vars* avec horloge ck', (*c*) l'instanciation de ck en utilisant *isub*, *osub* et *nck* est ck'.

À titre d'exemple de la règle Clk-EqApp, considérons l'interface de `n1` dans la figure 2.

$$n_1.(\mathsf{nins}) = [(c, (\mathsf{bool}, \mathsf{base})), (x, (\mathsf{bool}, \mathsf{base\ on}\ (c = \mathsf{true})))]$$
$$n_1.(\mathsf{nouts}) = [(v, (\mathsf{int}, \mathsf{base})), (o, (\mathsf{int}, \mathsf{base\ on}\ (c = \mathsf{true})))]$$

Ce nœud est instancié dans `f` sur l'horloge $\mathsf{base\ on}\ (m = \mathsf{true})$ avec deux arguments $es = [h, e_3]$ pour définir $xs = [w, y]$. Pour les substitutions $isub = [c \mapsto h, x \mapsto e_3]$ et $osub = [v \mapsto w, o \mapsto y]$, les horloges des entrées sont instanciées à $\mathsf{base\ on}\ (m = \mathsf{true})$ et $\mathsf{base\ on}\ (m = \mathsf{true})\ \mathsf{on}\ (h = \mathsf{true})$, en concordance avec celles déclarées pour h et e_3, et les horloges des sorties sont instanciées à $\mathsf{base\ on}\ (m = \mathsf{true})$ et $\mathsf{base\ on}\ (m = \mathsf{true})\ \mathsf{on}\ (h = \mathsf{true})$, en concordance avec celles déclarées pour w et y.

La généralisation des horloges des instanciations entraîne deux modifications peu invasives aux modèles sémantiques. La première a été de remplacer une contrainte forçant les valeurs des flots d'entrée et de sortie d'être tous présentes ou absentes ensemble aux mêmes instants par une contrainte forçant l'horloge de base du nœud à être vrai exactement quand au moins une des entrées est présente. La seconde a été de contraindre les présences et absences de variables à correspondre avec leurs horloges déclarées. L'utilisation des annotations d'horloge non seulement pour contrôler les comportements dynamiques, mais aussi pour les contraindre peut sembler peu satisfaisante. En Lustre-N cependant, on a voulu donner une signification déterministe aux composants « isolés » comme « `x = false fby (not x)` » qui sont autrement libres d'avoir une valeur présente ou absente à n'importe instant. Ce problème ne se pose pas dans Lustre non normalisé où la valeur à gauche d'un `fby` est une expression et non une constante. Une constante s'exécute par définition sur l'horloge de base alors que les `when`s peuvent être ajoutés à une expression pour adapter son rythme. La solution se trouvera dans le nouveau modèle sémantique de Lustre. Nous réservons à l'avenir également les preuves pour lier les règles d'horloges aux propriétés du modèle sémantique

2.1 Travaux connexes

Les travaux précédents [10] sur un langage flot de données d'ordre supérieur démontrent que les horloges statiques peuvent être formulées comme des types dépendants et traités par les techniques utilisés dans les systèmes de typage à la ML. Ces horloges sont stratifiées en schèmes d'horloges, horloges, horloges de flot et variables porteuses [10, §4]. Tout comme pour les types polymorphes de ML, un schème d'horloge est créé à la déclaration d'un nœud par une quantification sur les variables libres dans les horloges de ses entrées et sorties. Un schème est instancié par la substitution de ces variables libres chaque fois qu'un nœud est utilisé dans une équation. Dans notre cadre normalisé, les nœuds sont instanciés dans des équations distinctes, alors les noms des variables dans les horloges sont connus directement et les variables porteuses ne sont pas nécessaires.

Des travaux précédents [3, 4] décrivent le plongement superficiel d'un langage synchrone flot de données dans Coq où les horloges sont représentées comme des suites coinductives de booléens. Nous nous concentrons sur un plongement profond puisque notre but est d'extraire un compilateur qui contrôle et transforme les termes explicitement.

3. L'union de deux substitutions $S \lhd\!\!+ T$ avec priorité à S, c'est-à-dire, $(S \lhd\!\!+ T)(x)$, égal $S(x)$ si ce n'est pas None, sinon $T(x)$.

3 Compilation vers code impératif

Après ordonnancement, un programme Lustre-N est compilé en deux étapes. La première produit et optimise un code dans le langage impératif Obc. La deuxième génère un code Clight.

Le code Obc optimisé produit pour l'exemple est illustré à la figure 3 à l'exclusion des éléments en rouge. À ce stade, nous remarquons seulement que chaque équation devient une affectation et que des instructions conditionnelles sont introduites pour traduire les horloges du programme source. Par exemple, puisque l'horloge d'`e0` est base on $(m = \mathsf{true})$ on $(h = \mathsf{true})$, l'affectation correspondante est emboîtée dans les instructions conditionnelles suivantes.

```
if (m) { ··· if (h) { ··· } ··· }
```

En conséquence de cet encodage, dans une instanciation de nœud avec des arguments sur différentes horloges, les variables passées en argument n'ont pas nécessairement toujours été initialisées. Par exemple, lorsque `m = false`, les valeurs de `h`, `e1`, `e0` et `e2` sont indéterminées dans l'appel aux lignes 35 et 36.

Nous avons adapté Obc pour permettre l'utilisation de variables non initialisées dans les appels de méthode. Cependant la norme C99 précise que le passage de valeurs indéterminées dans un appel de fonction n'est pas bien défini : si un lvalue ne désigne pas un objet lors de son évaluation, le comportement n'est pas défini [4] [12, §6.3.2.1], pendant la préparation d'un appel de fonction, les arguments sont évalués, et à chaque paramètre est affecté la valeur de l'argument correspondant [5] [12, §6.5.2.2]. Cette interprétation est formalisée par les modèles sémantiques de Clight [2, figure 10]. Lors de l'évaluation d'un appel de fonction interne, les arguments sont d'abord évalués un à un [6]. La conversion de type appliquée à chacun doit rendre une valeur, mais une conversion à tout autre type que `void` de la valeur donnée aux variables locales non initialisées (`Vundef`) rend `None` [7].

Il y a au moins deux façons de contourner cette contrainte. On pourrait traduire Obc dans l'un des autres langages intermédiaires de CompCert, en l'occurrence Cminor où les arguments indéfinis sont permis. Cela nécessiterait toutefois de reproduire et de revérifier les fonctionnalités qui ne sont pas fournis par Cminor, comme l'empilement d'arguments. Une autre possibilité serait de modifier les modèles sémantiques de Clight et de corriger les preuves associées. Cette approche induirait du travail supplémentaire et le code généré ne serait plus conforme à la norme C. Nous avons donc décidé d'accepter les contraintes imposées par C99 et Clight et de chercher une autre solution.

L'approche adoptée dans Scade est de faire une extension inline des fonctions où une ou plusieurs entrées sont sous-échantillonnées [8]. Cela a trois avantages : les valeurs des arguments dans le code généré sont toujours définies, l'optimisation de fusion s'applique à travers les nœuds et les justifications de correction sont faites dans le langage flot de données. Nous n'avons pas suivi cette approche parce que l'extension inline n'a pas encore été implémentée dans Vélus et nous étions curieux d'évaluer une différente solution qui maintien la modularité du programme source dans le code généré.

Au moins deux autres solutions sont possibles lors de la traduction en Obc et la génération de Clight. La première est d'implémenter les appels de fonction avec un argument pointeur vers un enregistrement contenant toutes ou une partie des valeurs d'entrée. Le pointeur serait initialisé et on sait par construction que le code généré ne lit jamais les valeurs indéterminées.

4. « . . . if an lvalue does not designate an object when it is evaluated, the behavior is undefined. »
5. « In preparing for the call to a function, the arguments are evaluated, and each parameter is assigned the value of the corresponding argument. »
6. Voir `cfrontend/Clight.v` : `eval_exprlist`.
7. Voir `cfrontend/Cop.v` : `sem_cast`.

La seconde solution est de traiter certaines variables d'entrées comme des variables d'état qui sont mises dans la mémoire statique et sont donc toujours bien définies en C. L'une ou l'autre solution compliquerait les fonctions de compilation et leurs preuves de correction.

En fin de compte, la solution la plus simple est d'initialiser toutes les variables au début de chaque fonction. Dans bien des cas, CompCert sait éliminer les écritures qui ne sont pas nécessaires et le coût des autres initialisations est probablement insignifiant. Nous avons essentiellement adopté cette solution, mais avec deux améliorations. D'abord, nous exploitons la preuve de correction et quelques petites modifications à la sémantique d'Obc pour éviter d'introduire d'écritures inutiles, tel que décrit dans la section 4. Ensuite, nous ajoutons une étape de compilation pour réécrire les programmes Obc en ajoutant les initialisations nécessaires, telle que décrit dans la section 5. La nouvelle étape de compilation utilise une heuristique qui tente de minimiser à la fois les répétitions inutiles d'écritures et le nombre d'initialisations ajoutées sans trop compliquer les obligations de preuve concomitantes.

4 La génération du code intermédiaire

La traduction de Lustre-N en Obc est détaillée ailleurs [6, 5]. Ici nous nous concentrons sur la généralisation du passage des arguments dans les appels de méthodes.

Les expressions e et les commandes s du langage Obc sont définies par la grammaire suivante (avec l'omission des annotations de typage).

$$e ::= x \mid \texttt{st}(x) \mid c \mid \ \diamond e \mid e \oplus e \mid \langle e \rangle$$
$$s ::= x := e \mid \texttt{st}(x) := e \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\} \mid s\ ;\ s \mid xs := cls(i).f(es) \mid \texttt{skip}$$

Une expression est une variable (x), une variable d'état ($\texttt{st}(x)$), une constante (c), un opérateur unaire ($\diamond$), un opérateur binaire ($\oplus$) ou une *assertion de validité* ($\langle\cdot\rangle$). Une commande est une affectation à une variable, une affectation à une variable d'état, une instruction conditionnelle, une composition séquentielle, un appel de méthode ou une instruction nulle. Nous définissons une sémantique à grands pas pour les expressions et les commandes. Nous écrivons

$$menv, env \vdash e \Downarrow vo$$

pour affirmer que l'évaluation d'une expression e dans l'environnement de variables d'état $menv$ et l'environnement de variables env produit la valeur optionnelle vo. Nous écrivons

$$p, menv, env \vdash s \Downarrow (menv', env')$$

pour affirmer que l'évaluation d'une commande s dans le programme p et les environnements $menv$ et env produit les environnements $menv'$ et env'.

Un environnement de variables (env) est une application partielle entre variables et valeurs. Un environnement de variables d'état ($menv$) a deux composants : une application partielle entre variables d'état et valeurs ($menv_{\mathsf{v}}$) et une application entre noms de nœud et environnements de variables d'état ($menv_{\mathsf{o}}$)[8].

Les règles de sémantique d'Obc sont présentées dans les figures 5 et 6. La seule chose de remarquable est le traitement de valeurs partielles, autrement dit, la possibilité qu'une variable ne soit pas définie dans un environnement. Une expression x s'évalue à None si x n'est pas défini dans l'environnement de variables, c'est-à-dire, si une valeur n'a pas été affectée à lui. Toutes les autres expressions doivent s'évaluer à une valeur « Some ». L'évaluation d'une commande d'affectation n'est pas définie quand son expression s'évalue à None. Cependant, les appels de méthode ne sont pas ainsi contraints, il est donc possible de passer et de renvoyer une variable qui n'a pas forcément été écrite. La règle pour les appels de méthode utilise une fonction adds

8. Ces applications partielles sont réalisées dans Coq avec l'aide de la bibliothèque FMapPositive.

$$\textsc{Obc-Exp-Var}\ \frac{env\ x = vo}{menv, env \vdash x \Downarrow vo} \qquad \textsc{Obc-Exp-State}\ \frac{menv_{\mathsf{v}}\ x = \mathsf{Some}\ v}{menv, env \vdash \mathtt{st}(x) \Downarrow \mathsf{Some}\ v} \qquad \textsc{Obc-Exp-Const}\ menv, env \vdash c \Downarrow \mathsf{Some}\ [\![c]\!]$$

$$\textsc{Obc-Exp-Unop}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v \qquad [\![\diamond\ v]\!] = \mathsf{Some}\ v'}{menv, env \vdash \diamond\ e \Downarrow \mathsf{Some}\ v'} \qquad \textsc{Obc-Exp-Assert}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v}{menv, env \vdash \langle e \rangle \Downarrow \mathsf{Some}\ v}$$

$$\textsc{Obc-Exp-Binop}\ \frac{menv, env \vdash e_1 \Downarrow \mathsf{Some}\ v_1 \qquad menv, env \vdash e_2 \Downarrow \mathsf{Some}\ v_2 \qquad [\![v_1 \oplus v_2]\!] = \mathsf{Some}\ v'}{menv, env \vdash e_1 \oplus e_2 \Downarrow \mathsf{Some}\ v'}$$

FIGURE 5 – Les règles sémantiques des expressions Obc.

$$\textsc{Obc-Stmt-Assign}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v}{p, menv, env \vdash x := e \Downarrow (menv, env[x \mapsto v])}$$

$$\textsc{Obc-Stmt-StAssign}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some}\ v}{p, menv, env \vdash \mathtt{st}(x) := e \Downarrow (menv_{\mathsf{v}}[x \mapsto v], env)}$$

$$\textsc{Obc-Stmt-Comp}\ \frac{p, menv, env \vdash s_1 \Downarrow (menv_1, env_1) \qquad p, menv_1, env_1 \vdash s_2 \Downarrow (menv_2, env_2)}{p, menv, env \vdash s_1\ ;\ s_2 \Downarrow (menv_2, env_2)}$$

$$\textsc{Obc-Stmt-Ite-True}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some\ true} \qquad p, menv, env \vdash s_1 \Downarrow (menv', env')}{p, menv, env \vdash \mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\} \Downarrow (menv', env')}$$

$$\textsc{Obc-Stmt-Ite-False}\ \frac{menv, env \vdash e \Downarrow \mathsf{Some\ false} \qquad p, menv, env \vdash s_2 \Downarrow (menv', env')}{p, menv, env \vdash \mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\} \Downarrow (menv', env')}$$

$$\textsc{Obc-Stmt-Skip}\ p, menv, env \vdash \mathtt{skip} \Downarrow (menv, env)$$

$$\textsc{Obc-Stmt-Call}\ \frac{\begin{array}{c}\mathsf{Forall2}\ (\lambda e\ vo,\ menv, env \vdash e \Downarrow vo)\ es\ vos \\ \mathsf{find\text{-}class}\ cls\ p = \mathsf{Some}\ (c, p') \qquad \mathsf{find\text{-}method}\ f\ c.(\mathsf{methods}) = \mathsf{Some}\ m \qquad |vos| = |m.(\mathsf{nins})| \\ p', menv_{\mathsf{o}}\ i, \mathsf{adds}(\mathsf{map\ fst}\ m.(\mathsf{nins}), vos, \mathsf{empty}) \vdash m.(\mathsf{mbody}) \Downarrow (omenv', oenv') \\ \mathsf{Forall2}\ (\lambda x\ vo,\ oenv'\ x = vo)\ (\mathsf{map\ fst}\ m.(\mathsf{nouts}))\ ros\end{array}}{p, menv, env \vdash ys := cls(i).f(es) \Downarrow (menv_{\mathsf{o}}[i \mapsto omenv'], \mathsf{updates}(env, ys, ros))}$$

FIGURE 6 – Les règles sémantiques des commandes Obc.

$$\textsc{NoOps-base}\ \ \mathsf{NoOps\ base}\ e \qquad \textsc{NoOps-const}\ \ \mathsf{NoOps}\ ck\ c \qquad \textsc{NoOps-var}\ \ \mathsf{NoOps}\ ck\ x$$

$$\textsc{NoOps-when}\ \ \frac{\mathsf{NoOps}\ ck\ e}{\mathsf{NoOps}\ (ck\ \mathsf{on}\ (x = b))\ (e\ \texttt{when}\ (\texttt{not})\ x)}$$

FIGURE 7 – La condition de normalisation en Lustre-N entre une horloge d'une entrée d'un nœud et une expression passée comme argument.

pour créer un environnement initial pour les variables d'entrée et une fonction `updates` pour soit copier les valeurs qui sont définies, soit supprimer les valeurs non définies (qui sont donc égales à None).

Le traitement des valeurs partielles en Obc demande une attention particulière dans la preuve de correction de la traduction de Lustre-N en Obc. Cette preuve établit un rapport entre présence et absence dans le programme source et l'exécution conditionnelle des commandes dans le code généré. L'invariant sous-jacent est souvent suffisant pour garantir qu'une variable est bien définie dans les commandes où elle est lue. C'est le cas pour les variables dans les expressions générées d'équations simples et de `fby`s, et aussi pour celles dans les conditions générées à partir d'horloges statiques (p. ex., `h` à la ligne 16 de la figure 3). Par contre, un soin supplémentaire doit être apporté aux expressions dans les appels de méthode.

Considérons d'abord le problème posé dans un appel de méthode par une expression qui contient un opérateur unaire ou binaire. Par exemple, dans la figure 2 à la ligne 24 : le programme qui résulte du remplacement de `e3` par « `e3 + (r1 when h)` » est légitime, mais sa traduction en Obc, « `e3 + r1` », n'est pas définie lorsque `h = false` puisque le résultat d'une addition avec None n'est pas défini. Alors qu'une addition est toujours définie en Clight, ce n'est pas le cas pour tous les opérateurs : l'important est d'assurer qu'une propriété vérifiée sur un programme source, comme l'absence de division entière par zéro, l'est également vrai pour le code généré. Pour garantir l'existence d'un modèle sémantique pour le code généré, nous exigeons que le programme source satisfasse le prédicat NoOps dont la définition est présentée dans la figure 7. Ce prédicat doit être satisfait par tous les arguments de toutes les instanciations de nœud. Il définit des combinaisons admissibles entre une horloge statique de l'interface d'un nœud (celles dans nins) et une expression passée comme argument. Pour l'horloge base, toutes les expressions sont permises puisque les règles d'horloges et l'invariant de correction garantissent que les valeurs des variables dans l'expression sont toujours présentes lors de l'activation du nœud. Les constantes et les variables sont toujours permises. Autrement, un ou plusieurs niveaux d'échantillonnage peuvent être enlevés pourvu que l'expression échantillonnée soit présente quand l'horloge de base du nœud est vraie. On exploite cette condition syntaxique dans la preuve de correction de la traduction pour démontrer que le code généré a une sémantique. La condition doit être garantie par l'étape de normalisation ou contrôlée lors de l'élaboration.

Considérons maintenant les autres expressions d'arguments possibles. La sémantique d'une constante est toujours définie et ce cas ne pose donc aucun problème. Une variable x en Lustre-N est traduite en Obc soit comme un variable x soit comme une variable d'état `st`(x). Une variable d'état est toujours définie quelle que soit son horloge dans le programme source. Pour les autres variables, il y a assez d'information dans la preuve de correction pour garantir que celles sur l'horloge de base d'une instanciation de nœud sont toujours définies lors de l'exécution de l'appel de méthode correspondant dans le code généré. Pour marquer ce fait, nous enveloppons ces variables dans une assertion de validité : $\langle x \rangle$. L'évaluation d'une assertion ne rend une valeur que quand l'expression qu'elle enveloppe ne s'évalue pas à None. Certaines assertions de validité sont ajoutées par l'étape de traduction en Obc. La preuve de correction de la traduction garantit

qu'elles sont bien définies. On profite ensuite des assertions dans la preuve de correction de la génération de Clight.

Dans le code généré illustré dans la figure 3, les assertions (en noire) autour de `h` à la ligne 23 et de `m` à la ligne 35 sont ajoutées lors de la traduction vers Obc et donc justifiées par la preuve de correction correspondante. Nous pouvons savoir que ces variables sont toujours définies en amont des appels de méthode : `m`, car elle est toujours présente, et `h`, car elle est présente quand `m` est vraie et l'appel de méthode à la ligne 23 se trouve dans l'instruction conditionnelle `if (m) { ··· }`. Dans ce cas, les deux variables sont des entrées, mais le même raisonnement se tient pour les variables locales. Il n'est pas nécessaire d'ajouter une assertion de validité autour des expressions comme le `! x` à la ligne 35 : elles doivent satisfaire NoOps et leur sémantique est donc forcément définie.

Les variables dans un appel de méthode qui ne sont pas sur l'horloge de base du nœud instancié doivent être explicitement initialisées pour justifier l'addition des assertions de validité (celles illustrées en rouge) qui sont essentielles pour la preuve de correction de la génération de Clight. Cette transformation est réalisée par l'étape de compilation appelée, dans la figure 1, « initialisation d'arguments » et décrite dans la section suivante.

5 L'initialisation d'arguments

Le code Obc généré contient des assertions de validité pour toutes les variables d'argument qui sont garanties d'être définies lors d'un appel de méthode. L'idée maintenant est d'ajouter des assertions de validité à toutes les autres variables d'arguments et de les justifier en ajoutant des affectations supplémentaires auparavant. Les preuves de correction des nouveaux programmes produits ainsi et les codes Clight générés par la suite s'appuient sur trois invariants introduits dans cette section. En outre, bien que les variables qui sont définies dans l'ancien programme sont définies avec les mêmes valeurs dans le nouveau programme, les deux programmes ne sont pas strictement équivalents à cause des affectations supplémentaires. Nous devons donc introduire une notion de raffinement entre programmes Obc pour énoncer et démontrer la propriété de correction.

5.1 La nouvelle étape de compilation

Les fonctions pour ajouter les assertions de validité et affectations à une commande Obc sont illustrées dans la figure 8. La fonction *add_defaults_stmt* est appelée avec deux arguments : un ensemble[9] *required* de variables à initialiser et une commande *s* à transformer.

(s′ , required′ , sometimes, always) = add_defaults_stmt required s

Cette fonction rend s', une nouvelle commande, $required'$, l'ensemble des variables qui doivent être initialisées avant l'exécution de s' , *sometimes*, l'ensemble des variables qui sont parfois mais pas toujours écrites par s' et *always*, l'ensemble des variables qui sont toujours écrites par s' [10].

Les cas pour `skip`, l'affectation à une variable et l'affectation à une variable d'état sont simples. Notons que le cas pour l'affectation à une variable enlève cette variable de l'ensemble *required* et l'ajoute à l'ensemble *always*. Pour les appels de méthode, les variables dans xs sont enlevés de l'ensemble *required* avant d'appliquer la fonction *add_valid* successivement sur les expressions d'argument. Des assertions de validité sont ajoutées pour toutes les variables qui n'en ont pas déjà et ces variables sont ajoutées à l'ensemble *required*. Autrement dit, nous

9. Nous représentons les ensembles dans Coq avec la bibliothèque MSetPositive.
10. Nous démontrons que sometimes ∩ always = ∅.

```
Variable type_of_var : ident ⟹ option type.

Definition add_write x s :=
  match type_of_var x with None ⇒ s | Some ty ⇒ (x := (init_type ty)) ; s end.

Definition add_writes W s := PS.fold add_write W s.

Definition add_valid (e : exp) (esreq : list exp * PS.t) :=
  match e, esreq with
  | Var x ty, (es, req) ⇒ (Valid e :: es, req ∪ {x})
  | _,        (es, req) ⇒ (       e :: es, req)
  end.

Fixpoint add_defaults_stmt (req: PS.t) (s: stmt) : stmt * PS.t * PS.t * PS.t :=
  match s with
  | skip ⇒ (s, req, ∅, ∅)
  | x := e ⇒ (s, req − {x}, ∅, {x})
  | st(x) := e ⇒ (s, req, ∅, ∅)

  | xs := f(o).m(es) ⇒
    let (es', req') := fold_right add_valid ([], ps_removes xs req) es
    in (xs := f(o).m(es'), req', ∅, of_list xs)

  | s1 ; s2 ⇒
    let (t2, req2, st2, al2) := add_defaults_stmt req s2 in
    let (t1, req1, st1, al1) := add_defaults_stmt req2 s1 in
    (t1 ; t2, req1, (st1 − al2) ∪ (st2 − al1), al1 ∪ al2)

  | if e { s1 } else { s2 } ⇒
    let (t1, req1, st1, al1) := add_defaults_stmt ∅ s1 in
    let (t2, req2, st2, al2) := add_defaults_stmt ∅ s2 in
    let (al1_req, al2_req) := (al1 ∩ req, al2 ∩ req) in
    let (w1, w2) := (al2_req − al1_req, al1_req − al2_req) in
    let w := ((st1 ∩ req) − w1) ∪ ((st2 ∩ req) − w2) in
    let (al1', al2') := (al1 ∪ w1, al2 ∪ w2) in
    let (st1', st2') := (st1 − w1, st2 − w2) in
    (add_writes w (if e { add_writes w1 t1 } else { add_writes w2 t2 }),
     (((req − al1_req) − al2_req) ∪ req1 ∪ req2) − w,
     (st1' ∪ st2' ∪ (al1' − al2') ∪ (al2' − al1')) − w,
     (al1' ∩ al2') ∪ w)
  end.
```

FIGURE 8 – Fonction principale de l'initialisation d'arguments dans un programme Obc.

affirmons qu'une variable est définie et l'ajoutons à l'ensemble de variables à initialiser. La liste de variables *xs* est transformée dans un ensemble *always*. Dans le cas de la composition séquentielle, on travail en arrière, en propageant les ensembles *required*, avant de recalculer les ensembles *sometimes* et *always*.

Le cas pour les instructions conditionnelles est le plus complexe. Il y a un appel récursif pour la commande dans chaque branche avec l'ensemble vide pour l'argument *required*. Sont calculés, l'ensemble *w1* des initialisations à faire avant la commande *s1*, l'ensemble *w2* des initialisations à faire avant la commande *s2* et l'ensemble *w* des initialisations à faire avant les deux branches. Dans la première branche, on ajoute des affectations aux variables dans *required* qui sont toujours écrites par *s2* mais qui ne sont pas toujours écrites par *s1*. La seconde branche est transformée de façon similaire. Les variables dans *required* qui sont parfois mais pas toujours écrites dans les deux branches sont initialisées avant la nouvelle instruction conditionnelle, en prenant en compte les autres nouvelles initialisations. Il y a une fonction auxiliaire, *type_of_var*,

qui associe une variable à son type, et une autre, *init_type*, qui associe un type à sa valeur par défaut, et encore d'autres pour calculer les ensembles *required*, *sometimes* et *always*.

La fonction *add_defaults_stmt* est utilisée dans la fonction qui transforme les méthodes.

```
add_defaults_method (m : method) : method
```

Elle construit l'argument *type_of_var* à partir des déclarations de *m*, transforme le corps de la méthode avec *add_defaults_stmt* en passant l'ensemble des variables de sortie comme *req* et ajoute les initialisations qui restent à faire après exclusion des variables d'entrée.

```
let (body', req, st, al) := add_defaults_stmt tyenv (ps_adds (map fst
outs) ∅) body in
add_writes tyenv (ps_removes (map fst ins) req) body')
```

La fonction *add_defaults_stmt* est conçue pour traiter la structure du code Obc généré de programmes Lustre-N et transformé par l'optimisation de fusion. Elle exprime un compromis entre l'ajout d'initialisations inutiles et le nombre d'affections supplémentaires à faire. D'un côté, l'ajout d'affectations aux feuilles d'un arbre d'instructions conditionnelles pourrait augmenter énormément la taille de code, de l'autre, leur ajout inconsidéré aux sommets peut augmenter inutilement le nombre d'affectations. La fonction présentée ci-dessus n'est pas toujours optimale, mais un meilleur résultat demanderait une analyse plus compliquée qui ne ferait que reconstruire en Obc l'information encodée par les horloges statiques dans le programme source.

Les assertions de validité et les initialisations ajoutées pour l'exemple sont indiquées en rouge dans la figure 3. Pour chaque nouvelle assertion de validité aux lignes 23, 35, et 36 une initialisation est ajoutée auparavant dans le programme. Puisque `h` est une entrée, nous supposons par récurrence que l'appelant l'a déjà initialisé. Une initialisation est ajoutée pour la variable de sortie `o2` qui n'est pas forcément écrite sinon. Ainsi, on peut supposer que les variables définies par un appel de méthode sont toujours initialisées, ce qui est nécessaire dans la preuve de correction d'*add_defaults_stmt*.

5.2 La relation de raffinement

La commande s' générée par *add_defaults_stmt* ne calcul pas forcément les mêmes résultats que la commande initiale s. Intuitivement, on s'attend que toutes les variables définies après exécution de s soient définies avec les mêmes valeurs après exécution de s'. Les valeurs des variables nouvellement définies ne sont pas importantes pourvue que les entrées et les sorties du nœud principal en Lustre-N (le nœud « main ») sont toutes sur l'horloge de base et donc toujours définies par le code généré. Nous exprimons ces intuitions formellement avec une relation de raffinement entre environnements.

$$env_2 \sqsubseteq env_1 \equiv \forall x\ v,\ env_2\ x = \mathsf{Some}\ v \implies env_1\ x = \mathsf{Some}\ v$$

Ensuite, nous entendrons la relation aux commandes exécutées dans deux programmes p_1 et p_2. L'idée est que l'exécution d'une commande « enrichie » s_1 dans un environnement env_1 qui est « plus défini » que env_2 produit un environnement identique à celui produit par la commande initiale s_2 exécuté dans env_2 sauf qu'il peut y avoir plus de variables définies. Malheureusement, une précondition P s'avère nécessaire pour démontrer le raffinement entre deux commandes et la relation est plus compliquée que l'on pourrait souhaiter.

$$\begin{aligned} s_2 \sqsubseteq_P^{p_2,p_1} s_1 \equiv\ & \forall menv\ menv'\ env_1\ env_2\ env_2', \\ & P\ env_1\ env_2 \implies \\ & env_2 \sqsubseteq env_1 \implies \\ & p_2, menv, env_2 \vdash s_2 \Downarrow (menv', env_2') \implies \\ & \exists env_1',\ p_1, menv, env_1 \vdash s_1 \Downarrow (menv', env_1') \ \wedge\ env_2' \sqsubseteq env_1' \end{aligned}$$

La relation s'étend directement aux méthodes et aux classes. On aboutit finalement à une définition récursive de raffinement entre programmes (défini de façon standard par un combinateur de point fixe et une relation bien fondée).

$$p_2 \sqsubseteq_P p_1 \equiv \forall n\ c_2\ p_2',\ \mathsf{find\text{-}class}\ n\ p_2 = \mathsf{Some}\ (c_2, p_2') \implies \exists c_1\ p_1',\ \mathsf{find\text{-}class}\ n\ p_1 = \mathsf{Some}\ (c_1, p_1')\ \wedge\ c_1 \sqsubseteq^{p_1,p_2}_{(P\ n)}\ c_2\ \wedge\ p_2' \sqsubseteq_P p_1'$$

5.3 Les invariants et la preuve de correction

La relation de raffinement permet d'énoncer succinctement le cœur du lemme de correction principal.

$$\mathit{add\_defaults\_stmt}\ \mathit{tyenv}\ \mathit{req}\ s = (t,\ \mathit{req}',\ \mathit{st},\ \mathit{al}) \implies s \sqsubseteq^{p,p'}_{(\mathit{in1\_notin2}\ \mathit{req}'\ (\mathit{st} \cup \mathit{al}))}\ t$$

La précondition *in1_notin2* exige que les variables dans *req′* soient définies dans l'environnement initial de t — elles sont initialisées par *add_defaults_method* — et que les variables écrites dans t ne soient pas définies dans l'environnement initial de s — c'est le cas, car les variables d'entrée sont enlevées par *add_defaults_method* et ne sont donc jamais réécrites. Quelques hypothèses supplémentaires sont nécessaires pour démontrer le raffinement, à savoir qu'il y a un raffinement entre *p′* et *p*, que les méthodes des classes dans *p* définissent toutes leurs sorties si toutes leurs entrées sont définies et que *s* est bien typé et satisfait l'invariant NoOverwrites. Le prédicat NoOverwrites est présenté dans la figure 9. Il affirme, essentiellement, qu'un programme est en « static single assignment form (SSA) », c'est-à-dire, que les variables ne sont écrites qu'une fois au plus. La règle importante s'appelle NoO-Seq. Ce prédicat est indispensable pour démontrer que les initialisations ajoutées par *add_defaults_stmt* n'écrasent jamais les valeurs existantes. Il est vrai du code généré des programmes Lustre-N et préservé par l'optimisation de fusion. Par contre, ce prédicat n'est pas préservé par la fonction *add_defaults_stmt* elle-même.

Le code produit par *add_defaults_stmt* satisfait le prédicat NoNakedVars. Ce prédicat est aussi présenté dans la figure 9. Il affirme simplement que les variables ne sont jamais passées directement dans un argument de méthode. Il garantit l'existence d'un modèle sémantique pour le code Clight généré par la suite. Sinon les preuves en aval ne demandent que quelques modifications mineures et l'addition d'un contrôle pour assurer que les entrées et les sorties du nœud principal (le nœud « main ») sont toutes sur l'horloge de base.

6 Conclusion

Cet article décrit l'extension d'un compilateur Lustre vérifié pour qu'il traite les définitions de nœuds où certaines entrées ou sorties sont absentes lors d'une activation — c'est-à-dire, d'accepter l'utilisation de sous-horloges dans les interfaces de nœud. Nous décrivons une règle d'horloge pour l'instanciation d'un nœud en Lustre normalisé et une condition sur les arguments d'un nœud qui suffissent pour démontrer la correction du code généré. Nous modifions les règles de sémantique de notre langage intermédiaire pour permettre l'utilisation de variables non définies dans les appels de méthode. Cependant, de tels appels ne peut pas être convertis directement en Clight, car la sémantique de Clight respect la norme C99 qui ne permet pas l'utilisation d'arguments non initialisés dans un appel de fonction. Nous avons réglé ce problème par (a) l'ajout d'assertions dans le code généré pour affirmer que certaines variables sont sûrement bien définies au point de leur utilisation avec un raisonnement qui s'appuie sur la preuve de correction existante (*b*) la définition et la vérification d'une nouvelle étape de compilation pour ajouter les assertions qui manquent et les initialisations correspondantes.

$$\textsc{CWI-Assign}\ \ \mathsf{CanWrite}\ x\ (x := e) \qquad \textsc{CWI-AssignSt}\ \ \mathsf{CanWrite}\ x\ (\mathtt{st}(x) := e)$$

$$\textsc{CWI-Ifte-T}\ \ \frac{\mathsf{CanWrite}\ x\ s_1}{\mathsf{CanWrite}\ x\ (\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\})}$$

$$\textsc{CWI-Ifte-F}\ \ \frac{\mathsf{CanWrite}\ x\ s_2}{\mathsf{CanWrite}\ x\ (\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\})} \qquad \textsc{CWI-Seq-1}\ \ \frac{\mathsf{CanWrite}\ x\ s_1}{\mathsf{CanWrite}\ x\ (s_1\ ;\ s_2)} \qquad \textsc{CWI-Seq-2}\ \ \frac{\mathsf{CanWrite}\ x\ s_2}{\mathsf{CanWrite}\ x\ (s_1\ ;\ s_2)}$$

$$\textsc{CWI-Call}\ \ \frac{\mathsf{In}\ x\ xs}{\mathsf{CanWrite}\ x\ (xs := cls(i).f(es))} \qquad \textsc{NoO-Assign}\ \ \mathsf{NoOverwrites}\ (x := e) \qquad \textsc{NoO-Assignst}\ \ \mathsf{NoOverwrites}\ (\mathtt{st}(x) := e)$$

$$\textsc{NoO-Skip}\ \ \mathsf{NoOverwrites}\ \mathtt{skip} \qquad \textsc{NoO-Ifte}\ \ \frac{\mathsf{NoOverwrites}\ s_1 \qquad \mathsf{NoOverwrites}\ s_2}{\mathsf{NoOverwrites}\ (\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\})}$$

$$\textsc{NoO-Seq}\ \ \frac{\begin{array}{c}(\forall x, \mathsf{CanWrite}\ x\ s_1 \implies \neg\mathsf{CanWrite}\ x\ s_2) \\ (\forall x, \mathsf{CanWrite}\ x\ s_2 \implies \neg\mathsf{CanWrite}\ x\ s_1) \qquad \mathsf{NoOverwrites}\ s_1 \qquad \mathsf{NoOverwrites}\ s_2\end{array}}{\mathsf{NoOverwrites}\ (s_1\ ;\ s_2)}$$

$$\textsc{NoO-Call}\ \ \mathsf{NoOverwrites}\ (xs := cls(i).f(es)) \qquad \textsc{NNV-Assign}\ \ \mathsf{NoNakedVars}\ (x := e) \qquad \textsc{NNV-Assignst}\ \ \mathsf{NoNakedVars}\ (\mathtt{st}(x) := e)$$

$$\textsc{NNV-Skip}\ \ \mathsf{NoNakedVars}\ \mathtt{skip} \qquad \textsc{NNV-Ifte}\ \ \frac{\mathsf{NoNakedVars}\ s_1 \qquad \mathsf{NoNakedVars}\ s_2}{\mathsf{NoNakedVars}\ (\mathtt{if}\ e\ \{s_1\}\ \mathtt{else}\ \{s_2\})}$$

$$\textsc{NNV-Seq}\ \ \frac{\mathsf{NoNakedVars}\ s_1 \qquad \mathsf{NoNakedVars}\ s_2}{\mathsf{NoNakedVars}\ (s_1\ ;\ s_2)} \qquad \textsc{NNV-Call}\ \ \frac{\mathsf{Forall}\ (\lambda e,\ e \neq x)\ es}{\mathsf{NoNakedVars}\ (xs := cls(i).f(es))}$$

FIGURE 9 – Invariants Obc : CanWrite, NoOverwrites et NoNakedVars.

Remerciements Nous remercions J.-L. Colaço pour ses explications sur Scade, X. Leroy pour ses explications et ses conseils sur CompCert et A. Guatto et L. Mandel pour leurs suggestions. Ce travail a été soutenu par le projet ITEA 3 14014 ASSUME.

Références

[1] D. Biernacki, J.-L. Colaço, G. Hamon et M. Pouzet : Clock-directed modular code generation for synchronous data-flow languages. *In Proc. 9th ACM SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES 2008)*, p. 121–130, Tucson, AZ, USA, juin 2008. ACM Press.

[2] S. Blazy et X. Leroy : Mechanized semantics for the Clight subset of the C language. *J. Automated Reasoning*, 43(3):263–288, oct. 2009.

[3] S. Boulmé et G. Hamon : Certifying synchrony for free. *In* R. Nieuwenhuis et A. Voronkov, éds : *Proc. 8th Int. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001)*, vol. 2250 de *LNCS*, p. 495–506, Havana, Cuba, déc. 2001. Springer.

[4] S. Boulmé et G. Hamon : A clocked denotational semantics for Lucid-Synchrone in Coq. Rap. tech., LIP6, nov. 2001.

[5] T. Bourke, L. Brun, P.-É. Dagand, X. Leroy, M. Pouzet et L. Rieg : A formally verified compiler for Lustre. *In Proc. 2017 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, p. 586–601, Barcelona, Spain, juin 2017. ACM Press.

[6] T. Bourke, P.-É. Dagand, M. Pouzet et L. Rieg : Vérification de la génération modulaire du code impératif pour Lustre. *In* J. Signoles et S. Boldo, éds : *28^{ièmes} Journées Francophones des Langages Applicatifs (JFLA 2017)*, p. 165–179, Gourette, Pyrénées, France, jan. 2017.

[7] P. Caspi : Clocks in dataflow languages. *Theor. Comp. Sci.*, 94(1):125–140, mars 1992.

[8] J.-L. Colaço : Private communication, nov. 2017.

[9] J.-L. Colaço, B. Pagano et M. Pouzet : Scade 6 : A formal language for embedded critical software development. *In Proc. 11th Int. Symp. Theoretical Aspects of Software Engineering (TASE 2017)*, Nice, France, sept. 2017. IEEE Computer Society.

[10] J.-L. Colaço et M. Pouzet : Clocks as first class abstract types. *In* R. Alur et I. Lee, éds : *Proc. 3rd Int. Conf. on Embedded Software (EMSOFT 2003)*, vol. 2855 de *LNCS*, p. 134–155, Philadelphia, PA, USA, oct. 2003. Springer.

[11] N. Halbwachs, P. Caspi, P. Raymond et D. Pilaud : The synchronous dataflow programming language LUSTRE. *Proc. IEEE*, 79(9):1305–1320, sept. 1991.

[12] Programming languages—C. Standard, ISO/IEC, Geneva, Switzerland, déc. 1999.

[13] R. Kumar, M. O. Myreen, M. Norrish et S. Owens : CakeML : A verified implementation of ML. *In Proc. 41st ACM SIGPLAN-SIGACT Symp. Principles of Programming Languages (POPL 2014)*, p. 179–191, San Diego, CA, USA, jan. 2014. ACM Press.

[14] X. Leroy : Formal verification of a realistic compiler. *Comms. ACM*, 52(7):107–115, 2009.

[15] The Coq Development Team : *The Coq proof assistant reference manual*. Inria, 2018. v. 8.8.

Formalisation en Coq d'Algorithmes de Filtres Numériques

Diane Gallois-Wong[1,2*]

[1] Université Paris-Sud
[2] LRI, CNRS & Univ. Paris-Sud, Université Paris-Saclay, bâtiment 650, Université Paris-Sud, F-91405 Orsay Cedex, France

Abstract

Les algorithmes de filtres numériques sont un outil essentiel du traitement du signal et du contrôle-commande. Ils sont utilisés dans de nombreux domaines comme les télécommunications, l'automobile, la robotique, l'aéronautique etc. Comme certains de ces domaines sont critiques, nous souhaitons garantir formellement qu'ils se comportent bien numériquement malgré les erreurs d'arrondi inhérentes à leur implémentation en précision finie. Or il existe de nombreux algorithmes pour calculer un filtre numérique, que l'on appelle réalisations. En précision finie, des réalisations légèrement différentes, par exemple où seulement l'ordre des calculs est modifié, peuvent donner des résultats différents. La SIF (*Specialized Implicit Form*) est un formalisme très général, qui peut représenter toutes les réalisations en détaillant jusqu'à l'ordre des calculs. Cet article étend la formalisation en Coq des filtres numériques présentée dans [1] : il y ajoute la définition de la SIF, les traductions des réalisations de cette formalisation vers la SIF, et le théorème du filtre d'erreur associé à une SIF, qui décrit la propagation des erreurs de calcul au fil des itérations dans l'algorithme correspondant.

1 Introduction

Les filtres numériques sont utilisés dans de nombreux contextes, du traitement du signal au contrôle-commande, du lecteur MP3 au système de contrôle d'un avion. Ils permettent de transformer les signaux numériques, qui sont des suites de valeurs prises par des grandeurs physiques au cours du temps, en d'autres signaux numériques. Ils peuvent ainsi amplifier un signal sonore, ou encore produire la suite de commandes à fournir à l'avion à partir de toutes les données mesurées par ses capteurs. Implémenter un filtre numérique nécessite l'utilisation d'une arithmétique en précision finie, généralement virgule flottante ou virgule fixe, ce qui implique des erreurs d'arrondi. De plus, les algorithmes de filtres numériques sont fortement itératifs. D'une part, cela introduit un risque d'accumulation de nombreuses erreurs qui paraissent négligeables en une erreur critique. D'autre part, cela rend l'étude de ces erreurs d'arrondi complexe : chaque erreur peut influencer toutes les itérations futures.

Cet article s'inscrit dans la continuité de [1], une formalisation en Coq des filtres numériques qui contient les premières étapes vers une analyse formelle complète des erreurs d'arrondi. Cette formalisation définit les filtres linéaires invariants dans le temps, la famille usuelle de filtres que nous souhaitons étudier. Elle présente plusieurs algorithmes de filtres utilisés en pratique, appelés réalisations. Elle prouve deux théorèmes essentiels à cette analyse d'erreurs. Le théorème du filtre d'erreur décrit la propagation des erreurs au fil des calculs pour une réalisation appelée State-Space. Le théorème du Worst-Case Peak-Gain permet de borner le

*Ce travail a bénéficié du soutien financier de la "Fondation CFM pour la Recherche".

signal de sortie d'un filtre lorsqu'on connaît une borne sur l'entrée, et son application permet justement de borner l'erreur finale déterminée par le théorème précédent.

Le sujet principal de cet article est une formalisation[1] en Coq [2, 3] de la SIF (*Specialized Implicit Form*) [4], un formalisme qui permet de décrire tous les algorithmes de filtres numériques linéaires invariants dans le temps, en explicitant l'ordre dans lequel les calculs sont effectués. Nous définissons en Coq une traduction d'un State-Space vers une SIF, en prouvant que le filtre implémenté est conservé. Cela nous permet de décrire sous forme de SIF plusieurs réalisations usuelles ayant été traduites vers un State-Space dans [1]. Enfin, nous prouvons formellement le théorème du filtre d'erreur dans le cadre d'un filtre défini par une SIF. Comme dans [1], nous ne considérons pour l'instant pas de standard particulier pour les nombres en précision finie. Nous travaillons dans le cadre de n'importe quelle arithmétique en précision finie, avec des nombres réels modifiés par des termes d'erreur arbitraires. Choisir une de ces arithmétiques, et obtenir ainsi plus d'informations sur ces termes d'erreur, sera une étape future naturelle.

Notre formalisation en Coq étant un surensemble de celle de [1], elle utilise les mêmes bibliothèques et axiomes. Nous considérons les réels axiomatiques de la bibliothèque standard [5]. Nos vecteurs et matrices sont basés sur ceux de la bibliothèque d'analyse réelle Coquelicot [6]. Deux axiomes nous permettent de manipuler facilement fonctions et preuves, et sont généralement acceptés comme étant sûrs par la communauté. D'une part, `FunctionalExtensionality` affirme que deux fonctions prenant les mêmes valeurs sur toutes les entrées sont égales. D'autre part, `ProofIrrelevance` énonce que deux preuves d'une même propriété sont égales.

Différentes études formelles des erreurs d'arrondi dans les filtres numériques ont déjà été menées. Akbarpour et Tahar prouvent en HOL un résultat similaire au théorème du filtre d'erreur dans le cadre de plusieurs réalisations canoniques [7]. Akbarpour *et al.* comparent des implémentations de filtres en virgule flottante et en virgule fixe, sous l'hypothèse que les opérations arithmétiques ne peuvent pas produire d'*overflow* [8]. Park *et al.* bornent les erreurs d'arrondi pour une itération de la boucle principale d'une algorithme de filtre, mais n'étudient pas leur propagation [9]. Grâce à la SIF, notre formalisation couvre tous les algorithmes de filtres numériques et tient compte de l'ordre des calculs. Le théorème de filtre d'erreur que nous prouvons décrit la propagation des erreurs d'arrondi pour n'importe quelle arithmétique en précision finie.

La section 2 définit les filtres numériques et quelques réalisations usuelles. Son contenu a été intégralement formalisé en Coq dans [1]. Nous reprenons simplement ses éléments les plus pertinents pour les deux sections suivantes, dont les formalisations sont originales. La section 3 présente la SIF, son intérêt, sa définition en Coq et des correspondances prouvées entre celle-ci et le State-Space. La section 4 présente le théorème du filtre d'erreur dans le cas d'un filtre décrit par une SIF. Enfin, la section 5 conclut et propose quelques perspectives.

Notations. Nous écrirons les vecteurs en gras (et en minuscule), les matrices en gras et en majuscule : ainsi $\boldsymbol{a}$ représentera un vecteur et $\boldsymbol{A}$ une matrice, tandis que a sera un scalaire. Nous noterons $\boldsymbol{I}$ une matrice identité et $\boldsymbol{0}$ une matrice nulle de dimensions adaptées au contexte, ainsi que $\boldsymbol{I}_n$ la matrice identité de taille explicite $n \times n$.

2 Filtre numérique

Cette section présente les éléments de traitement du signal nécessaires pour comprendre le fonctionnement et l'intérêt de la SIF présentée en section 3. Elle introduit aussi les définitions Coq correspondantes les plus essentielles. Celles-ci sont directement reprises de [1], qui fournit

[1] Fichiers disponibles à www.lri.fr/~gallois/code/coq-filtresnum-JFLA19.tgz

des explications plus détaillées, notamment concernant les choix techniques.

2.1 Signal numérique

Un signal numérique représente l'évolution discrète d'une grandeur physique (vitesse, température, position etc.) au cours du temps. Un signal numérique est donc défini mathématiquement comme une suite de nombres réels, ou de vecteurs réels (dans le cas d'une position par exemple). On parle de signal réel ou signal vectoriel pour distinguer ces deux possibilités. Cette suite est indexée sur un intervalle de nombres entiers, fréquemment $\mathbb{Z}$, $\mathbb{N}$ ou de la forme $\{0, 1, ..., N\}$. Nous avons choisi de considérer n'importe quel indice dans $\mathbb{Z}$, mais avec la restriction que le signal doit prendre la valeur zéro (ou vecteur nul) pour les indices strictement négatifs : pour tout signal x et entier $k < 0$, $x(k) = 0$. C'est équivalent à définir les signaux seulement sur $\mathbb{N}$, mais présente des intérêts liés à l'initialisation et à la simplicité visuelle des énoncés de théorèmes, discutés dans [1].

Les définitions Coq sont reprises de [1]. Un signal réel se compose d'une fonction de $\mathbb{Z}$ dans $\mathbb{R}$ accompagnée de la preuve qu'elle vaut zéro pour $k < 0$. Un signal vectoriel est défini similairement. Les vecteurs `vect n` sont basés sur ceux de la bibliothèque Coquelicot mais leurs indices sont des entier relatifs (une valeur par défaut étant renvoyée pour un indice non compris entre 1 et la taille `n`).

```
Definition causal (x : Z → R) := (forall k : Z, (k < 0)%Z → x k = 0%R).
Record signal := { signal_val :> Z → R ; signal_prop : causal signal_val }.
Context { n : Z }.
Definition vect_causal (x : Z → vect n) := forall k, (k < 0)%Z → x k = vect_0.
Record vect_signal := { vect_signal_val :> Z → vect n ;
                        vect_signal_prop : vect_causal vect_signal_val }.
```

On définit les trois opérations élémentaires suivantes sur les signaux (présentées avec des signaux réels mais facilement étendues à des vecteurs, en notant que c reste un réel). Les définir en Coq nécessite de prouver que le résultat est encore un signal (nul pour les indices strictement négatifs).

- L'addition (terme à terme) : $y = x_1 + x_2$ signifie $\forall k,\ y(k) = x_1(k) + x_2(k)$.
- La multiplication par une constante $c \in \mathbb{R}$: $y = c\,x$ signifie $\forall k,\ y(k) = c\,x(k)$.
- Le retard, ou translation dans le temps : "y est le signal x retardé de $K \geq 0$ unités de temps" signifie $\forall k,\ y(k) = x(k - K)$.
  Noter que l'initialisation imposée sur nos signaux implique $\forall k < K,\ y(k) = 0$.
  En Coq, on choisit de renvoyer arbitrairement le signal identiquement nul lorsque $K < 0$, car il est plus facile de travailler avec des fonctions totales.

2.2 Filtres LTI

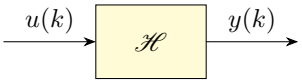


Figure 1: Filtre numérique.

Un filtre, généralement noté $\mathscr{H}$, est une fonction qui transforme un signal d'entrée u en un signal de sortie $y = \mathscr{H}\{u\}$. Ceci est représenté par la figure 1. La sortie $y(k)$ à l'instant k peut

dépendre de l'entrée correspondante $x(k)$, mais aussi de toutes les entrées antérieures (mais évidemment pas des entrées futures dans les filtres qui sont étudiés en pratique). Un filtre est appelé SISO (*Single-Input Single-Output*) s'il transforme des signaux réels en signaux réels, ou MIMO (*Multiple-Input Multiple-Output*) pour des signaux vectoriels.

```
Definition filter := signal -> signal.
Definition MIMO_filter {N_in N_out : Z} :=
  @vect_signal N_in -> @vect_signal N_out.
```

Nous nous intéressons ici à une famille de filtres très utilisée en pratique : les filtres linéaires invariants dans le temps (LTI pour *Linear Time Invariant*), qui satisfont les propriétés suivantes (avec des quantifications universelles sur les signaux qui peuvent être réels ou vectoriels, et sur la constante $c \in \mathbb{R}$) :

- Compatibilité avec l'addition : $\mathscr{H}\{u_1 + u_2\} = \mathscr{H}\{u_1\} + \mathscr{H}\{u_2\}$
- Compatibilité avec la multiplication par une constante : $\mathscr{H}\{c\,u\} = c\,\mathscr{H}\{u\}$
- Invariance dans le temps, i.e. si l'entrée est retardée de $K \geq 0$ unités de temps alors la sortie l'est aussi :
$$\forall k,\, u'(k) = u(k-K) \quad \Longrightarrow \quad \forall k,\, \mathscr{H}\{u'\}(k) = \mathscr{H}\{u\}(k-K)$$

2.3 Différentes réalisations

Il existe beaucoup de familles d'algorithmes de filtres, appelées réalisations. Un même filtre peut être exprimé par diverses réalisations, qui calculent toutes la même relation entrée-sortie lorsqu'on considère des nombres réels. Cependant, en précision finie, le choix de la réalisation est très important : en plus des différences de complexité en temps et en mémoire, apparaissent des différences de comportement numérique. Ainsi, pour un filtre donné, selon la réalisation et l'arithmétique en précision finie utilisées, la sortie calculée en pratique pourra plus ou moins s'écarter de la sortie du modèle en précision infinie.

La réalisation la plus classique est l'équation aux différences, aussi appelée forme directe I (DFI pour *Direct Form I*) :

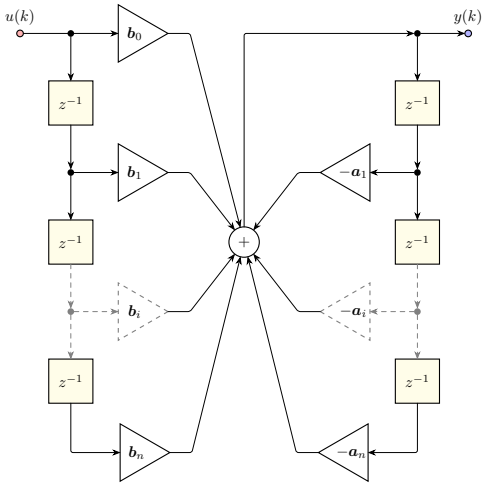
$$\forall k \geq 0,\; y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i) \tag{1}$$

Les $(a_i)_{1 \leq i \leq n}$ et $(b_i)_{0 \leq i \leq n}$ sont les coefficients du filtre, et n est appelé l'ordre du filtre. Ce sont ces mêmes coefficients qui apparaissent dans la fonction de transfert, un objet mathématique permettant de décrire les filtres dans le domaine fréquentiel. Mais cet article s'intéresse seulement au domaine temporel et aux algorithmes itératifs associés. Par exemple, la forme directe I permet naturellement de calculer chaque sortie $y(k)$ à condition d'avoir gardé en mémoire les entrées et sorties des n étapes précédentes.

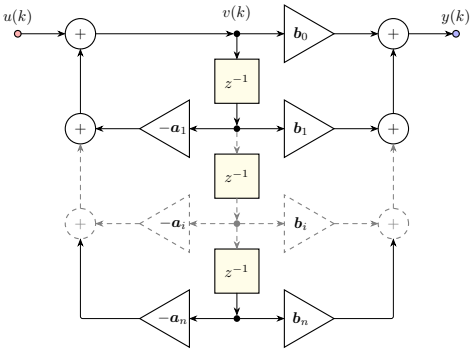
La forme directe II (DFII pour *Direct Form II*) utilise aussi les coefficients de la fonction de transfert :

$$\forall k \geq 0, \quad \begin{cases} e(k) = u(k) - \sum_{i=1}^{n} a_i e(k-i) \\ y(k) = \sum_{i=1}^{n} b_i e(k-i) \end{cases} \tag{2}$$

où e est un signal auxiliaire. À chaque étape, on peut calculer les valeurs de e et y en ayant retenu seulement les n valeurs précédentes de e, ce qui fait deux fois moins de mémoire nécessaire que pour la forme directe I.



(a) Forme directe I



(b) Forme directe II

Figure 2: Graphes à flot de données des formes directes I et II.

Les formes directes I et II peuvent être représentées par les graphes à flot de données de la figure 2. Les noeuds marqués d'un + sont bien sûr des additions, et les triangles des multiplications par une constante. Un z^{-1} représente un retard d'une unité de temps : à chaque étape, il produit la valeur qu'il a reçue à l'étape précédente. Le nombre de valeurs à garder en mémoire correspond au nombre de retards dans le graphe : $2n$ pour la forme directe I, n pour la forme directe II.

Le *State-Space* est une réalisation plus générale, couramment utilisée par la communauté de l'automatique. Contrairement aux formes directes précédentes, il permet de représenter un filtre MIMO. Le State-Space utilise un signal auxiliaire $\boldsymbol{x}$, appelé vecteur d'état, pour stocker toutes les informations dont on a besoin pour calculer la sortie courante, mais aussi les sorties futures. La relation entre l'entrée $\boldsymbol{u}$ et la sortie $\boldsymbol{y}$ est la suivante (les signaux étant bien sûr nuls pour $k < 0$) :

$$\begin{aligned} \boldsymbol{x}(k+1) &= \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) &= \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{aligned} \tag{3}$$

où les matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ et $\boldsymbol{D}$ sont des coefficients qui caractérisent le filtre.

N'importe quelle réalisation décrivant un filtre MIMO peut représenter un filtre SISO en considérant des vecteurs d'entrée $\boldsymbol{u}(k)$ et de sortie $\boldsymbol{y}(k)$ de taille 1. Dans le cas du State-Space, $\boldsymbol{B}$ devient alors un vecteur colonne, $\boldsymbol{C}$ un vecteur ligne, et $\boldsymbol{D}$ une matrice de taille 1×1, qu'on peut identifier à un scalaire. En revanche, le vecteur d'état $\boldsymbol{x}(k)$ peut quand même être de taille strictement supérieure à 1 : même pour un filtre SISO, on peut avoir besoin de stocker de multiples valeur d'une étape sur l'autre.

Le State-Space est très général : on peut transformer n'importe quelle réalisation en un State-Space qui représente le même filtre, c'est-à-dire la même relation entrée-sortie. Par exemple, la forme directe I est facilement transformée en un State-Space en sauvegardant les n sorties précédentes dans le vecteur d'état. Cependant, la traduction d'un algorithme vers un State-Space risque de modifier l'ordre des calculs, comme dans l'exemple de la section 3.2. Dans ce cas, même si les filtres théoriques représentés sont les mêmes, les sorties calculées en pra-

tique peuvent être différentes. La SIF, présentée en section 3, est une extension du State-Space qui résout ce problème car elle est capable de préserver l'ordre des calculs de n'importe quel algorithme.

Ces trois réalisations (forme directe I, forme directe II et State-Space) et leur formalisation en Coq, qui inclut des traductions des deux premières vers le State-Space, sont discutées avec plus de détails dans [1].

3 La SIF : utilité et formalisation

Il existe de nombreuses réalisations de filtres numériques en plus des formes directes I et II et du State-Space présentés en section 2.3 : formes directes transposées, opérateur ρ, structures LCW et LGS, décomposition en cascade ou en parallèle, etc. [10] Chacune présente ses propres avantages en termes de complexité en temps et en mémoire, compatibilité logicielle et matérielle, et comportement numérique pour une arithmétique en précision finie donnée. Cependant, formaliser l'analyse d'erreurs d'arrondi séparément pour chacune des nombreuses réalisations utilisées en pratique représenterait un travail très long. À la place, nous formalisons la SIF (*Specialized Implicit Form*) [4], une réalisation canonique qui permet de décrire n'importe quelle autre réalisation, en conservant l'ordre dans lequel les calculs sont effectués. Ce dernier point est très important, car l'ordre des opérations modifie les erreurs d'arrondi que nous souhaitons étudier. Ainsi, les propriétés prouvées pour la SIF, notamment le théorème du filtre d'erreur de la section 4, peuvent être appliquées à n'importe quelle autre réalisation, à condition de traduire formellement celle-ci en SIF.

En pratique, une SIF n'est pas implémentée directement sous forme de calcul matriciel. Cependant, on déduit facilement d'une SIF une implémentation sous forme de séquence d'affectations scalaires comportant principalement des sommes de produits.

3.1 Définition de la SIF et ordre des calculs

La SIF est une extension du State-Space. Une SIF est caractérisée par neuf matrices ($\boldsymbol{J}$, $\boldsymbol{K}$, $\boldsymbol{L}$, $\boldsymbol{M}$, $\boldsymbol{N}$, $\boldsymbol{P}$, $\boldsymbol{Q}$, $\boldsymbol{R}$, $\boldsymbol{S}$) au lieu de quatre pour le State-Space. De plus, la matrice carrée $\boldsymbol{J}$ doit toujours être triangulaire inférieure, avec des 1 sur toute la diagonale principale : par exemple, si elle est de taille 3×3, elle aura la forme $\begin{pmatrix} 1 & 0 & 0 \\ ? & 1 & 0 \\ ? & ? & 1 \end{pmatrix}$. Cette condition sera appelée *Jprop*, et implique notamment que $\boldsymbol{J}$ est inversible. Le filtre décrit par une SIF est donné par la relation entrée $\boldsymbol{u}$ - sortie $\boldsymbol{y}$ suivante :

$$\begin{aligned} \boldsymbol{J}\boldsymbol{t}(k+1) &= \boldsymbol{M}\boldsymbol{x}(k) + \boldsymbol{N}\boldsymbol{u}(k) \\ \boldsymbol{x}(k+1) &= \boldsymbol{K}\boldsymbol{t}(k+1) + \boldsymbol{P}\boldsymbol{x}(k) + \boldsymbol{Q}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) &= \boldsymbol{L}\boldsymbol{t}(k+1) + \boldsymbol{R}\boldsymbol{x}(k) + \boldsymbol{S}\boldsymbol{u}(k) \end{aligned} \tag{4}$$

Le signal auxiliaire $\boldsymbol{x}$ est toujours le vecteur d'état, mais il y a un deuxième signal auxiliaire $\boldsymbol{t}$, qui sert à calculer des valeurs intermédiaires et qu'on appellera vecteur intermédiaire. C'est la présence de $\boldsymbol{t}$ qui distingue la SIF du State-Space. En effet, si on enlevait toutes les apparitions de $\boldsymbol{t}$, c'est-à-dire la première ligne entière qui sert à calculer $\boldsymbol{t}(k+1)$ (on rappelle que $\boldsymbol{J}$ est inversible), le terme $\boldsymbol{K}\boldsymbol{t}(k+1)$ de la deuxième ligne et le terme $\boldsymbol{L}\boldsymbol{t}(k+1)$ de la troisième ligne, alors on retomberait exactement sur la relation (3) pour un State-Space de coefficients

$(\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$. Bien sûr, c'est l'ajout de $\boldsymbol{t}$ qui permet à la SIF, contrairement au State-Space, de décrire n'importe quel ordre pour les calculs d'un algorithme.

La condition *Jprop* peut sembler étrange, mais elle assure que pour un vecteur $\boldsymbol{w}$ donné, on peut calculer $\boldsymbol{v}$ tel que $\boldsymbol{J}\boldsymbol{v} = \boldsymbol{w}$ à l'aide d'un algorithme simple. Par exemple, considérons des vecteurs de taille 3 : $\begin{pmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ b & c & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$. En développant le produit matriciel et en isolant v_i sur la i-ème ligne, on obtient :

$$\begin{aligned} v_1 &= w_1, \\ v_2 &= w_2 - a v_1, \\ v_3 &= w_3 - b v_1 - c v_2. \end{aligned}$$

Le principe de la première ligne de (4) est donc que les composantes de $\boldsymbol{t}(k+1)$ doivent être calculées **dans l'ordre**, chacune pouvant dépendre des précédentes et de $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$. Puis, comme pour un State-Space, on calcule le vecteur d'état de l'étape suivante $\boldsymbol{x}(t+1)$ et la sortie $\boldsymbol{y}(k)$, à partir de $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$, mais aussi des valeurs intermédiaires de $\boldsymbol{t}(t+1)$ qui viennent d'être obtenues.

Les signaux $\boldsymbol{t}$ et $\boldsymbol{x}$ ont des rôles très différents, et ce n'est pas seulement à cause de la présence de $\boldsymbol{J}$ ni ce qu'elle impose sur le calcul de $\boldsymbol{t}$. Le vecteur d'état $\boldsymbol{x}$ sert à sauvegarder des valeurs d'une étape à la suivante : on calcule sa valeur suivante $\boldsymbol{x}(k+1)$, mais on n'utilise que sa valeur courante $\boldsymbol{x}(k)$ dans les membres droits des égalités. Au contraire, le vecteur intermédiaire $\boldsymbol{t}(t+1)$ est calculé et immédiatement utilisé au cours de la même étape, mais oublié entre chaque étape : on remarque que $\boldsymbol{t}(k)$ n'apparaît pas du tout. L'indice $k+1$ provient de l'écriture implicite de la relation (4) :

$$\begin{pmatrix} \boldsymbol{J} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{K} & \boldsymbol{I} & \boldsymbol{0} \\ -\boldsymbol{L} & \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{0} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k) \\ \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix} \tag{5}$$

Sous cette forme, on voit que $\boldsymbol{t}(k)$ ne peut pas être réutilisé à cause des blocs de matrices nulles dans la matrice du membre droit.

Pour formaliser la SIF en Coq, nous utilisons les matrices définies dans [1] à partir de celles de la bibliothèque Coquelicot. `mtx h w` désigne une matrice de dimensions `h`×`w`, dont les composantes ont un type qui dépend du contexte. Ici, ce type sera les réels munis d'une structure d'anneau, ce qu'on explicitera en écrivant `@mtx R_Ring h w`.

Nous définissons d'abord la propriété *Jprop* :

```
Definition J_prop (A : mtx n n) := (forall i, (1 <= i <= n)%Z -> A i i = 1)
                                /\ (forall i j, (1 <= i < j /\ j <= n)%Z -> A i j = 0).
```

Puis, une SIF est un record rassemblant les tailles des signaux vectoriels $\boldsymbol{x}$ et $\boldsymbol{t}$, les neuf matrices dont les tailles se déduisent de l'écriture de (4) (`N_in` étant la taille de l'entrée $\boldsymbol{u}$ et `N_out` celle de la sortie $\boldsymbol{y}$), et la preuve que $\boldsymbol{J}$ vérifie *Jprop* :

```
Record SIF := { SIF_N_x : Z ; (** taille de x **)
                SIF_N_t : Z ; (** taille de t **)
                SIF_J : @mtx R_Ring SIF_N_t SIF_N_t ;
                SIF_K : @mtx R_Ring SIF_N_x SIF_N_t ;
                SIF_L : @mtx R_Ring N_out SIF_N_t ;
```

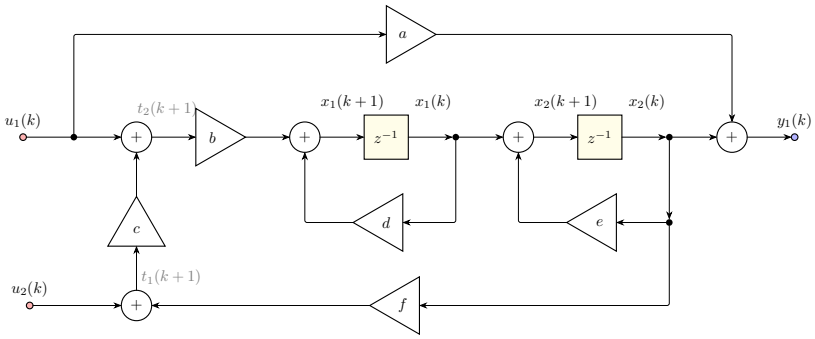


Figure 3: Graphe à flot de données de l'exemple.

```
SIF_M : @mtx R_Ring SIF_N_t SIF_N_x ;
SIF_N : @mtx R_Ring SIF_N_t N_in ;
SIF_P : @mtx R_Ring SIF_N_x SIF_N_x ;
SIF_Q : @mtx R_Ring SIF_N_x N_in ;
SIF_R : @mtx R_Ring N_out SIF_N_x ;
SIF_S : @mtx R_Ring N_out N_in ;
SIF_J_prop : J_prop SIF_J            }.
```

Afin de définir le filtre correspondant à une SIF donnée, nous écrivons d'abord une fonction `mtx_inv_J_prop : mtx ?n ?n -> mtx ?n ?n` qui calcule l'inverse d'une matrice qui vérifie *Jprop*. Puis, nous définissons
`SIF_u2x : forall sif : SIF, vect_signal -> vect_signal`, qui à une SIF $(\boldsymbol{J}, \boldsymbol{K}, ..., \boldsymbol{S})$ et un signal $\boldsymbol{u}$ associe le signal $\boldsymbol{x}$ défini par la relation de récurrence :
$\forall k,\ \boldsymbol{x}(k+1) = \boldsymbol{K}(\boldsymbol{J}^{-1}(\boldsymbol{M}\boldsymbol{x}(k) + \boldsymbol{N}\boldsymbol{u}(k))) + \boldsymbol{P}\boldsymbol{x}(k) + \boldsymbol{Q}\boldsymbol{u}(k)$. Enfin, nous construisons $\boldsymbol{y}$ comme dans (4). Nous associons ainsi à une SIF une fonction qui transforme un signal d'entrée $\boldsymbol{u}$ en un signal de sortie $\boldsymbol{y}$, c'est-à-dire un filtre : `MIMO_filter_from_SIF : SIF -> MIMO_filter`.

3.2 Exemple : détailler l'ordre des calculs à l'aide d'une SIF

Considérons le filtre $\mathscr{H}$ défini par le graphe à flot de données de la figure 3. Il prend une entrée $\boldsymbol{u} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$ de taille 2 et produit une sortie $\boldsymbol{y} = \begin{pmatrix} y_1 \end{pmatrix}$ de taille 1 selon les équations suivantes, où x_1 et x_2 sont des signaux auxiliaires (tous les signaux étant nuls pour $k < 0$) :

$$\begin{aligned} x_1(k+1) &= b(u_1(k) + c(u_2(k) + fx_2(k))) + dx_1(k), \\ x_2(k+1) &= x_1(k) + ex_2(k), \\ y(k) &= au_1(k) + x_2(k). \end{aligned} \tag{6}$$

Dans un premier temps, traduisons ce filtre vers un State-Space $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$. Cela permettra à la fois de s'habituer au raisonnement avec une construction plus simple que celle d'une SIF, et d'exposer les limites du State-Space. Les $x_1(k)$ et $x_2(k)$ sont exactement les informations que nous avons besoin de retenir d'une étape sur l'autre, donc le vecteur d'état sera $\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$.

D'après (3), nous devons avoir :

$$\begin{pmatrix} bu_1(k) + bcu_2(k) + bcfx_2(k) + dx_1(k) \\ x_1(k) + ex_2(k) \end{pmatrix} = \boldsymbol{x}(k+1) = \boldsymbol{A}\begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{B}\begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix},$$
$$\big(au_1(k) + x_2(k)\big) = \boldsymbol{y}(k) = \boldsymbol{C}\begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{D}\begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix},$$

donc $\boldsymbol{A} = \begin{pmatrix} d & bcf \\ 1 & e \end{pmatrix}$, $\boldsymbol{B} = \begin{pmatrix} b & bc \\ 0 & 0 \end{pmatrix}$, $\boldsymbol{C} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ et $\boldsymbol{D} = \begin{pmatrix} a \\ 0 \end{pmatrix}$.

Le problème, c'est que nous avons supposé que $b(u_1(k) + c(u_2(k) + fx_2(k))) + dx_1(k) = bu_1(k) + bcu_2(k) + bcfx_2(k) + dx_1(k)$, ce qui n'est pas toujours vrai en précision finie. Pour vraiment transcrire l'algorithme associé au graphe de la figure 3, nous avons besoin de prendre en compte l'ordre des opérations, et c'est exactement ce que fait la SIF.

Traduisons maintenant ce filtre vers une SIF. Pour pouvoir décrire correctement le calcul de $x_1(k+1)$, nous introduisons un signal de valeurs intermédiaires $\boldsymbol{t}$ de taille 2, tel que :

$$\begin{aligned} &t_1(k+1) = u_2(k) + fx_2(k), \\ &t_2(k+1) = u_1(k) + ct_1(k+1) \quad i.e. \quad -ct_1(k+1) + t_2(k+1) = u_1(k), \\ &x_1(k+1) = bt_2(k+1) + dx_1(k). \end{aligned}$$

Nous devons alors avoir, en appliquant (4) :

$$\boldsymbol{J}\begin{pmatrix} u_2(k) + fx_2(k) \\ u_1(k) + ct_1(k+1) \end{pmatrix} = \boldsymbol{M}\begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{N}\begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix},$$
$$\begin{pmatrix} bt_2(k+1) + dx_1(k) \\ x_1(k) + ex_2(k) \end{pmatrix} = \boldsymbol{K}\begin{pmatrix} t_1(k+1) \\ t_2(k+1) \end{pmatrix} + \boldsymbol{P}\begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{Q}\begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix},$$
$$\big(au_1(k) + x_2(k)\big) = \boldsymbol{L}\begin{pmatrix} t_1(k+1) \\ t_2(k+1) \end{pmatrix} + \boldsymbol{R}\begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{S}\begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix}.$$

Nous obtenons $\boldsymbol{J} = \begin{pmatrix} 1 & 0 \\ -c & 1 \end{pmatrix}$, $\boldsymbol{M} = \begin{pmatrix} 0 & f \\ 0 & 0 \end{pmatrix}$, $\boldsymbol{N} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, $\boldsymbol{K} = \begin{pmatrix} 0 & b \\ 0 & 0 \end{pmatrix}$, $\boldsymbol{P} = \begin{pmatrix} d & 0 \\ 1 & e \end{pmatrix}$, $\boldsymbol{Q} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$, $\boldsymbol{L} = \begin{pmatrix} 0 & 0 \end{pmatrix}$, $\boldsymbol{R} = \begin{pmatrix} 0 & 1 \end{pmatrix}$ et $\boldsymbol{S} = \begin{pmatrix} a & 0 \end{pmatrix}$. La SIF ainsi construite décrit exactement le graphe de la figure 3, y compris l'ordre des opérations. La traduction de n'importe quel graphe à flot de données vers une SIF est présentée dans [11]. Elle n'est cependant pas formalisée en Coq, car les graphes eux-mêmes seraient longs et complexes à définir.

3.3 Traduction d'un State-Space vers une SIF

Nous avons vu que la SIF peut représenter n'importe quel algorithme. Cependant, pour pouvoir analyser formellement l'impact des erreurs d'arrondi sur un algorithme donné, il nous faut une traduction certifiée de cet algorithme vers une SIF. Nous avons formalisé une telle traduction pour le State-Space présenté dans la section 2.3, qui est souvent utilisé par la communauté de l'automatique.

Soit $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ un State-Space, que nous souhaitons transformer en SIF. Nous avons vu en section 3.1 que si on enlève toutes les apparitions de $\boldsymbol{t}$ dans les équations (4), c'est-à-dire la première équation entière et un terme de chacune des deux autres, on retombe sur les équations

(3) correspondant à un State-Space $(\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$. Il suffit donc de définir :

$$\begin{aligned} \boldsymbol{K} &= \boldsymbol{0}, & \boldsymbol{L} &= \boldsymbol{0}, \\ \boldsymbol{P} &= \boldsymbol{A}, & \boldsymbol{Q} &= \boldsymbol{B}, \\ \boldsymbol{R} &= \boldsymbol{C}, & \boldsymbol{S} &= \boldsymbol{D}. \end{aligned} \tag{7}$$

Comme les termes en $\boldsymbol{t}$ ont été annulés en même temps que les matrices $\boldsymbol{K}$ et $\boldsymbol{L}$, on peut définir n'importe quelles valeurs pour $\boldsymbol{J}$, $\boldsymbol{M}$ et $\boldsymbol{N}$, par exemple l'identité pour $\boldsymbol{J}$ (qui doit tout de même vérifier la propriété *Jprop*) et encore des matrices nulles pour les deux autres. La SIF $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ ainsi obtenue décrit le même filtre que le State-Space $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$.

Cette méthode fonctionne en choisissant n'importe quelle taille pour $\boldsymbol{t}$. On peut faire plus simple et fixer la taille de $\boldsymbol{t}$ à zéro, ce qui évite d'avoir à préciser des valeurs pour $\boldsymbol{J}$, $\boldsymbol{K}$, $\boldsymbol{L}$, $\boldsymbol{M}$ et $\boldsymbol{N}$, qui ont chacune au moins une dimension nulle. Néanmoins, dans notre formalisation en Coq, nous avons choisi de donner la taille 1 à $\boldsymbol{t}$, au cas où nous aurions besoin plus tard de propriétés qui se comportent différemment sur des matrices ayant une dimension nulle.

En Coq, nous définissons la transformation ci-dessus, et nous prouvons que la SIF obtenue décrit bien le même filtre que le State-Space de départ :

```
Definition SIF_from_StateSpace : StateSpace -> SIF := ...
Theorem SIF_from_StateSpace_same_filter : forall stsp : StateSpace,
 MIMO_filter_from_SIF (SIF_from_StateSpace stsp) = MIMO_filter_from_StSp stsp.
```

En combinant la transformation ci-dessus et celles vers un State-Space formalisées dans [1], nous pouvons décrire explicitement sous forme de SIF deux autres réalisations usuelles présentées dans la section 2 : la forme directe I et la forme directe II. Cependant, l'ordre des opérations est celui imposé par le State-Space intermédiaire, et n'est pas forcément identique à celui de l'algorithme d'origine. Pour étudier les erreurs d'arrondi dans ces algorithmes exacts, il faudra les traduire indépendamment en SIF.

3.4 Traduction d'une SIF vers un State-Space

Intéressons-nous maintenant à la transformation inverse de celle vue précédemment : d'une SIF à un State-Space. Elle permet d'obtenir une représentation plus simple avec seulement quatre matrices au lieu de neuf. L'inconvénient est bien sûr que comme le State-Space est moins expressif, l'ordre des calculs risque d'être modifié. Cependant, ceci n'est pas un problème en précision infinie. Cette transformation reste donc intéressante pour étudier le filtre modèle exprimé avec des nombres réels.

Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF. Le filtre correspondant peut être également décrit par le State-Space $(\boldsymbol{A_Z}, \boldsymbol{B_Z}, \boldsymbol{C_Z}, \boldsymbol{D_Z})$ où :

$$\begin{aligned} \boldsymbol{A_Z} &= \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{P}, & \boldsymbol{B_Z} &= \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{Q}, \\ \boldsymbol{C_Z} &= \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{R}, & \boldsymbol{D_Z} &= \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{S}. \end{aligned} \tag{8}$$

On rappelle que $\boldsymbol{J}$ vérifie la propriété *Jprop*, ce qui implique qu'elle est inversible et que son inverse est facile à calculer. L'indice Z est couramment utilisé pour marquer le lien avec une SIF.

Nous définissons cette transformation en Coq, et prouvons que le State-Space construit décrit le même filtre que la SIF d'origine :

```
Definition StateSpace_from_SIF : SIF -> StateSpace := ...
Theorem StateSpace_SIF_same_filter : forall sif : SIF,
 MIMO_filter_from_StSp (StateSpace_from_SIF sif) = MIMO_filter_from_SIF sif.
```

4 Filtre d'erreur appliqué à la SIF

Nous nous intéressons ici à la propagation des erreurs d'arrondi, c'est-à-dire l'effet des erreurs locales dans chaque calcul sur les sorties du filtre. Cela s'applique à n'importe quelle arithmétique en précision finie. Cette propagation est décrite par le théorème du filtre d'erreur. Ce théorème est déjà présenté et prouvé dans [1] pour un filtre défini par un State-Space. Nous le prouvons ici dans le cas d'une SIF, qui est plus expressive.

Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF. On appelle $\mathscr{H}$ le filtre correspondant, calculé avec une précision infinie. Les signaux $\boldsymbol{t}$, $\boldsymbol{x}$ et $\boldsymbol{y}$ sont respectivement le vecteur intermédiaire, le vecteur d'état et la vecteur de sortie, obtenus en (4), toujours avec une précision infinie. Soit $\boldsymbol{t}^*$, $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$ ces mêmes éléments obtenus en appliquant (4) en précision finie, et $\mathscr{H}^*$ le filtre implémenté, qui à l'entrée $\boldsymbol{u}$ associe la sortie $\boldsymbol{y}^*$. L'erreur finale qui nous intéresse est alors $\boldsymbol{\Delta y}(k) = \boldsymbol{y}^*(k) - \boldsymbol{y}(k)$.

Dans le filtre implémenté $\mathscr{H}_\varepsilon$, les relations (4) ne sont respectées qu'aux erreurs d'arrondi près. On représente cela par l'ajout de termes d'erreurs $\boldsymbol{\varepsilon_t}(k)$, $\boldsymbol{\varepsilon_x}(k)$ et $\boldsymbol{\varepsilon_y}(k)$:

$$\begin{aligned} \boldsymbol{J}\boldsymbol{t}^*(k+1) &= \boldsymbol{M}\boldsymbol{x}^*(k) + \boldsymbol{N}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_t}(k) \\ \boldsymbol{x}^*(k+1) &= \boldsymbol{K}\boldsymbol{t}^*(k+1) + \boldsymbol{P}\boldsymbol{x}^*(k) + \boldsymbol{Q}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_x}(k) \\ \boldsymbol{y}^*(k) &= \boldsymbol{L}\boldsymbol{t}^*(k+1) + \boldsymbol{R}\boldsymbol{x}^*(k) + \boldsymbol{S}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_y}(k) \end{aligned} \tag{9}$$

On note $\boldsymbol{\varepsilon}(k)$ la concaténation de ces vecteurs d'erreur : $\boldsymbol{\varepsilon}(k) = \begin{pmatrix} \boldsymbol{\varepsilon_t}(k) \\ \boldsymbol{\varepsilon_x}(k) \\ \boldsymbol{\varepsilon_y}(k) \end{pmatrix}$. On a ainsi défini un signal vectoriel $\boldsymbol{\varepsilon}$ contenant toutes les erreurs locales.

Théorème (Filtre d'erreur). *Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF, décrivant un filtre modèle $\mathscr{H}$ en précision infinie. Soit $\mathscr{H}^*$ le filtre obtenu en appliquant l'algorithme correspondant en précision finie. On note $\boldsymbol{y}$ la sortie théorique de $\mathscr{H}$, $\boldsymbol{y}^*$ la sortie calculée de $\mathscr{H}^*$, et $\boldsymbol{\Delta y} = \boldsymbol{y}^* - \boldsymbol{y}$ l'erreur finale de l'implémentation par rapport au modèle. Enfin, soit $\boldsymbol{\varepsilon}$ le signal vectoriel des erreurs locales défini ci-dessus. Alors, le signal $\boldsymbol{\Delta y}$ s'obtient comme la sortie correspondant à l'entrée $\boldsymbol{\varepsilon}$ à travers le filtre d'erreur $\mathscr{H}_\varepsilon$, défini comme le filtre de State-Space $(\boldsymbol{A_Z}, \boldsymbol{B}_\varepsilon, \boldsymbol{C_Z}, \boldsymbol{D}_\varepsilon)$, où :*

$$\boldsymbol{B}_\varepsilon = (\boldsymbol{K}\ \boldsymbol{J}^{-1}\ \boldsymbol{I}_n\ \boldsymbol{0}), \qquad \boldsymbol{D}_\varepsilon = (\boldsymbol{L}\ \boldsymbol{J}^{-1}\ \boldsymbol{0}\ \boldsymbol{I}_p), \tag{10}$$

(on rappelle que $\boldsymbol{J}^{-1}$ est facile à calculer grâce à Jprop), et $\boldsymbol{A_Z}$ et $\boldsymbol{C_Z}$ sont les matrices du State-Space associé à la SIF $(\boldsymbol{J}, \boldsymbol{K}, ..., \boldsymbol{S})$, définies par l'équation (8) :

$$\boldsymbol{A_Z} = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{P}, \qquad \boldsymbol{C_Z} = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{R}. \tag{rappel de (8))}$$

De manière équivalente, comme illustré par la figure 4, la sortie $\boldsymbol{y}^*$ calculée en pratique est égale à la somme de la sortie $\boldsymbol{y}$ du filtre idéal $\mathscr{H}$ en précision infinie, et de la sortie $\boldsymbol{\Delta y}(k)$ du filtre d'erreur $\mathscr{H}_\varepsilon$ auquel on donne en entrée le signal d'erreurs locales $\boldsymbol{\varepsilon}$.

En Coq, nous considérons une SIF `sif`, un signal vectoriel d'entrée `u`, ainsi que des signaux vectoriels d'erreurs `err_t`, `err_x` et `err_y`, de tailles correspondant à celles de `sif`. Ces éléments sont bien sûr quantifiés universellement. Nous définissons récursivement le signal `x'`, puis le signal `y'`, qui correspondent respectivement à $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$ obtenus en appliquant (9). Nous calculons également `B_err` et `D_err` données par (10). Nous pouvons alors définir `stsp_err` comme le State-Space dont les matrices $\boldsymbol{B}$ et $\boldsymbol{D}$ sont `B_err` et `D_err`, et dont les matrices $\boldsymbol{A}$ et $\boldsymbol{C}$ sont celles du

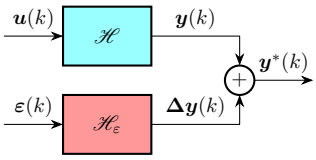


Figure 4: Décomposition du filtre implémenté en modèle $\mathscr{H}$ et filtre d'erreur $\mathscr{H}_\varepsilon$

State-Space `stsp_model` := `StateSpace_from_SIF sif`, obtenu en appliquant la transformation de la section 3.4 à `sif`. Enfin, nous prouvons que la sortie `y'` calculée en pratique est égale à la somme de la sortie du filtre théorique `h_model` décrit par `sif`, et de la sortie du filtre d'erreur `h_err` décrit par `stsp_err` pour l'entrée `err` qui est la concaténation des vecteurs d'erreurs locales `err_t`, `err_x` et `err_y` :

```
Definition h_model : MIMO_filter := MIMO_filter_from_SIF sif.
Definition h_err : MIMO_filter := MIMO_filter_from_StSp stsp_err.
Definition err := vect_signal_concat err_t (vect_signal_concat err_x err_y).
Theorem actual_is_model_plus_err :
    y' = vect_signal_plus (h_model u) (h_err err).
```

Ceci est exactement la formulation équivalente décrite par la figure 4 du théorème du filtre d'erreur.

La caractérisation de l'erreur finale $\boldsymbol{\Delta y}$ fournie par le théorème du filtre d'erreur va ensuite permettre de borner cette erreur. En effet, il existe un théorème qui permet de borner la sortie d'un filtre à partir d'une borne sur son entrée. Ce théorème, appelé le *Worst-Case Peak-Gain*, est prouvé en Coq dans [1]. Comme $\boldsymbol{\Delta y}$ est la sortie à travers le filtre d'erreur $\mathscr{H}_\varepsilon$ correspondant à l'entrée $\boldsymbol{\varepsilon}$, il suffit de borner le vecteur d'erreurs locales $\boldsymbol{\varepsilon}$. Or, les arithmétiques en précision finie usuelles garantissent l'arrondi correct des opérations, c'est-à-dire que le résultat d'une seule opération est la valeur représentable la plus proche de la valeur théorique. De plus, ces arithmétiques fournissent généralement une opération "somme de produits" qui permet de calculer une valeur de la forme $\Sigma_{1\leq i\leq N}\ a_i b_i$ sans arrondis intermédiaires, ce qui est très utile pour les produits vectoriels et matriciels. Les erreurs locales contenues dans $\boldsymbol{\varepsilon}$ proviennent donc chacune d'un très petit nombre d'opérations, et peuvent être bornées explicitement pour une arithmétique donnée. On connaît alors une borne sur le signal d'entrée fourni à $\mathscr{H}_\varepsilon$, de laquelle on déduit une borne sur sa sortie $\boldsymbol{\Delta y}$.

5 Conclusion

Nous avons formalisé en Coq la SIF, qui permet d'exprimer tous les algorithmes de filtres numériques en précisant dans quel ordre sont effectuées les opérations. Ce dernier point est particulièrement pertinent en précision finie : modifier l'ordre des calculs entraine des erreurs d'arrondi différentes, ce qui peut donner des résultats de qualité très variable. Nous avons défini en Coq la traduction du State-Space vers la SIF, et prouvé que le filtre qu'elles décrivent est conservé par ces traductions. Cela nous permet d'exprimer sous forme de SIF d'autre réalisations usuelles, les formes directes I et II, qui ont déjà été traduites en Coq vers un State-Space [1]. Enfin, nous avons prouvé formellement le théorème du filtre d'erreur dans le contexte de la SIF, établissant ainsi une caractérisation certifiée de la propagation des erreurs d'arrondi pour n'importe quel algorithme traduit en Coq vers une SIF.

Les preuves Coq ne présentent pas de difficulté particulière une fois que nous disposons des bons lemmes pour raisonner par récurrence forte sur $\mathbb{Z}$ (généralement initialisée pour tout $k < 0$) et surtout pour développer les calculs matriciels. La plupart de ces lemmes ont déjà été établis dans [1]. Notamment, nous utilisons les matrices introduites dans [1], basées sur celles de Coquelicot [6] mais dont les indices sont des entiers relatifs. Une valeur par défaut est renvoyée pour des indices excédant les bornes, ce qui inclut tous les entiers négatifs. Ces matrices sont utilisées à travers une interface qui est indépendante de leur construction. L'intérêt est que cela nous permet de changer facilement la bibliothèque sur laquelle nous nous appuyons. Notamment, nous envisageons d'utiliser les matrices de la bibliothèque Mathcomp [12], afin de s'appuyer sur ses théorèmes d'algèbre linéaire pour calculer précisément la borne donnée par le théorème du *Worst-Case Peak-Gain* mentionné à la fin de la section 4.

Une perspective naturelle est la prise en compte du choix d'arithmétique en précision finie. En effet, le théorème du filtre d'erreur s'applique à n'importe laquelle d'entre elles : il exprime l'erreur finale en fonction des erreurs locales à chaque étape, indépendamment des caractéristiques de ces dernières. Tenir compte des restrictions imposées par un standard donné sur les erreurs locales permettra donc d'obtenir plus d'informations sur l'erreur finale. Le cas de l'arithmétique en virgule flottante pourra s'appuyer sur la bibliothèque Flocq [13]. Cependant, les filtres numériques présents dans les systèmes embarqués utilisent surtout l'arithmétique en virgule fixe. Celle-ci est également formalisée dans Flocq, mais il manque à cette bibliothèque deux éléments pour pouvoir effectuer une étude approfondie des filtres en virgule fixe. D'une part, ces filtres utilisent généralement la représentation du complément à deux, tandis que Flocq a choisi une autre convention usuelle qui représente les nombres négatifs différemment. D'autre part, il existe différents comportements en cas d'*overflow* : par exemple, utiliser un modulo pour toujours avoir une valeur dans le bon intervalle, ou encore la règle de saturation, qui ramène un résultat trop grand à la valeur maximale autorisée. Dans certains cas, nous souhaiterons étudier formellement chacun de ces comportements utilisés en pratique. Dans d'autres cas, nous voudrons prouver que dans un algorithme donné, aucun *overflow* ne peut arriver. Ainsi, la première étape de l'étude approfondie des filtres des systèmes embarqués consistera à ajouter à Flocq le complément à deux et diverses options de gestion des *overflows*.

Dans un contexte industriel, comme les systèmes de contrôle en automobile ou en aéronautique, un filtre cible est spécifié, par exemple sous la forme d'une fonction de transfert souhaitée. Puis une réalisation est sélectionnée, c'est-à-dire un algorithme, ainsi qu'un support matériel, sur lequel il sera implémenté selon une arithmétique en précision finie bien précise. Nous décrivons alors l'algorithme sous forme de SIF et appliquons le théorème du filtre d'erreur, puis d'autres étapes qu'il reste encore à formaliser, notamment pour tenir compte de l'arithmétique choisie. L'objectif est d'obtenir une borne certifiée sur l'erreur finale causée par la précision finie sur la sortie du filtre. Une perspective complémentaire sera la génération automatique de code certifié correspondant à une SIF, donc à n'importe quel algorithme traduit formellement vers la SIF. L'étape suivante consistera à automatiser la sélection de l'algorithme, dans le but de minimiser l'erreur finale résultant des erreurs d'arrondi. On pourra s'appuyer sur des techniques de transformation automatique permettant d'améliorer la précision numérique d'un programme [14]. Cependant, ces techniques généralistes ne pourront peut-être pas explorer d'autres transformations du type changement de base d'une SIF [4], qui ont également pour but d'améliorer le comportement numérique en précision finie. La complémentarité des deux méthodes sera une piste d'étude intéressante.

Références

[1] Gallois-Wong, D., Boldo, S., Hilaire, T.: A coq formalization of digital filters. In Rabe, F., Farmer, W.M., Passmore, G.O., Youssef, A., eds.: Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings. Volume 11006 of Lecture Notes in Computer Science., Springer (2018) 87–103

[2] Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Texts in Theoretical Computer Science. Springer (2004)

[3] The Coq Development Team: The Coq Proof Assistant Reference Manual. (2017)

[4] Hilaire, T., Chevrel, P., Whidborne, J.: A unifying framework for finite wordlength realizations. IEEE Trans. on Circuits and Systems **8**(54) (August 2007) 1765–1774

[5] Mayero, M.: Formalisation et automatisation de preuves en analyses réelle et numérique. PhD thesis, Université Paris VI (décembre 2001)

[6] Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: A user-friendly library of real analysis for Coq. Mathematics in Computer Science **9**(1) (2015) 41–62

[7] Akbarpour, B., Tahar, S.: Error analysis of digital filters using hol theorem proving. Journal of Applied Logic **5**(4) (2007) 651–666 from the 4th International Workshop on Computational Models of Scientific Reasoning and Applications.

[8] Akbarpour, B., Tahar, S., Dekdouk, A.: Formalization of fixed-point arithmetic in HOL. Formal Methods in System Design **27**(1) (Sep 2005) 173–200

[9] Park, J., Pajic, M., Sokolsky, O., Lee, I. In: Automatic Verification of Finite Precision Implementations of Linear Controllers. Springer Berlin Heidelberg, Berlin, Heidelberg (2017) 153–169

[10] Lopez, B.: Implémentation optimale de filtres linéaires en arithmétique virgule fixe. PhD thesis, Université Pierre et Marie Curie, Sorbonne Université (Nov. 2014)

[11] Hilaire, T., Volkova, A., Ravoson, M.: Reliable fixed-point implementation of linear data-flows. In: Proc. IEEE Workshop on Signal Processing Systems (SiPS). (2016)

[12] Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: A Machine-Checked Proof of the Odd Order Theorem. In Blazy, S., Paulin, C., Pichardie, D., eds.: 4th Conference on Interactive Theorem Proving. Volume 7998 of LNCS., France, Springer (2013) 163–179

[13] Boldo, S., Melquiond, G.: Computer Arithmetic and Formal Proofs. ISTE Press - Elsevier (December 2017)

[14] Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. International Journal on Software Tools for Technology Transfer (September 2016)

Combinatoire Formelle avec Why3 et Coq

Alain Giorgetti[1], Catherine Dubois[2] et Rémi Lazarini

[1] Institut FEMTO-ST, Univ. of Bourgogne Franche-Comté, CNRS, Besançon, France
alain.giorgetti@femto-st.fr
[2] Samovar, ENSIIE, CNRS, Évry, France
catherine.dubois@ensiie.fr

Résumé

Nous présentons une approche formelle de la combinatoire, qui applique les méthodes du test unitaire et de la preuve formelle à des problèmes et algorithmes de combinatoire énumérative ou bijective. Cette combinatoire formelle utilise la plateforme de vérification déductive Why3, pour soumettre les démonstrations à des prouveurs automatiques ou les traduire dans le langage des assistants de preuve. Elle utilise aussi l'assistant de preuve Coq, dont le langage permet de formaliser des démonstrations et de les mener de manière interactive.

1 Introduction

La combinatoire est la branche des mathématiques qui étudie diverses configurations d'un ensemble fini d'objets, comme les permutations ou les mots. La combinatoire énumérative liste et dénombre ces structures combinatoires selon leur *taille*, qui est (le plus souvent) le nombre d'objets qui les composent. La combinatoire bijective établit des bijections structurelles non triviales entre diverses familles de structures combinatoires.

Nous désignons par *combinatoire formelle* l'application des méthodes formelles du génie logiciel à la recherche en combinatoire, en particulier pour prouver formellement des théorèmes de combinatoire énumérative ou bijective, ou des propriétés de programmes de comptage, d'énumération ou de transformation bijective de structures combinatoires.

D'importants travaux de formalisation des mathématiques ont produit des démonstrations formelles de théorèmes célèbres, comme le théorème des quatre couleurs en théorie des graphes [25] ou le théorème de Feit-Thompson en théorie des groupes [26]. Ces tours de force mobilisent de nombreux chercheurs pendant plusieurs années et portent sur des résultats majeurs, préalablement établis de manière informelle. De manière complémentaire, nous souhaitons encourager la pratique de la formalisation lors de recherches mathématiques plus modestes, dans des délais plus courts et sur des problèmes ouverts, afin que les efforts investis en formalisation et recherche de preuve produisent des gains de productivité de nouveaux théorèmes, et de confiance dans leur correction ainsi vérifiée. Des interactions et collaborations récentes avec des combinatoriciens nous ont convaincus que cette approche formelle pouvait bien convenir à certains sujets de recherche actuels sur les permutations ou les cartes combinatoires [2, 15, 24].

Nous étudions dans cet article l'adéquation des environnements Why3 [5] et Coq [10] à cette pratique de la combinatoire formelle. Why3 [5] est une plateforme de vérification déductive de programmes écrits en WhyML, un dialecte de ML avec certains traits fonctionnels, comme les types algébriques polymorphes ou le filtrage, mais aussi des traits impératifs, comme les boucles ou les enregistrements avec des champs mutables. Why3 traduit un programme et sa spécification en un ensemble de conditions logiques de vérification, qui peuvent être soumises à des prouveurs automatiques, tels que les solveurs modulo théorie (SMT), comme Alt-Ergo [1], CVC4 [12] ou Z3 [13]. Si ces prouveurs échouent, les conditions de vérification peuvent être prouvées interactivement au sein d'un assistant de preuve comme Coq.

Dans un assistant de preuve, il est précieux de se convaincre qu'un lemme est correct, avant d'investir du temps dans sa démonstration interactive. Dans ce but, nous avons complété Coq avec un outil de test automatique de conjectures, qui utilise des programmes d'énumération de données de test écrits en OCaml (partie 3). Afin d'accroître la confiance dans ces programmes, nous proposons de les produire par extraction de programmes WhyML dont la correction et la complétude sont démontrées formellement avec Why3 et Coq (partie 2). Lorsque ces programmes énumèrent des structures combinatoires, cette preuve formelle est une activité de combinatoire formelle, qui complète, voire remplace, une preuve informelle d'un ouvrage de combinatoire énumérative. Notre troisième contribution est une étude de cas originale, composée des deux premières étapes d'une démonstration formelle d'une bijection entre une structure combinatoire particulière – les nombres factoriels – et les permutations de même taille (partie 4). La première étape est la certification d'un programme d'énumération de tableaux factoriels, utile pour tester toute conjecture sur les nombres factoriels. La deuxième étape est une preuve formelle de correction de la traduction des tableaux factoriels en permutations.

Le code présenté est disponible dans la version 2.2 de notre outil CUT[1] (*Coq Unit Testing*) de test de conjectures Coq.

2 Preuve de programmes d'énumération

Cette partie présente une méthode de spécification, programmation et preuve formelle de programmes d'énumération avec Why3. Elle généralise une étude précédente limitée aux permutations [23] et adapte à Why3 une méthode de preuve de programmes d'énumération écrits en C et spécifiés en ACSL [22].

Nous étudions la famille des programmes d'énumération de structures combinatoires appelés *générateurs lexicographiques*, car ils énumèrent toutes les structures de même taille dans l'ordre lexicographique (spécifié en WhyML dans la partie 2.1). Par souci de simplicité, notre exposé est limité aux structures combinatoires représentées par un tableau à valeurs dans un intervalle borné d'entiers. La partie 2.2 introduit l'exemple fil rouge des tableaux à valeurs dans un intervalle de la forme $[0..b-1]$. Les parties 2.3 et 2.4 décrivent respectivement une formalisation des générateurs lexicographiques et de leurs propriétés en WhyML. La partie 2.5 décrit une bibliothèque de générateurs certifiés.

2.1 Ordre lexicographique

Soit A un ensemble muni d'un ordre strict $<$ (relation binaire irréflexive et transitive). On appelle *ordre lexicographique (strict, induit par $<$)* la relation binaire sur les tableaux à valeurs dans A, notée $\prec$, telle que, pour tout entier $n \geq 0$ et pour tous les tableaux a et b de longueur n dont les éléments sont dans A, on a $a \prec b$ si et seulement s'il existe un indice i $(0 \leq i \leq n-1)$ tel que $a[j] = b[j]$ pour tout j entre 0 et $i-1$ inclus et $a[i] < b[i]$.

Le listing 1 reproduit un extrait du module `Lex` que nous proposons pour spécifier formellement l'ordre lexicographique en WhyML. Par souci de simplicité, l'ensemble A est ici l'ensemble des entiers relatifs du type `int` de Why3, et l'ordre $<$ sur A est le prédicat

```
val predicate (<) int int : bool
```

défini en WhyML de telle sorte que sa fermeture réflexive

```
let predicate (≤) (x y : int) = x < y || x = y
```

[1] http://members.femto-st.fr/alain-giorgetti/en/coq-unit-testing.

```
module Lex
  use import int.Int
  use import array.Array
  use import array.ArrayEq

  predicate lt_lex_sub (a1 a2: array int) (l u: int) = ∃ i:int. l ≤ i < u
    ∧ array_eq_sub a1 a2 l i ∧ a1[i] < a2[i]
  predicate lt_lex (a1 a2: array int) = a1.length = a2.length ∧ lt_lex_sub a1 a2 0 a1.length

  predicate le_lex_sub (a1 a2: array int) (l u: int) = lt_lex_sub a1 a2 l u ∨
    array_eq_sub a1 a2 l u
  predicate le_lex (a1 a2: array int) = a1.length = a2.length ∧ le_lex_sub a1 a2 0 a1.length
end
```

Listing 1: Spécification de l'ordre lexicographique.

soit un ordre total. Le prédicat `array_eq_sub` est défini dans la bibliothèque standard de Why3 par

```
  predicate array_eq_sub (a1 a2: array 'a) (l u: int) = a1.length = a2.length ∧
    0 ≤ l ≤ a1.length ∧ 0 ≤ u ≤ a1.length ∧ map_eq_sub a1.elts a2.elts l u
```

avec

```
  predicate map_eq_sub (a1 a2 : map int 'a) (l u : int) =
    ∀ i:int. l ≤ i < u → a1[i] = a2[i]
```

L'expression `(array_eq_sub a1 a2 l u)` formalise que les tableaux a_1 et a_2 sont de même longueur et que les sous-tableaux $a_1[l..u-1]$ et $a_2[l..u-1]$ sont égaux. Le prédicat `lt_lex` formalise sur les tableaux d'entiers Why3 l'ordre lexicographique strict $\prec$ induit par l'ordre strict `<` sur les entiers Why3. Ce prédicat est généralisé à tout sous-tableau par le prédicat `lt_lex_sub`. Les prédicats `le_lex` et `le_lex_sub` sont leur fermeture réflexive respective.

Totalité. L'ordre lexicographique $\prec$ est total si l'ordre $<$ sur A est total, ce qui est le cas pour l'ordre `<` sur `int` de Why3. Nous démontrons formellement cette propriété de totalité, importante car elle permet de démontrer la complétude de certains générateurs lexicographiques (la propriété de complétude d'un générateur est définie plus loin, dans la partie 2.4).

La *totalité* de l'ordre lexicographique $\prec$ est la propriété $(\forall a\ b.\ a \not\prec b \Rightarrow b \preceq a)$ selon laquelle, si le tableau a n'est pas inférieur au tableau b, alors il lui est supérieur ou égal. Nous la généralisons à tout sous-tableau entre les indices l et $u-1$ inclus, par le lemme

```
  lemma total_order: ∀ a b: array int, l u: int.
    0 ≤ l < u ≤ a.length = b.length ∧ not (lt_lex_sub b a l u) → le_lex_sub a b l u
```

en WhyML. Ce lemme est démontré automatiquement, après l'ajout de la *lemma function*

```
  let rec lemma not_array_eq_sub (a b: array int) (l u: int)
    requires { 0 ≤ l < u ≤ a.length = b.length }
    requires { not (array_eq_sub a b l u) }
    variant  { u - l }
    ensures  { ∃ i:int. l ≤ i < u ∧ array_eq_sub a b l i ∧ a[i] ≠ b[i] }
  = if a[l] = b[l] then not_array_eq_sub a b (l+1) u
```

Une *lemma function* est un lemme dont l'énoncé est un contrat de fonction et la preuve est un programme *pur*, qui termine toujours et sans effets de bord. Ce programme démontre le lemme selon lequel la précondition de la fonction implique sa postcondition. Cette fonctionnalité de Why3 est présentée et utilisée, par exemple, dans la partie 3.1 de [9]. Nous avons ici une preuve

du lemme

```
lemma not_array_eq_sub: ∀ a b: array int, l u: int.
  0 ≤ l < u ≤ a.length = b.length ∧ not (array_eq_sub a b l u) →
  ∃ i:int. l ≤ i < u ∧ array_eq_sub a b l i ∧ a[i] ≠ b[i]
```

qui stipule que si les tableaux $a[l..u-1]$ et $b[l..u-1]$ sont différents alors il existe un indice i tel que les sous-tableaux $a[l..i-1]$ et $b[l..i-1]$ sont égaux et $a[i] \neq b[i]$. La fonction récursive `not_array_eq_sub` démontre ce lemme par récurrence sur la longueur des sous-tableaux considérés. Ce lemme et sa démonstration pallient le manque de support du raisonnement par induction dans les solveurs SMT.

2.2 Exemple des tableaux bornés

Comme exemple fil rouge de structure combinatoire à énumérer, considérons la famille des tableaux à valeurs dans l'intervalle initial d'entiers naturels $[0..b-1]$, pour une certaine *borne* $b > 0$. Cette famille des tableaux *bornés* (ou *initiaux*) – baptisée `barray` en anglais, pour "*bounded array*" – est une structure combinatoire, car il y a un nombre fini de tableaux initiaux de taille n et de borne b. Son prédicat caractéristique peut être formalisé par le prédicat

```
predicate is_barray (a:array int) (b:int) =
  ∀ i:int. 0 ≤ i < a.length → 0 ≤ a[i] < b
```

2.3 Fonctions d'énumération

Les programmes d'énumération modifient un état, appelé *curseur*, dont le type est défini par

```
type cursor = {
  current: array int;
  mutable new: bool; }
```

en WhyML. Le champ `current` stocke la dernière structure énumérée. Ici c'est un tableau d'entiers mutable, mais d'autres types peuvent être utilisés. Le champ booléen `new` prend la valeur `false` si et seulement si la structure stockée dans le champ `current` a déjà été énumérée. Ce curseur est une variante du curseur proposé par Filliâtre et Pereira pour l'itération [19].

Considérons une famille `X` de structures combinatoires, caractérisée par un prédicat

```
predicate is_X (a: array int) (...) = ...
```

où `(...)` représente d'éventuels paramètres supplémentaires. Un *générateur lexicographique* pour cette famille `X` est composé de deux fonctions : une fonction d'initialisation et une fonction de passage au suivant.

La fonction d'*initialisation*

```
create_cursor (n: int) : cursor
```

retourne un curseur initialisé avec le plus petit tableau de taille n dans cette famille `X`, selon l'ordre lexicographique $\prec$.

La fonction de *passage au suivant*

```
next (c : cursor) : unit
```

remplace la structure a stockée dans le curseur c par la structure de la famille `X` qui la suit immédiatement selon l'ordre lexicographique, si la structure a n'est pas maximale. Sinon, la fonction donne la valeur `false` au champ booléen `new` du curseur c.

Ce couple de fonctions est appelé générateur *à petits pas*, pour le distinguer du *générateur à grands pas* du listing 2, qui réalise l'énumération exhaustive à l'aide de ces deux fonctions. Ce programme produit une à une toutes les structures combinatoires d'une taille donnée n (dans

```
let big_step_gen (f: list int → 'a) (n: int) : unit =
  requires { n ≥ 0 }
  diverges
  let c = create_cursor n in
  while c.new do
    let a = c.current in
    f (to_list a 0 a.length);
    next c
  done
```

Listing 2: Générateur à grands pas.

le curseur c) et leur applique le même traitement (une fonction f donnée). Techniquement, ces tableaux d'entiers sont d'abord convertis en listes par la fonction `to_list`, car Why3 ne permet pas l'application directe d'une fonction sur un tableau d'entiers.

Ce générateur à grands pas est un cas particulier d'itérateur d'ordre supérieur, écrit ici dans le style impératif. Les qualificatifs "à grands pas" et "à petits pas" ont été introduits par Filliâtre et Pereira [18] pour distinguer les itérateurs qui contrôlent l'itération de ceux qui laissent le contrôle à l'utilisateur. Nos générateurs sont des cas particuliers de ces deux sortes d'itérateurs. Ils parcourent des collections qui ne sont pas stockées en mémoire mais dont chaque donnée est produite en place et utilisée à la volée.

```
let create_cursor (n b: int) : cursor
  requires { n ≥ 0 }
  requires { b > 0 }
= let a = make n 0 in
  { current = a; new = true; bound = b }

let next (c: cursor) : unit
= let a = c.current in
  let n = a.length in
  let b = c.bound in
  let r = ref (n-1) in
  while !r ≥ 0 && a[!r] = b-1 do
    r := !r - 1
  done;
  if (!r < 0) then
    c.new ← false
  else begin
    a[!r] ← a[!r] + 1;
    for i = !r+1 to n-1 do
      a[i] ← 0
    done;
    c.new ← true
  end
```

Listing 3: Énumération des tableaux bornés en WhyML.

Exemple. Le listing 3 présente des fonctions d'énumération pour les tableaux bornés. Pour cette structure, le curseur est complété avec un champ

```
bound: int;
```

qui stocke la borne des tableaux. La fonction `create_cursor` construit le plus petit tableau borné a de longueur $n \geq 0$, tel que $a[i] = 0$ pour tout indice i, sous la (pré)condition que la borne b soit strictement positive, car sinon aucun tableau de cette nature n'existe. La première boucle de la fonction `next` détermine l'indice $!r$ le plus élevé tel que $a[!r] < b - 1$ puisse être incrémenté. Si $!r$ atteint -1, le tableau a est maximal et l'énumération est terminée. Sinon, $a[!r]$ est incrémenté et la deuxième boucle remplace par 0 la valeur de chaque élément du sous-tableau $a[!r + 1..n - 1]$.

2.4 Preuve de propriétés des générateurs lexicographiques

Nous souhaitons spécifier et démontrer formellement que chaque générateur lexicographique satisfait les propriétés suivantes de correction et de complétude.

Correction. La *correction* est la propriété que chaque structure générée de la famille `X` satisfait son prédicat caractéristique `is_X`. Nous introduisons le prédicat

```
predicate sound (c: cursor) = is_X c.current ...
```

et nous spécifions la correction par les postconditions respectives

```
ensures { sound result }
```

et

```
ensures { sound c }
```

des fonctions `create_cursor` et `next`, en ajoutant la précondition

```
requires { sound c }
```

à la fonction `next`. Ceci exige que la fonction d'initialisation construise une structure de la famille `X` et que la fonction de passage au suivant transforme toute structure de la famille `X` en une structure de la même famille.

Complétude. La propriété de *complétude* exige que le générateur à grands pas produise toutes les structures d'une taille donnée. Puisque les structures sont générées selon l'ordre lexicographique total $\prec$, la complétude du générateur à grands pas résulte de la conjonction des trois propriétés suivantes :

1. la fonction d'initialisation génère la plus petite structure selon $\prec$ (propriété *min_lex*),
2. la fonction de passage au suivant transforme chaque structure en la structure immédiatement supérieure selon $\prec$ (propriété d'*incrémentation*), et
3. le générateur à grands pas termine lorsque le curseur `c` contient la plus grande structure selon $\prec$ (propriété *max_lex*).

Nous détaillons la formalisation de ces trois propriétés :

1. La propriété *min_lex*, qui impose que la première structure générée soit la plus petite structure selon l'ordre lexicographique, est spécifiée par la postcondition

   ```
   ensures { min_lex result.current }
   ```

   dans le contrat de la fonction `create_cursor`, avec le prédicat

   ```
   predicate min_lex (a: array int) = ∀ b: array int.
     a.length = b.length ∧ is_X b → le_lex_sub a b 0 a.length
   ```

2. La propriété d'*incrémentation* spécifie qu'une structure a_2 générée par la fonction `next` à partir d'une structure a_1 est toujours la plus petite structure strictement supérieure à la structure a_1, selon l'ordre lexicographique. Autrement dit, il n'existe aucune structure a_3 telle que $a_1 < a_3 < a_2$. Cette propriété est spécifiée par la postcondition

   ```
   ensures { c.new → inc (old c.current) c.current }
   ```

   de la fonction `next`, avec

   ```
   predicate inc (a1 a2: array int) = lt_lex a1 a2 ∧ ∀ a3: array int.
     lt_lex a1 a3 ∧ is_X a3 → le_lex a2 a3
   ```

3. La propriété *max_lex*, qui exige que la dernière structure générée soit la plus grande selon l'ordre lexicographique, est spécifiée par la postcondition

```
ensures { not c.new → max_lex result.current }
```

dans le contrat de la fonction `next`, avec le prédicat

```
predicate max_lex (a: array int) = ∀ b: array int.
  a.length = b.length ∧ is_X b → le_lex_sub b a 0 a.length
```

2.5 Catalogue de générateurs certifiés

Genestier, Petiot et Giorgetti [21, 22] ont développé en C et spécifié en ACSL une bibliothèque de programmes d'énumération de structures combinatoires, nommée enum. La version la plus récente de la bibliothèque enum contient 24 générateurs et un mécanisme de construction de générateurs par filtrage des données produites par un générateur plus général.

Les structures de données générées sont les tableaux bornés, les parties d'un ensemble à n éléments, les tableaux bornés triés, les combinaisons de p éléments parmi n, les injections de $[0..n-1]$ dans $[0..k]$ (pour $n \leq k+1$), les surjections de $[0..n-1]$ dans $[0..k]$ (pour $n \geq k+1$), les permutations, les involutions, les dérangements (permutations sans point fixe), les fonctions à croissance limitée, les involutions sans point fixe, les permutations connectées, les involutions connectées quelconques et les involutions connectées sans point fixe. Toutes ces structures sont définies dans la thèse de Genestier [20]. Pour certaines d'entre elles, plusieurs générateurs sont proposés et vérifiés.

Les propriétés vérifiées avec la plateforme Frama-C et son greffon de vérification déductive WP sont la correction et le *progrès*, qui est la propriété que chaque structure générée par la fonction de passage au suivant est supérieure à la précédente, selon l'ordre lexicographique. Cependant, la propriété de complétude n'a pas pu être démontrée formellement dans cet environnement. Cette propriété plus complexe inclut une quantification sur les tableaux. Pour faciliter sa démonstration, Giorgetti et Lazarini ont adapté l'un des programmes – énumérant des permutations – au langage WhyML et à sa structure de tableaux mutables [23]. Ces travaux antérieurs sont respectivement disponibles dans les versions 1.0[2] et 1.1[3] de la bibliothèque enum.

Depuis, deux autres générateurs ont été certifiés avec Why3 : le générateur de tableaux bornés présenté dans la partie 2.2 et le générateur de tableaux factoriels présenté dans la partie 4.2. Ce catalogue de générateurs certifiés avec Why3 est disponible en *open source*, pour être étendu et dépasser la portée de la version 1.0 de la bibliothèque enum en C/ACSL. Les propriétés des générateurs de tableaux bornés, factoriels et de permutations ont été prouvées avec les solveurs Alt-Ergo 1.30, CVC4 1.5 et Z3 4.7.1 – parfois en augmentant la limite de temps par preuve de 5 à 10 secondes – sauf la complétude du générateur de permutations, qui a nécessité deux preuves interactives, réalisées avec Coq 8.7.0, pour un total de 177 lignes de preuve.

3 Test de conjectures

Cette partie présente une application majeure des programmes d'énumération de structures combinatoires, qui justifie leur développement et leur certification avec Why3. Cette application est le test des propriétés qui ne sont pas démontrées automatiquement dans un délai

[2] Archive `enum-1.0.tar.gz` sur la page http://members.femto-st.fr/alain-giorgetti/en.
[3] https://github.com/alaingiorgetti/enum.

raisonnable. Les démontrer interactivement est une activité délicate, qui requiert une connaissance approfondie des tactiques de preuve et de la bibliothèque de définitions et théorèmes de l'assistant de preuve utilisé. Tester une propriété avant d'en commencer une démonstration interactive permet de se convaincre qu'elle est correcte, ou sinon de gagner un temps précieux en détectant rapidement une faille de raisonnement ou une erreur de formulation.

Le *test de propriété* (PBT, pour *Property-Based Testing*) est la recherche d'un contre-exemple pour une propriété d'un programme en cours de vérification. Il est populaire pour les langages fonctionnels, notamment avec l'outil QuickCheck [8] dans Haskell. Le PBT a également été adapté à des assistants de preuve, comme Isabelle [4], Agda [17], PVS [32], FoCaLiZe [7] et plus récemment Coq [33], avec l'outil de test aléatoire QuickChick. Dans ce cadre des assistants de preuve, nous parlerons de *test de conjecture*.

Parmi les méthodes de test automatique, le test exhaustif borné (BET, pour *Bounded Exhaustive Testing*) d'une fonction ou propriété consiste à générer toutes ses données d'entrée jusqu'à une certaine taille. Le BET est particulièrement bien adapté aux structures combinatoires, car ses contre-exemples sont toujours de taille minimale (ce qui facilite le déboguage), sa couverture est bien identifiée (puisqu'il constitue une preuve par énumération de tous les cas jusqu'à la borne de test), et une donnée de petite taille suffit souvent pour révéler une erreur [27]. Ainsi, le BET est complémentaire du test aléatoire, plus adapté aux domaines de données de plus grande taille.

Certains outils de test généraux sont capables de générer des données ou de dériver des générateurs à partir de la définition de ces données, selon diverses techniques, détaillées dans les introductions de travaux récents sur ce sujet [29, 11]. Par exemple, Dubois, Giorgetti et Genestier ont complété QuickChick avec une première approche de test exhaustif borné fondée sur des générateurs dérivés de définitions des données en programmation logique (Prolog) [15].

Cependant, pour certaines structures de données, ces outils peuvent être trop lents, échouer dans la dérivation d'un générateur ou dériver des générateurs peu efficaces. Il devient alors pertinent de concevoir un *générateur dédié* à chaque structure, voire de le certifier pour l'intégrer avec confiance dans un outil de test. Par exemple, Bowles et Caminati ont vérifié un algorithme d'énumération de structures d'événements [6]. Dubois et Giorgetti ont développé l'outil CUT, qui ajoute à Coq les commandes SmallCheck et SmallCheckWhy3 de test exhaustif borné avec des programmes d'énumération dédiés, respectivement définis dans les langages de Coq et de Why3 [14].

Cette partie décrit l'intégration dans la commande SmallCheckWhy3 d'un programme d'énumération en WhyML, éventuellement certifié avec Why3, comme présenté dans la partie 2. Nous détaillons ainsi une présentation précédente [14] en l'illustrant avec un exemple original.

Considérons une conjecture Coq de la forme

$$\forall \mathsf{x} : \mathsf{T},\ \mathsf{precondition}\ \mathsf{x} \rightarrow \mathsf{conclusion}\ \mathsf{x}, \tag{1}$$

où precondition et conclusion sont des prédicats logiques de type T → Prop. Pour tester cette conjecture, nous devons disposer d'une fonction booléenne – nommée conclusionb – exécutable et sémantiquement équivalente au prédicat logique conclusion.

L'exécution de la commande Coq

SmallCheckWhy3 *path* (it_X *size*) conclusionb.

réalise le BET de cette conjecture à l'aide d'un générateur OCaml de termes de type T satisfaisant la propriété precondition, si *path* est un chemin relatif vers le dossier du code source OCaml du générateur, si it_X est un itérateur Coq qui interface ce générateur OCaml avec Coq, et si *size* est la taille des données à générer. Nous nous intéressons ici au cas où le générateur OCaml est obtenu par extraction d'un générateur écrit en WhyML et certifié avec Why3.

Exemple. Illustrons cette commande Coq avec l'exemple de la structure de liste d'entiers naturels bornés (version Coq des tableaux bornés WhyML de la partie 2.2). Dans ce cas, T dans la conjecture (1) est le type list nat des listes d'entiers naturels en Coq. La famille des listes Coq dont les éléments sont des entiers naturels strictement inférieurs à une borne donnée b est caractérisée par le prédicat logique is_blist, qui est défini inductivement par

```
Inductive is_blist (b : nat) : list nat → Prop :=
| Blist_nil : is_blist b nil
| Blist_cons : ∀ v l, v < b → is_blist b l → is_blist b (v :: l).
```

et qui constitue la precondition de la conjecture (1). Ce prédicat n'est pas exécutable, mais nous pouvons prouver son équivalence avec la fonction booléenne is_blistb suivante :

```
Fixpoint is_blistb (b : nat) (l : list nat) :=
  match l with
    nil    ⇒ true
  | v :: l' ⇒ (ltb v b) && (is_blistb b l')
  end.
Lemma is_blist_dec : ∀ b l, (is_blistb b l = true ↔ is_blist b l).
```

Nous voulons tester la conjecture Coq

```
∀ (l : list nat) (b : nat), is_blist b l → is_blist (rev b l).
```

qui affirme que le miroir de toute liste l bornée par b est une liste bornée par la même valeur b.

Supposons que le générateur Why3 de tableaux bornés de la partie 2.3 soit défini dans le module Enum d'un fichier nommé Barray.mlw. Son extraction en OCaml par Why3 est un fichier Barray__Enum.ml, accompagné de fichiers auxiliaires. Soit `../barray` le dossier d'extraction.

```
Parameter cursor : Type.
Parameter create : nat → nat → cursor.
Parameter next_list : cursor → (option (list nat)) * cursor.
Parameter hasnext_list : cursor → bool.

Definition it_blist (n b : nat) := {|
 st_t := cursor;
 start := create n b;
 next := next_list;
 hasnext := hasnext_list |}.

Extract Constant cursor ⇒ "Barray__Enum.cursor".
Extract Constant create ⇒ "fun n → fun b →
  Barray__Enum.create_cursor (Why3extract.Why3__BigInt.of_int n)
  (Why3extract.Why3__BigInt.of_int b)".
Extract Constant next_list ⇒ "(fun c →
  Barray__Enum.next c;
  let a = c.current in (Some (to_list a), c))".
Extract Constant hasnext_list ⇒ "Barray__Enum.has_next".
```

Listing 4: Itérateur Coq sur les listes bornées.

Le listing 4 reproduit le code Coq d'un itérateur it_blist sur les listes bornées, interfacé avec le générateur OCaml de tableaux bornés. Les commandes d'extraction peuvent utiliser

des fonctions Why3extract.* issues de l'extraction Why3. Ici, elles utilisent aussi une fonction `to_list` de conversion de tableaux en listes. De plus, une fonction booléenne

```
let has_next (c: cursor) : bool = c.new
```

qui détermine si un curseur a un successeur doit être ajoutée au générateur WhyML.

Alors, la commande Coq

```
SmallCheckWhy3 "../barray" ( it_blist  7 4) (fun l ⇒ is_blistb  4 (rev  l )).
```

teste la conjecture avec toutes les listes de longueur 7 dont les éléments sont des entiers dans l'intervalle [0..3]. Techniquement, la commande SmallCheckWhy3 est ajoutée à l'outil de test aléatoire QuickChick, afin de ré-utiliser son mécanisme d'extraction OCaml pour exécuter les cas de test.

4 Étude de cas

Cette partie illustre la démarche de combinatoire formelle par une étude de cas originale, qui utilise Why3 pour démontrer un théorème lié aux permutations.

En 1888, C.-A. Laisant [28] introduit un système de numération, appelé *numération factorielle*, qui distingue $n!$ nombres à n chiffres, pour représenter les $n!$ permutations de n éléments. Un *code de permutation* est une transformation bijective de ces nombres en permutations de taille n. Divers codes de permutation sont connus et largement étudiés en combinatoire [30, 16, 31, 34, 3]. Nous souhaitons étudier formellement ces codes, c'est-à-dire les programmer et démontrer formellement certaines de leurs propriétés. La présente étude initie ce programme de recherche, en proposant un premier code de permutation programmé en WhyML, et une démonstration formelle qu'il ne produit que des permutations.

Les nombres de Laisant sont parfois dits *subexcédants*, avec les deux orthographes *subexceedant* [16] et *subexcedant* [34] en anglais. Nous les appelons plus simplement *(nombres) factoriels* en français, et *factorial numbers* ou *factorials* en anglais.

Les étapes de cette étude sont : une spécification formelle de la propriété des nombres factoriels (partie 4.1), une implémentation d'un générateur lexicographique de factoriels en WhyML, une spécification et une preuve formelle de ses propriétés de correction et de complétude (partie 4.2), une implémentation du code de permutation en WhyML, en tant que transformation de tableaux de même taille (partie 4.3), et enfin une spécification et une preuve formelle de la propriété que ce code transforme tout factoriel en permutation (partie 4.4).

4.1 Factoriels

Un mot $f_{n-1} \dots f_0$ de longueur n est un *(nombre) factoriel* si et seulement si sa lettre f_i est un entier naturel inférieur ou égal à i, pour $0 \leq i \leq n-1$. Par exemple, 23110 est un factoriel, tandis que 3020 n'est pas un factoriel, car son deuxième chiffre le plus à droite est 2, qui n'est pas inférieur ou égal à 1. En particulier, le dernier chiffre f_0 d'un factoriel est toujours zéro. Le nombre factoriel $f_{n-1} \dots f_0$ est la *numération factorielle* de l'entier naturel $\Sigma_{i=0}^{n-1} f_i \; i!$ et cette numération est une bijection entre nombres factoriels de longueur n et nombres entiers dans l'intervalle $[0..(n!-1)]$.

Nous représentons ces mots par des tableaux mutables WhyML et nous spécifions leur prédicat caractéristique de *(tableau) factoriel* avec les définitions suivantes :

```
predicate is_fact_sub (a:array int) (l u:int) = ∀ i:int.
  l ≤ i < u → 0 ≤ a[i] ≤ i
```

qui spécifie que le sous-tableau $a[l..u-1]$ est factoriel, et

```
predicate is_fact (a:array int) = is_fact_sub a 0 a.length
```

qui spécifie que le tableau a en paramètre représente un nombre factoriel.

Le chiffre f_i du nombre factoriel $f_{n-1} \dots f_0$ est stocké dans la case $a[i]$ du tableau factoriel correspondant. Remarquons cependant que, selon les usages d'écriture des nombres et des tableaux, le nombre factoriel $f_{n-1} \dots f_0$ s'écrit avec le chiffre $f_0 = 0$ de poids faible (égal à 0!) à droite, tandis que son tableau factoriel

0	1	2	...	$n-2$	$n-1$
f_0	f_1	f_2	...	f_{n-2}	f_{n-1}

est présenté dans le sens inverse, avec f_0 comme chiffre le plus à gauche. Pour éviter cette confusion, nous ne traitons plus de nombres factoriels, mais uniquement de tableaux factoriels, en présentant leur contenu $f_0 \dots f_{n-1}$ dans l'ordre croissant des indices.

4.2 Générateur lexicographique de tableaux factoriels

Un générateur lexicographique de tableaux factoriels est reproduit dans le listing 5. La fonction `create_cursor` construit le plus petit tableau factoriel a, tel que $a[i] = 0$ pour tout indice i. Le contrat de la fonction `next` spécifie ses propriétés de *correction*, d'*incrémentation* et de dernier tableau généré maximal. Les variables auxiliaires a et n désignent respectivement la structure courante et sa taille. Le label `'L` permet de décrire par `(at a 'L)` le tableau `a` tel qu'il est au niveau du label. La boucle `while` implémente la recherche de l'indice $!r$ de révision du tableau. Cette boucle est spécifiée par deux invariants, qui bornent les valeurs de $!r$ et assurent que le sous-tableau $a[r+1..n-1]$ est maximal, avec le prédicat `is_id_sub` défini par

```
predicate is_id_sub (a:array int) (l u:int) = ∀ i:int. l ≤ i < u → a[i] = i
```

Un variant permet de démontrer la terminaison de cette boucle. Si l'indice de révision $!r$ vaut -1, le tableau est maximal et la génération est terminée. Sinon, on a $a[!r] < !r$, ce qui permet d'incrémenter $a[!r]$, puis de remplir le sous-tableau $a[r+1..n-1]$ avec des 0, grâce à la boucle `for`. Ses trois premiers invariants bornent son indice i et assurent que la valeur de $a[!r]$ n'est pas modifiée par la boucle et que le sous-tableau $a[!r+1..i-1]$ ne contient que des 0, avec le prédicat `cte_sub` défini par

```
predicate cte_sub (a:array int) (l u:int) (b:int) = ∀ i:int. l ≤ i < u → a[i] = b
```

Avec la définition

```
predicate lt_lex_at (a1 a2: array int) (i:int) = 0 ≤ i < a1.length = a2.length
  ∧ array_eq_sub a1 a2 0 i ∧ a1[i] < a2[i]
```

le dernier invariant assure que le tableau courant est strictement supérieur au tableau en entrée, tout en précisant que le plus petit indice auquel les valeurs des deux tableaux diffèrent est $!r$. Cette précision permet aux solveurs SMT de choisir $!r$ pour démontrer la propriété d'incrémentation.

4.3 Code de permutation en WhyML

Soit n un entier naturel et f un tableau factoriel de longueur n. L'algorithme de la figure (1a) transforme le factoriel f en une permutation p, également représentée dans un tableau de longueur n. Dans cet algorithme, $t(x)$ désigne le $(x+1)$-ème élément de la liste ou du tableau t. La figure (1b) illustre l'algorithme avec les valeurs successives des variables, pour $n = 5$ et $f = f(0)\ f(1)\ f(2)\ f(3)\ f(4) = 0\ 1\ 0\ 3\ 1$.

L'idée de l'algorithme est d'utiliser l'élément $f(i)$ du tableau f pour choisir la valeur $p(i)$ de la permutation p dans la liste l des valeurs non encore choisies. Le tableau f étant factoriel,

```
let create_cursor (n: int) : cursor
  requires { n ≥ 0 }
  ensures { sound result }
  ensures { min_lex result.current }
= let a = make n 0 in
  { current = a; new = true }

let next (c: cursor) : unit
  requires { sound c }
  ensures { sound c }
  ensures { c.new →
     inc (old c.current) c.current }
  ensures { not c.new →
     max_lex c.current }
= let a = c.current in
  let n = a.length in
  'L: let r = ref (n-1) in
  while !r ≥ 0 && a[!r] = !r do
    invariant { -1 ≤ !r ≤ n-1 }
    invariant { is_id_sub a (!r+1) n }
    variant  { !r + 1 }
    r := !r - 1
  done;
  if (!r < 0) then (* Last array reached. *)
    c.new ← false
  else begin
    a[!r] ← a[!r] + 1;
    for i = !r+1 to n-1 do
      invariant { !r+1 ≤ i ≤ n }
      invariant { (at a 'L)[!r]+1 = a[!r] }
      invariant { cte_sub a (!r+1) i 0 }
      invariant { lt_lex_at (at a 'L) a !r }
      a[i] ← 0
    done;
    c.new ← true
  end
```

Listing 5: Énumération des factoriels en WhyML.

1. Initialiser une variable auxiliaire l avec la liste $[0, 1, \ldots, n-1]$ des n premiers entiers naturels dans l'ordre croissant.
2. Pour i décroissant de $n-1$ à 0, calculer $j = f(i)$, $k = l(j)$, puis stocker k dans $p(i)$ et supprimer k dans l.

(a) Algorithme du code de permutation

				p				
l	i	j	k	0	1	2	3	4
$[0,1,2,3,4]$	4	1	1	_	_	_	_	1
$[0,2,3,4]$	3	3	4	_	_	_	4	1
$[0,2,3]$	2	0	0	_	_	0	4	1
$[2,3]$	1	1	3	_	3	0	4	1
$[2]$	0	0	2	2	3	0	4	1

(b) Exemple d'exécution

Figure 1: Algorithme et exemple de trace d'exécution

son élément $f(i)$ désigne toujours un élément dans la liste, même si la taille de la liste est décrémentée à chaque itération, par suppression de la valeur choisie.

Le listing 6 présente une fonction `code` qui implémente l'algorithme en WhyML, avec l'aide de quelques fonctions auxiliaires. Pour $n \geq 0$, l'expression `(id_aux n i)` construit la liste $[i, i+1, \ldots, i+n-1]$ (vide si $n \leq 0$). Comme le spécifie le contrat formel de la fonction `id`, l'expression `(id n)` construit la liste $[0, 1, \ldots, n-1]$ pour tout entier $n \geq 0$. Sa deuxième postcondition est exprimée à l'aide de la fonction `nth` d'accès au `n`-ème élément d'une liste `l`, spécifiée par

```
function nth (n: int) (l: list 'a) : 'a
axiom nth_cons_0: ∀ x:'a, r:list 'a. nth 0 (Cons x r) = x
axiom nth_cons_n: ∀ x:'a, r:list 'a, n:int. n > 0 → nth n (Cons x r) = nth (n-1) r
```

dans la bibliothèque standard de Why3. Après ajout d'un contrat à la fonction `id_aux` – non détaillé car élémentaire – le contrat de la fonction `id` est démontré avec Alt-Ergo 1.30.

Comme le spécifie le contrat de la fonction `rm_nth`, l'exécution de l'expression `(rm_nth x l)`

```
let rec id_aux (n i: int) : list int =
  if n ≤ 0 then Nil else
    Cons i (id_aux (n-1) (i+1))

predicate is_id (l:list int) = ∀ i:int.
  0 ≤ i < L.length l → nth i l = i

let id (n: int) : list int
  requires { 0 ≤ n }
  ensures { L.length result = n }
  ensures { is_id result }
= id_aux n 0

let rec rm_nth (x:int)
  (l:ref (list int)) : int
  requires { 0 ≤ x < L.length !l }
  ensures { result = nth x (old !l) }
  ensures { ∀ i. 0 ≤ i < x
    → nth i !l = nth i (old !l) }
  ensures { ∀ i. x < i < L.length !l
    → nth i !l = nth (i+1) (old !l) }
  ensures { length !l =
              length (old !l) - 1 }
  variant { L.length !l }
= match (!l) with
  | Nil → l := Nil; 0
  | Cons y m →
      if x = 0 then begin
        l := m; y
      end else begin
        let r = ref m in
        let z = rm_nth (x-1) r in
        l := Cons y !r;
        z
      end
  end

let code (f: array int) : array int
  requires { is_fact f }
  ensures { is_permut result }
= let n = f.length in
  let p = make n 0 in
  let l = ref (id n) in
  for i = n-1 downto 0 do
    let j = f[i] in
    let k = rm_nth j l in
    p[i] ← k
  done;
  p
```

Listing 6: Code de permutation en WhyML.

a deux effets : (1) elle retourne le $(x+1)$-ème élément de la liste $!l$ référencée par l, avant modification de cette liste par la fonction (première postcondition), et (2) elle supprime cet élément dans cette liste $!l$ (deuxième, troisième et quatrième postconditions). La fonction n'assure ses postconditions que lorsque la position x existe dans la liste, comme spécifié dans sa précondition. Sous cette condition, la longueur de la liste $!l$ est un variant de la fonction `rm_nth`, puisqu'elle est décrémentée à chaque appel récursif de la fonction.

4.4 Preuve de correction du code de permutation

Nous démontrons formellement que la fonction `code` respecte son contrat, qui formalise la conjecture suivante : Si le paramètre f est un tableau factoriel (précondition), la fonction retourne (le tableau de valeurs d')une permutation (postcondition).

Les permutations sont caractérisées par le prédicat

```
predicate is_permut (a:array int) = (range a) ∧ (injective a)
```

où `(range a)` spécifie que le tableau `a` est à valeurs dans $[0..$`a.length` $-1]$ et `(injective a)` spécifie l'injectivité de la fonction représentée par le tableau `a`.

Les invariants suivants pour la boucle de la fonction `code` permettent la vérification automatique de sa correction :

```
1  invariant { i = L.length !l - 1 }
2  invariant { is_blist !l n }
3  invariant { injlist !l }
```

```
4  invariant { range_sub p (i+1) n n }
5  invariant { inj_sub p (i+1) n }
6  invariant { disj_sub p (i+1) n !l }
```

Le premier invariant déclare que la longueur de la liste $!l$ est toujours égale à $i+1$. C'est le cas puisque chaque itération décrémente i et enlève un élément dans cette liste. Nous expliquons les autres invariants d'abord globalement, puis un par un, en justifiant leur pertinence et leur nécessité.

En général, les invariants de boucle sont introduits pour automatiser la démonstration qu'une certaine propriété est satisfaite après exécution de la boucle. Il s'agit ici de démontrer la propriété (`is_permut p`) que le tableau p est une permutation. C'est le cas puisque, d'une part, chaque itération préserve la propriété que le sous-tableau $p[i+1..n-1]$ est une permutation, et, d'autre part, l'indice i vaut -1 après la dernière itération de la boucle.

La caractérisation `is_permut` d'une permutation comme endofonction sur $[0..n-1]$ (`range p`) et injection (`injective p`) permet de raisonner indépendamment sur ces deux propriétés, d'où les invariants 4 et 5, qui affirment respectivement que le sous-tableau $p[i+1..n-1]$ est à valeurs dans $[0..n-1]$ et injectif (sans doubles).

L'invariant 4 ($p[i+1..n-1]$ à valeurs dans $[0..n-1]$) est préservé car la valeur k stockée dans $p[i]$ à chaque itération vient d'une liste d'entiers dans $[0..n-1]$, comme spécifié par l'invariant 2 (le prédicat `is_blist` n'est pas détaillé car c'est seulement l'analogue pour une liste du prédicat `is_barray` présenté dans la partie 2.2).

Enfin, l'invariant 5 ($p[i+1..n-1]$ sans doubles) découle de l'invariant 6, selon lequel les contenus du tableau $p[i+1..n-1]$ et de la liste $!l$ sont disjoints, et de l'invariant 3, selon lequel la liste $!l$ est sans doubles.

Ainsi, ces annotations de boucle ne sont que la formalisation de propriétés intermédiaires qu'on établirait naturellement lors d'un raisonnement informel sur la correction de ce programme. Why3 et le prouveur Alt-Ergo suffisent pour vérifier toutes ces annotations et donc prouver formellement la conjecture initiale.

5 Conclusion

Le domaine de la combinatoire énumérative est encore peu perméable aux pratiques du génie logiciel. La recherche en combinatoire énumérative est essentiellement une activité de réflexion humaine et de démonstration sur papier, demandant beaucoup d'expérience et d'intuition. Nous pensons que cette activité délicate peut être facilitée par une formalisation machine des problèmes dès le début de leur étude, accompagnée d'une utilisation systématique d'outils logiciels et de méthodes formelles de spécification et de vérification.

Nous désignons par "combinatoire formelle" l'étude de la combinatoire avec un ordinateur. Ceci inclut l'usage très répandu d'un système de calcul formel, mais aussi la programmation d'algorithmes de génération exhaustive bornée ou aléatoire d'objets combinatoires, et l'utilisation d'un prouveur automatique ou d'un assistant de preuve pour démontrer avec un ordinateur des théorèmes combinatoires ou des propriétés de programmes combinatoires.

Cet article contribue à la popularisation de la combinatoire formelle, en présentant des outils et activités de preuve formelle de programmes d'énumération et de test de conjectures combinatoires. Plusieurs générateurs exhaustifs bornés ont été implémentés et spécifiés en WhyML. Leur correction et leur complétude ont été prouvées, avec des prouveurs automatiques et un assistant de preuve. Nous avons ainsi complété des vérifications précédentes [22, 23] avec des preuves de complétude. Dans une étude de cas originale, nous avons démontré une propriété d'un code de permutation, qui est une bijection entre nombres factoriels et permutations de

même taille. C'est un travail préliminaire pour la démonstration d'autres propriétés de ce code et l'étude formelle de divers codes de permutation.

Remerciements. Merci aux relecteurs anonymes pour leurs remarques et suggestions.

Références

[1] The Alt-Ergo SMT solver. http://alt-ergo.lri.fr, 2018.

[2] J.-L. Baril, R. Genestier, A. Giorgetti, and A. Petrossian. Rooted planar maps modulo some patterns. *Discrete Mathematics*, 339(4):1199–1205, 2016.

[3] J.-L. Baril and V. Vajnovszki. A permutation code preserving a double Eulerian bistatistic. *Discrete Applied Mathematics*, 224:9–15, 2017.

[4] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *SEFM'04*, pages 230–239. IEEE Computer Society, 2004.

[5] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 Platform*, 2018. http://why3.lri.fr/manual.pdf.

[6] J. Bowles and M. B. Caminati. A verified algorithm enumerating event structures. In *CICM'17*, volume 10383 of *LNCS*, pages 239–254. Springer, 2017.

[7] M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in FOCALTEST. In *Proceedings of the 5th International Conference on Software and Data Technologies - Volume 2: ICSOFT*, pages 82–91. SciTePress, 2010.

[8] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, volume 35 of *SIGPLAN Not.*, pages 268–279. ACM, New York, 2000.

[9] M. Clochard, J.-C. Filliâtre, C. Marché, and A. Paskevich. Formalizing semantics with an automatic program verifier. In *VSTTE'14*, volume 8471 of *LNCS*, pages 37–51. Springer, 2014.

[10] The Coq Development Team. The Coq Proof Assistant Reference Manual. http://coq.inria.fr/, 2017. Version 8.7.

[11] S. Cruanes. Satisfiability modulo bounded checking. In *Automated Deduction – CADE 26*, volume 10395 of *LNCS*, pages 114–129. Springer, 2017.

[12] The CVC4 SMT solver. http://cvc4.cs.stanford.edu/web/, 2018.

[13] L. De Moura and N. Björner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[14] C. Dubois and A. Giorgetti. Tests and proofs for custom data generators. *Formal Aspects of Computing*, 2018. https://doi.org/10.1007/s00165-018-0459-1.

[15] C. Dubois, A. Giorgetti, and R. Genestier. Tests and proofs for enumerative combinatorics. In *TAP'16*, volume 6792 of *LNCS*, pages 57–75. Springer International Publishing, 2016.

[16] D. Dumont and G. Viennot. A combinatorial interpretation of the Seidel generation of Genocchi numbers. In *Combinatorial Mathematics, Optimal Designs and Their Applications*, volume 6 of *Annals of Discrete Mathematics*, pages 77 – 87. Elsevier, 1980.

[17] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In *TPHOLs'03*, volume 2758 of *LNCS*, pages 188–203. Springer, 2003.

[18] J.-C. Filliâtre and M. Pereira. Itérer avec confiance. In *JFLA'16*, 2016. https://hal.inria.fr/hal-01240891.

[19] J.-C. Filliâtre and M. Pereira. A modular way to reason about iteration. In *NFM'16*, volume 9690 of *LNCS*, pages 322–336. Springer, 2016.

[20] R. Genestier. *Vérification formelle de programmes de génération de données structurées*. PhD thesis, Université de Franche-Comté, 2016.

[21] R. Genestier, A. Giorgetti, and G. Petiot. Gagnez sur tous les tableaux. In *JFLA'15*, 2015. https://hal.inria.fr/hal-01099135.

[22] R. Genestier, A. Giorgetti, and G. Petiot. Sequential generation of structured arrays and its deductive verification. In *TAP'15*, volume 9154 of *LNCS*, pages 109–128. Springer, 2015.

[23] A. Giorgetti and R. Lazarini. Preuve de programmes d'énumération avec Why3. In *AFADL'18*, pages 14–19, 2018. http://afadl2018.ls2n.fr/wp-content/uploads/sites/38/2018/06/AFADL_Procs_2018.pdf.

[24] A. Giorgetti and V. Senni. Specification and Validation of Algorithms Generating Planar Lehman Words. GASCom'12, https://hal.inria.fr/hal-00753008, 2012.

[25] G. Gonthier. The four colour theorem: Engineering of a formal proof. In *ASCM'07*, volume 5081 of *LNCS (LNAI)*, pages 333–333. Springer, 2008.

[26] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *ITP'13*, pages 163–179. Springer, 2013.

[27] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng.*, 22(7):484–495, 1996.

[28] C.-A. Laisant. Sur la numération factorielle, application aux permutations. In *Bulletin de la S. M. F., tome 16*, pages 176–183, 1888. http://www.numdam.org/item?id=BSMF_1888__16__176_0.

[29] L. Lampropoulos, D. Gallois-Wong, C. Hriţcu, J. Hughes, B. C. Pierce, and L. Xia. Beginner's luck: a language for property-based generators. In *POPL'17*, pages 114–129. ACM, 2017.

[30] D. H. Lehmer. Teaching combinatorial tricks to a computer. In *Proc. Sympos. Appl. Math. Combinatorial Analysis*, volume 10, pages 179–193. Amer. Math. Soc., 1960.

[31] R. Mantaci and F. Rakotondrajao. A permutations representation that knows what "Eulerian" means. *Discrete Mathematics and Theoretical Computer Science*, 4(2):101–108, 2001.

[32] S. Owre. Random testing in PVS. Workshop on Automated Formal Methods (AFM), 2006. http://fm.csl.sri.com/AFM06/papers/5-Owre.pdf.

[33] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In *ITP'15*, volume 9236 of *LNCS*, pages 325–343. Springer, 2015.

[34] V. Vajnovszki. Lehmer code transforms and Mahonian statistics on permutations. *Discrete Mathematics*, 313(5):581 – 589, 2013.

Axiomes de Continuité en Géométrie Neutre : une Étude Mécanisée en Coq

Charly Gries, Julien Narboux, and Pierre Boutry

Laboratoire ICube, UMR 7357 CNRS, Université de Strasbourg
charly.gries@etu.unistra.fr, {narboux, boutry}@unistra.fr

Abstract

Dans cet article, nous présentons les résultats sur la continuité que nous avons obtenus dans le cadre du projet *GeoCoq*, qui constitue une formalisation en Coq de la géométrie : nous avons défini 11 axiomes de continuité et établi leur hiérarchie. Nos démonstrations ont été formalisées dans le cadre de la géométrie neutre de Tarski en dimension quelconque et de la logique intuitionniste.

Introduction

Les commentateurs des *Éléments* d'Euclide ont remarqué depuis longtemps que certaines démonstrations d'Euclide ne découlent pas formellement de ses postulats, car Euclide admet implicitement l'existence de certains points. C'est le cas par exemple dans la première proposition du livre I : celle-ci construit un triangle équilatéral sur un segment donné en utilisant l'intersection de deux cercles, sans que l'existence d'une telle intersection n'ait été prouvée ou postulée.

Dans cet article nous dénotons par le mot continuité des axiomes qui affirment l'existence de points sous certaines conditions, par exemple l'existence de l'intersection de deux cercles intriqués. Nous nous concentrons sur des propriétés géométriques, bien que des propriétés algébriques analogues existent.

En 1882, Moritz Pasch a proposé un axiome qui permet de combler un autre type de lacune dans les *Éléments* [Pas82]. L'axiome de Pasch consiste à supposer que quand une droite pénètre à l'intérieur d'un triangle par un côté, elle ressort par un des deux autres côtés. On peut considérer cette propriété comme un axiome de continuité concernant l'intersection des droites ; cependant, l'axiome de Pasch ne fait pas partie des axiomes étudiés dans cet article, car toutes nos démonstrations supposent l'axiome de Pasch.

Au tournant du XXème siècle, David Hilbert puis Alfred Tarski ont proposé des développements systématiques de la géométrie sans supposer d'axiome concernant l'intersection de deux cercles [Hil77, SST83]. En effet, une grande part de la géométrie euclidienne peut être développée sans axiome de continuité (l'axiome de Pasch mis à part) : il est possible de construire des coordonnées et de prouver la plupart des théorèmes de géométrie [BBN19]. Hilbert et Tarski introduisent tout de même des axiomes de continuité dans un objectif principalement métathéorique.

Notre bibliothèque *GeoCoq* contient une formalisation des fondements de la géométrie basée sur les axiomatiques d'Euclide, Hilbert et Tarski. Nous avons prouvé précédemment que les axiomes de Hilbert des groupes I, II et III sont mutuellement interprétables avec les axiomes A_1–A_8 de Tarski [BN12, BBN16]. Dans un premier temps, notre formalisation des systèmes d'axiomes de Tarski et de Hilbert n'incluait pas la continuité. En géométrie euclidienne, la

continuité n'est nécessaire que pour certaines constructions géométriques et pour définir les fonctions trigonométriques inverses.

Dans le cadre de la géométrie neutre [1], la continuité joue également un rôle dans la classification des différentes versions du postulat des parallèles [BGNS17] : certaines sont équivalentes entre elles dans toute géométrie neutre, comme l'unicité de la parallèle et la transitivité du parallélisme ; d'autres ne le sont que sous l'hypothèse d'axiomes de continuité plus ou moins forts. Ainsi, le postulat du triangle, qui stipule que la somme des angles d'un triangle est égale à π (ou plutôt à deux droits), est une conséquence de l'unicité de la parallèle, mais la réciproque n'est pas vraie : Max Dehn a démontré l'existence d'un modèle non-archimédien dans lequel la somme des triangle vaut π mais l'unicité de la parallèle n'est pas vérifiée [Deh00]. Cependant, sous l'hypothèse de l'axiome d'Archimède, les deux postulats sont bel et bien équivalents.

Dans cet article, nous étendons notre étude des liens entre les axiomatique de Hilbert et de Tarski aux propriétés de continuité. Nos preuves formelles reposent sur le développement systématique de la géométrie que Wolfram Schwabhäuser, Wanda Szmielew et Alfred Tarski ont produit à partir du système de Tarski [SST83]. Elles s'inspirent des travaux de Marvin J. Greenberg [Gre10] et de George E. Martin [Mar98], ainsi que de l'ouvrage de Franz Rothe [Rot14] et d'un article de Victor Pambuccian [Pam18]. Cependant, les preuves de ces écrits ne sont parfois qu'esquissées, et elles ne sont pas vérifiées mécaniquement, ce qui permet d'échapper à la démonstration parfois laborieuse de résultats intermédiaires semblant évidents, notamment concernant la non-dégénérescence des figures et la position relative des points. De plus, comme nous nous restreignons la plupart du temps à la logique intuitionniste, certaines preuves utilisant la loi du tiers exclu ont dû être adaptées pour ce contexte.

Cet article a pour objet de faire le bilan des propriétés de continuité que nous avons étudiées. Nous allons présenter dans la partie 1 les axiomes de continuité que nous avons formalisés, avant de présenter leur hiérarchie et expliquer quelques démonstrations dans la partie 2.

1 Formalisation des axiomes de continuité

Nos preuves sont données dans le cadre des axiomes A_1–A_8 du système de Tarski pour la géométrie neutre en dimension quelconque [2] [SST83], que nous ne détaillons pas ici. Nous avons décrit ces axiomes et leur formalisation en français dans [GBN16] et de manière plus étendue dans [BGNS17]. Nous avons présenté dans ces articles plusieurs axiomes de continuité, dont ceux que nous appelons axiome d'Aristote et axiome de Greenberg.

Rappelons que le système de Tarski est basé sur un seul type primitif, représentant les points, et deux prédicats, à savoir la congruence (ou équidistance) et l'interposition (en anglais *betweeness*) :

— Cong $ABCD$ indique que les segments AB et CD sont de même longueur ;
— Bet ABC signifie que A, B et C sont colinéaires et que B est entre A et C.

Ces deux prédicats permettent notamment de définir les prédicats suivants de [SST83] :

— Col ABC : A, B et C sont colinéaires ;
— Midpoint MAB : M est le milieu du segment AB ;
— Out PAB : les points A et B se situent sur la même demi-droite ouverte d'origine P ;
— Le $ABCD$: la longueur AB est inférieure ou égale à la longueur CD, c'est-à-dire qu'il existe un point E tel que CE est congru à AB et D est entre C et E ;

1. La géométrie neutre recouvre l'ensemble des résultats communs entre la géométrie euclidienne et la géométrie hyperbolique. C'est-à-dire qu'il existe des droites parallèles mais elles ne sont pas forcément uniques. Cela exclut la géométrie sphérique.
2. Nous ne considérons que les espaces de dimension supérieure ou égale à 2.

— Lt $ABCD$: la longueur AB est strictement inférieure à la longueur CD.

Nous travaillons principalement en logique intuitionniste ; toutefois, nous supposons décidable l'égalité de points afin de pouvoir raisonner par tiers exclu sur ce type de proposition. Cela permet notamment de démontrer la décidabilité des prédicats que nous venons de lister [BNSB14].

1.1 Axiomes de continuité de Tarski

Considérons à présent les axiomes de continuité utilisés par Tarski dans son article fondateur [Tar51]. Nous en exposerons d'abord la présentation donnée par Tarski, avant de développer leur signification et expliquer notre formalisation.

1.1.1 Axiome de Dedekind

L'axiome de continuité de Dedekind est un axiome du deuxième ordre [3] : il comprend une quantification sur des ensembles α et β.

Axiome de Dedekind A_{11}

$$\forall \alpha\beta\ (\exists a\ \forall xy\ (x \in \alpha \wedge y \in \beta \Rightarrow \text{Bet}\ axy)) \Rightarrow (\exists b\ \forall xy\ (x \in \alpha \wedge y \in \beta \Rightarrow \text{Bet}\ xby))$$

La formalisation de cet axiome en Coq est directe ; nous quantifions sur les ensembles α et β formalisés par des prédicats sur les points :

```
Definition dedekind_s_axiom := forall (Alpha Beta : Tpoint -> Prop),
  (exists A, forall X Y, Alpha X -> Beta Y -> Bet A X Y) ->
  (exists B, forall X Y, Alpha X -> Beta Y -> Bet X B Y).
```

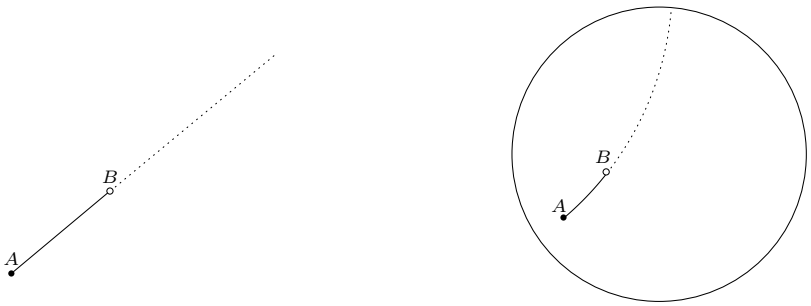


FIGURE 1 – Axiome de Dedekind

On peut déduire de cet axiome l'ensemble des axiomes de continuité. Sans recourir à la quantification sur les ensembles, qui ne sont pas définissables dans le système de Tarski, il exprime la résolution des coupures de Dedekind (fig. 1 [4]), qui caractérise les espaces complets.

3. Les formules du premier ordre sont des formules dans lesquelles on ne peut quantifier que sur les variables, c'est-à-dire, dans ce contexte, sur les points.

4. Nous présenterons la plupart des figures à la fois dans le modèle euclidien et dans un modèle non-euclidien : le disque de Poincaré. La figure de gauche illustrera la validité de l'axiome en géométrie euclidienne. La figure de droite jouera un rôle analogue dans le disque de Poincaré.

En effet, sous les hypothèses de cet axiome, les points X vérifiant `Alpha` X et les points Y vérifiant `Beta` Y sont situés sur une même demi-droite d'origine A[5], et on peut ordonner totalement cette demi-droite en utilisant la relation d'interposition et le point A. Munis de cet ordre, nous pouvons dire que tout point X est inférieur à tout point Y. Si de plus les points X et Y forment un partition de la demi-droite (c'est-à-dire qu'ils décrivent l'ensemble de la demi-droite sans se rencontrer), on dit qu'il forment une coupure de Dedekind de cette demi-droite. Construire un point B situé entre tout point X et tout point Y revient alors à exhiber la borne inférieure de l'ensemble des points Y, c'est-à-dire à résoudre la coupure de Dedekind associée.

Sur la figure 1, les points X vérifiant `Alpha` X se situent au sein du trait plein, tandis que les points Y vérifiant `Beta` Y se trouvent parmi le trait pointillé.

Pour les besoins de notre étude, nous avons introduit une variante plus explicite de cet axiome :

```
Definition dedekind_variant := forall (Alpha Beta : Tpoint -> Prop) A C,
  Alpha A -> Beta C -> (forall P, Out A P C -> Alpha P \/ Beta P) ->
  (forall X Y, Alpha X -> Beta Y -> Bet A X Y /\ X <> Y) ->
  (exists B, forall X Y, Alpha X -> Beta Y -> Bet X B Y).
```

Dans cette version, les points X vérifiant `Alpha` X et les points Y vérifiant `Beta` Y forment nécessairement une coupure de Dedekind de la demi-droite $[AC$. Cette précision fait de cette version une propriété plus facile à prouver ; les deux formulations sont toutefois équivalentes en logique classique.

1.1.2 Schéma d'axiomes de Dedekind de premier ordre

La propriété de continuité A'_{11} introduite par Tarski est un schéma d'axiomes de premier ordre qui restreint la résolution des coupures de Dedekind à celles qui sont définissables au premier ordre :

Schéma de Dedekind de premier ordre A'_{11}

$$(\exists a\, \forall xy\, (\alpha(x) \wedge \beta(y) \Rightarrow \text{Bet}\ axy)) \Rightarrow (\exists b\, \forall xy\, (\alpha(x) \wedge \beta(y) \Rightarrow \text{Bet}\ xby)$$

où $\alpha(x)$ (respectivement $\beta(y)$) représente n'importe quelle formule qui ne contienne aucune occurrence libre de a, b, y (resp. a, b, x) et qui soit du premier ordre dans le langage du système d'axiomes de Tarski.

Pour formaliser ce schéma d'axiomes dans la logique d'ordre supérieur de Coq, nous devons transformer ce schéma en un axiome du deuxième ordre tout en restreignant la quantification sur $\alpha(x)$ et $\beta(y)$ aux formules définissables au premier ordre dans le langage du système d'axiomes de Tarski.

Nous utilisons la définition inductive suivante, qui définit un prédicat permettant d'indiquer si une formule est de premier ordre (en anglais *first-order formula*).

```
Inductive FOF : Prop -> Prop :=
| eq_fof : forall A B:Tpoint, FOF (A = B)
| bet_fof : forall A B C, FOF (Bet A B C)
| cong_fof : forall A B C D, FOF (Cong A B C D)
| not_fof : forall P, FOF P -> FOF (~ P)
| and_fof : forall P Q, FOF P -> FOF Q -> FOF (P /\ Q)
```

5. À condition qu'il existe un tel point X distinct de A.

```
| or_fof : forall P Q, FOF P -> FOF Q -> FOF (P \/ Q)
| implies_fof : forall P Q, FOF P -> FOF Q -> FOF (P -> Q)
| forall_fof : forall P, (forall (A:Tpoint), FOF (P A)) -> FOF (forall A, P A)
| exists_fof : forall P, (forall (A:Tpoint), FOF (P A)) -> FOF (exists A, P A).
```

Afin de nous convaincre que cette définition est correcte, nous avons proposé une définition alternative : nous disons qu'une formule est de premier ordre si elle est l'interprétation d'un terme dans un langage de la théorie du premier ordre. Nous avons d'abord formalisé ce langage :

```
Inductive tFOF :=
| eq_fof1 : Tpoint -> Tpoint -> tFOF
| bet_fof1 : Tpoint -> Tpoint -> Tpoint -> tFOF
| cong_fof1 : Tpoint -> Tpoint -> Tpoint -> Tpoint -> tFOF
| not_fof1 : tFOF -> tFOF
| and_fof1 : tFOF -> tFOF -> tFOF
| or_fof1 : tFOF -> tFOF -> tFOF
| implies_fof1 : tFOF -> tFOF -> tFOF
| forall_fof1 : (Tpoint -> tFOF) -> tFOF
| exists_fof1 : (Tpoint -> tFOF) -> tFOF.
```

Nous avons ensuite défini l'interprétation de notre langage dans les propositions de Coq, par récurrence sur la structure des termes :

```
Fixpoint fof1_prop (F:tFOF) := match F with

| eq_fof1 A B => A = B
| bet_fof1 A B C => Bet A B C
| cong_fof1 A B C D => Cong A B C D
| not_fof1 F1 => ~ fof1_prop F1
| and_fof1 F1 F2 => fof1_prop F1 /\ fof1_prop F2
| or_fof1 F1 F2 => fof1_prop F1 \/ fof1_prop F2
| implies_fof1 F1 F2 => fof1_prop F1 -> fof1_prop F2
| forall_fof1 P => forall A, fof1_prop (P A)
| exists_fof1 P => exists A, fof1_prop (P A) end.
```

Enfin, nous avons prouvé que les deux définitions sont équivalentes dans le sens suivant :

— l'interprétation de tout terme du premier ordre est une formule du premier ordre ;

— pour toute formule du premier ordre, il existe un terme du premier ordre dont l'interprétation est équivalente à la formule [6].

```
Lemma fof1__fof : forall F1:tFOF, FOF (fof1_prop F1).

Lemma fof__fof1 : FunctionalChoice_on Tpoint tFOF ->
  forall F:Prop, FOF F -> exists F1:tFOF,  F <-> fof1_prop F1.
```

Cela justifie notre définition formelle du schéma d'axiomes de continuité :

```
Definition first_order_dedekind := forall Alpha Beta,
  (forall X, FOF (Alpha X)) -> (forall Y, FOF (Beta Y)) ->
  (exists A, forall X Y, Alpha X -> Beta Y -> Bet A X Y) ->
  (exists B, forall X Y, Alpha X -> Beta Y -> Bet X B Y).
```

6. La preuve de cette affirmation repose sur l'hypothèse `FunctionalChoice`, qui établit qu'une fonction peut être construite à partir d'une relation totale à gauche. Le reste du développement Coq ne repose pas sur cette hypothèse.

1.2 Axiomes de continuité de Hilbert

Dans cette partie, nous allons décrire notre formalisation des axiomes de continuité de Hilbert tels qu'ils sont présentés dans les dernières éditions des *Fondements de la Géométrie* [Hil77].

1.2.1 Axiome d'Archimède

Le premier axiome de continuité utilisé par Hilbert est l'axiome d'Archimède [7] (fig. 2).

V-1 : Axiome d'Archimède Si AB et CD sont deux segments quelconques, il existe un nombre entier n tel que le report du segment AB répété n fois à partir de C sur la demi-droite déterminée par D, conduit à un point situé au-delà de D [8].

En d'autres termes, étant donnés deux segments AB et CD avec A distinct de B, il existe un entier n et $n+1$ points $A_0, \cdots, A_n$ de la droite CD, tels que A_j se situe entre A_{j-1} et A_{j+1} pour $0 < j < n$, A_jA_{j+1} et AB ont même longueur pour $0 \leq j < n$, $A_0 = C$ et D se situe entre A_0 et A_n. Sur la figure 2, l'entier n est égal à 4.

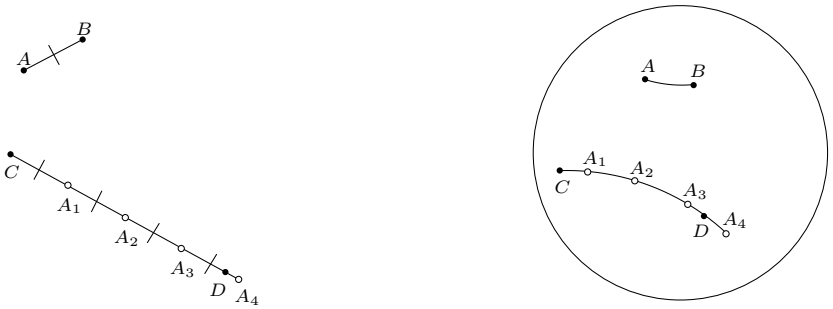


FIGURE 2 – Axiome d'Archimède

On dira qu'un espace géométrique est archimédien s'il vérifie l'axiome d'Archimède. Cet axiome peut être dérivé de A_{11}, mais pas de A'_{11}, car il s'agit d'une propriété du deuxième ordre.

En nous inspirant du travail de Jean Duprat [Dup10], nous avons défini l'axiome d'Archimède de façon inductive, sans introduire explicitement les entiers naturels. Pour cela, nous avons d'abord traduit le fait "il existe un entier n et $n+1$ points $A_0, \cdots, A_n$ de la droite AB, tels que A_j se situe entre A_{j-1} et A_{j+1} pour $0 < j < n$, A_jA_{j+1} et AB ont même longueur pour $0 \leq j < n$, $A_0 = A$ et $A_n = C$" par le prédicat `Grad` : `Grad A B C` signifie que C appartient à la demi-droite $[AB$ et que la distance AC est un multiple de la distance AB.

```
Inductive Grad : Tpoint -> Tpoint -> Tpoint -> Prop :=
  | grad_init : forall A B, Grad A B B
  | grad_stab : forall A B C C',
                Grad A B C ->
                Bet A C C' -> Cong A B C C' ->
                Grad A B C'.
```

7. L'axiome d'Archimède était le seul axiome de continuité présent dans la première édition des *Fondements de la Géométrie*.

8. Par souci de cohérence avec notre développement Coq, nous avons interverti les rôles de AB et CD par rapport à la formulation de Hilbert dans [Hil77].

```
Definition Reach A B C D := exists B', Grad A B B' /\ Le C D A B'.
```

`Grad A B B'` indique que B' fait partie de la graduation basée sur le segment AB. La notation `Reach A B C D` signale donc que le segment CD est inférieur ou égal à un segment de la forme A_0A_n basé sur le segment AB, c'est-à-dire qu'il existe un point E tel que CE est congru à n copies de AB et que D est entre C et E. Nous pouvons donc en dériver une définition de l'axiome d'Archimède :

```
Definition archimedes_axiom := forall A B C D, A <> B -> Reach A B C D.
```

Notons que nous avons précédemment étudié une propriété analogue portant sur les angles, à savoir que tout angle aigu non nul peut être ajouté à lui-même jusqu'à obtenir un angle obtus. En nous inspirant de la démonstration présentée par Hartshorne [Har00], nous avons formalisé la preuve que cette propriété était une conséquence de l'axiome d'Archimède. Cependant, comme nous n'avons pas pu la relier à un autre axiome de continuité, nous n'en donnerons pas ici une définition formelle, qui nécessiterait d'introduire les notions nécessaires à la somme d'angles [9], ainsi que la variante du prédicat `Grad` associée.

1.2.2 Axiomes d'intégrité de Hilbert

Dans la première édition des *Fondements de la géométrie* [Hil99], l'axiome d'Archimède est le seul axiome de continuité. De la deuxième édition à la sixième édition, il introduit également l'axiome d'intégrité [10], qui n'est pas utilisé dans le développement si ce n'est pour des considérations métamathématiques. Dans la septième édition, suite à une remarque de Paul Bernays, l'axiome d'intégrité est remplacé par l'axiome d'intégrité linéaire, qui implique l'axiome d'intégrité.

L'axiome d'intégrité peut paraître déroutant au premier abord. En effet, cet axiome est une propriété à propos des modèles des autres axiomes :

Axiome d'intégrité Les éléments de la géométrie (les points, les droites et les plans) constituent un ensemble qui n'est susceptible d'aucune extension si les axiomes d'appartenance, d'ordre, de congruence et l'axiome d'Archimède sont conservés.

Nous avons interprété cette définition en considérant qu'un modèle $\mathcal{M}$ des axiomes de la géométrie respecte l'axiome d'intégrité s'il n'existe aucune fonction f de $\mathcal{M}$ dans un modèle archimédien $\mathcal{M}'$ vérifiant les trois propriétés suivantes :

— f préserve les propriétés géométriques (ici l'interposition et la congruence) ;
— f est injective (f envoie deux points distincts sur deux images distinctes) ;
— f n'est pas surjective (certains points de $\mathcal{M}'$ sont inaccessibles par f).

Une autre façon de le formuler est que sous l'axiome d'intégrité, toute fonction injective vers un modèle archimédien et préservant les propriétés géométriques est également surjective, ce qui fait de f un isomorphisme. Ainsi, les isométries sont des isomorphismes. On peut considérer les isomorphismes comme des extensions non propres, et qualifier de propres les extensions non surjectives.

9. Nous avons précédemment présenté en français notre définition formelle de la somme d'angles [GBN16].
10. En allemand, le terme de complétude (*Vollständigkeit*) est utilisé pour décrire cet axiome. Richard Baldus a fait remarquer que la complétude obtenue n'est pas à comprendre au sens métathéorique. En effet, l'axiome des parallèles n'est pas mentionné dans l'axiome d'intégrité, la théorie n'est donc pas complète puisque l'axiome des parallèles est indépendant des axiomes considérés [Bal28].

Remarquons que notre formalisation de l'axiome fait référence aux axiomes A_1–A_8 de Tarski en lieu et place des groupes d'axiomes d'appartenance (I), d'ordre (II) et de congruence (III) de Hilbert. Ce choix est motivé par souci de cohérence avec le reste de la bibliothèque *GeoCoq* mais n'a pas d'impact, puisque nous avons déjà formalisé la preuve que les systèmes d'axiomes en question sont mutuellement interprétables [BN12, BBN16].

Eduardo Giovannini a établi que Hilbert était familier avec l'axiome de continuité de Dedekind A_{11}. Hilbert aurait choisi l'axiome d'intégrité parce d'une part il n'implique pas l'axiome d'Archimède (tandis que la conjonction de l'axiome d'intégrité et de celui d'Archimède caractérise les espaces complets) et d'autre part, Hilbert n'aurait pas souhaité utiliser l'axiome de Cantor dont l'énoncé repose sur le concept de suite qui n'est pas purement géométrique [Gio13].

Nous avons pu formaliser la preuve que la négation de l'axiome d'Archimède implique l'axiome d'intégrité : un modèle non-archimédien ne peut disposer d'une extension (propre ou non propre) vers un modèle archimédien. À l'opposé, tout modèle (archimédien ou non) peut être muni d'une extension propre vers un modèle non-archimédien [Hil77] ; omettre l'axiome d'Archimède dans l'énoncé de l'axiome d'intégrité aboutit donc à un énoncé contradictoire. Le rôle de cet axiome et sa relation avec celui d'Archimède ont été analysés par Philip Ehrlich [Ehr97].

Dans [Hil77], les espaces géométriques sont supposés être de dimension 3, même s'il est possible de modifier les axiomes pour travailler sur le plan ou en dimension quelconque. Cette formulation de l'axiome d'intégrité suppose donc implicitement que les deux modèles concernés sont de même dimension finie (en l'occurrence 3). Cette propriété est fondamentale pour l'*intégrité* de cet axiome : si on oublie cette hypothèse, on obtient un axiome contradictoire avec les espaces archimédiens. Ainsi, tout espace archimédien $\mathcal{M}'$ de dimension 3 dispose d'un infinité de sous-espaces $\mathcal{M}$ de dimension 2, pour lesquels l'inclusion de $\mathcal{M}$ dans $\mathcal{M}'$ est une extension propre. Ne disposant pas d'un prédicat pour exprimer le fait pour deux espaces d'être de même dimension finie, nous avons formalisé deux versions de cette axiome convenant respectivement aux espaces de dimension 2 et 3 : elles consistent à affirmer que toute extension vers un espace respectivement de dimension 2 ou 3 est non propre [11].

```
Definition inj {T1 T2:Type} (f:T1->T2) := forall A B, f A = f B ->
  A = B.

Definition pres_bet {Tm: Tarski_neutral_dimensionless}
  (f : @Tpoint Tn -> @Tpoint Tm) := forall A B C, Bet A B C ->
  Bet (f A) (f B) (f C).

Definition pres_cong {Tm: Tarski_neutral_dimensionless}
  (f : @Tpoint Tn -> @Tpoint Tm) := forall A B C D, Cong A B C D ->
  Cong (f A) (f B) (f C) (f D).

Definition extension {Tm: Tarski_neutral_dimensionless}
  (f : @Tpoint Tn -> @Tpoint Tm) := inj f /\ pres_bet f /\ pres_cong f.

Definition completeness_for_planes := forall
  (Tm: Tarski_neutral_dimensionless)
  (Tm2 : Tarski_neutral_dimensionless_with_decidable_point_equality Tm)
  (M : Tarski_2D Tm2)
  (f : @Tpoint Tn -> @Tpoint Tm),
  @archimedes_axiom Tm ->
  extension f ->
```

11. Dans notre développement, `Tn` désigne le contexte courant, à savoir les axiomes de Tarski pour la géométrie neutre.

```
  forall A, exists B, f B = A.

Definition completeness_for_3d_spaces := forall
  (Tm: Tarski_neutral_dimensionless)
  (Tm2 : Tarski_neutral_dimensionless_with_decidable_point_equality Tm)
  (M : Tarski_3D Tm2)
  (f : @Tpoint Tn -> @Tpoint Tm),
  @archimedes_axiom Tm ->
  extension f ->
  forall A, exists B, f B = A.
```

Considérons maintenant l'axiome d'intégrité linéaire :

V-2 : Axiome d'intégrité linéaire L'ensemble des points d'une droite, soumis aux relations d'ordre et de congruence, n'est susceptible d'aucune extension dans laquelle sont valables les relations précédentes et les propriétés fondamentales d'ordre linéaire et de congruence déduites des axiomes I–III et de l'axiome V-1 (axiome d'Archimède).

Remarquons que dans la septième édition des *Fondements de la Géométrie*, Hilbert donne une variante de cet axiome avec une liste plus restreinte des propriétés fondamentales valables dans l'extension. Werner Weber a montré que cette variante était contradictoire [Web38]. L'usage de la preuve formelle peut permettre d'éviter ce genre de désagrément.

Ce nouvel axiome présente l'avantage de ne nécessiter aucune hypothèse sur la dimension des espaces géométriques. En effet, il ne s'agit plus d'étudier les extensions de l'espace $\mathcal{M}$ lui-même, mais des droites de $\mathcal{M}$; pour satisfaire l'axiome d'intégrité linéaire, il suffit alors de vérifier que toute extension f d'une droite quelconque PQ couvre l'ensemble de la droite $f(P)f(Q)$, ce qui ne dépend pas de la dimension de l'espace archimédien $\mathcal{M}'$ dans lequel elle se trouve.

Nous avons donc pu formaliser cet axiome sans peine, après avoir défini le fait pour f d'être une extension de la droite PQ :

```
Definition inj_line {T:Type} (f:Tpoint->T) P Q := forall A B,
  Col P Q A -> Col P Q B -> f A = f B -> A = B.

Definition pres_bet_line {Tm: Tarski_neutral_dimensionless}
  (f : @Tpoint Tn -> @Tpoint Tm) P Q := forall A B C,
  Col P Q A -> Col P Q B -> Col P Q C ->
  Bet A B C -> Bet (f A) (f B) (f C).

Definition pres_cong_line {Tm: Tarski_neutral_dimensionless}
  (f : @Tpoint Tn -> @Tpoint Tm) P Q := forall A B C D,
  Col P Q A -> Col P Q B -> Col P Q C -> Col P Q D ->
  Cong A B C D -> Cong (f A) (f B) (f C) (f D).

Definition line_extension {Tm: Tarski_neutral_dimensionless} f P Q :=
  P <> Q /\ inj_line f P Q /\ pres_bet_line f P Q /\ pres_cong_line f P Q.

Definition line_completeness := forall
  (Tm: Tarski_neutral_dimensionless)
  (Tm2 : Tarski_neutral_dimensionless_with_decidable_point_equality Tm)
  P Q (f : @Tpoint Tn -> @Tpoint Tm),
  @archimedes_axiom Tm ->
  line_extension f P Q ->
  forall A, Col (f P) (f Q) A -> exists B, Col P Q B /\ f B = A.
```

La démonstration de Paul Bernays prouve que l'axiome d'intégrité linéaire implique l'axiome d'intégrité dans les espaces de dimension 3. En plus de la formaliser, nous avons pu l'adapter pour les espaces de dimension 2.

1.3 Axiome des segments emboîtés de Cantor

Georg Cantor a introduit la propriété des segments emboîtés dans sa première preuve que l'ensemble des points d'un segment est indénombrable [Can74]. Nous utilisons ici une version de l'axiome de Cantor qui ne suppose pas que la suite des longueurs de segments tend vers zéro. Cette variante a notamment été étudiée par Richard Baldus [Bal28, Bal30].

Axiome des segments emboîtés Étant donnée une suite $([A_nB_n])_{n\in\mathbb{N}}$ de segments fermés emboîtés, il existe un point X appartenant à tous les segments A_nB_n.

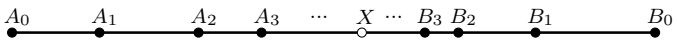


FIGURE 3 – Axiome des segments emboîtés de Cantor

Les segments A_nB_n sont dits emboîtés si pour tout entier n, le segment A_nB_n contient le segment $A_{n+1}B_{n+1}$; par commodité, il est d'usage de considérer que les points A_n, A_{n+1}, B_{n+1} et B_n sont alignés dans cet ordre (fig. 3).

Là encore, nous avons affaire à un axiome du deuxième ordre, car il contient une quantification sur des suites de points.

Pour décrire la suite des extrémités des segments, nous allons utiliser des prédicats `A` et `B` prenant en argument un entier n et un point P : `A n P` (respectivement `B n P`) signifie que P correspond au point A_n (resp. B_n). Dans cette optique, `Nested A B` indique que les prédicats `A` et `B` jouent ce rôle pour des segments emboîtés :

```
Definition Nested (A B:nat -> Tpoint -> Prop) :=
  (forall n, exists An Bn, A n An /\ B n Bn) /\
  forall n An Am Bm Bn,
    A n An -> A (S n) Am -> B (S n) Bm -> B n Bn ->
    Bet An Am Bm /\ Bet Am Bm Bn /\ Am <> Bm.

Definition cantor_s_axiom := forall A B, Nested A B ->
  exists X, forall n An Bn, A n An -> B n Bn -> Bet An X Bn.
```

On peut remarquer que notre définition ne suppose pas que pour tout entier n, les points A_n et B_n soient définis de façon unique. Le contraire aurait alourdi la formalisation sans jouer un rôle crucial, ce que nous avons vérifié formellement. De plus, nous supposons que les points A_n et B_n sont toujours différents entre eux, ce qui ne modifie pas foncièrement l'axiome mais facilite son usage. Remarquons que cet axiome dépend de la notion d'entier naturel, ce n'est donc pas un énoncé purement géométrique.

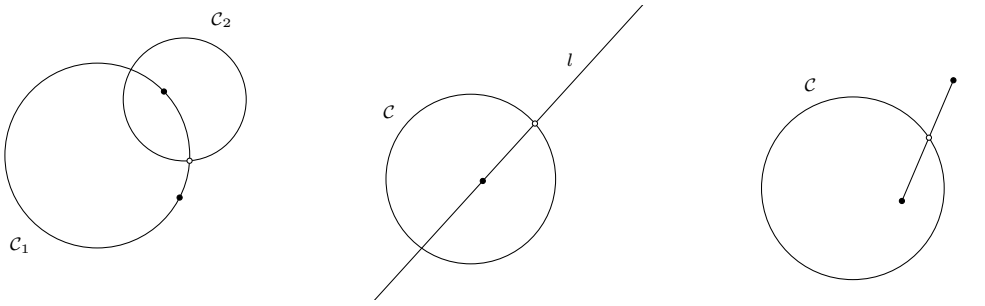
1.4 Axiomes de continuité du cercle

Les axiomes de continuité du cercle, ajoutés aux axiomes de la géométrie, décrivent ce qu'on appelle la géométrie à la règle et au compas, pratiquée déjà par Euclide. On en distingue souvent trois (fig. 4) :

Continuité cercle-cercle Étant donnés deux cercles $\mathcal{C}_1$ et $\mathcal{C}_2$, si deux points de $\mathcal{C}_1$ sont respectivement à l'intérieur et à l'extérieur de $\mathcal{C}_2$, alors il existe un point d'intersection des deux cercles.

Continuité cercle-droite Étant donnée une droite l contenant un point à l'intérieur du cercle $\mathcal{C}$, il existe un point de l qui se trouve également sur $\mathcal{C}$.

Continuité cercle-segment Étant donnés un segment PQ et un cercle $\mathcal{C}$ avec P à l'intérieur et Q à l'extérieur de $\mathcal{C}$, il existe un point du segment PQ qui se trouve également sur $\mathcal{C}$.



(a) Cercle–cercle (b) Cercle–droite (c) Cercle–segment

FIGURE 4 – Continuité du cercle

Pour formaliser des axiomes en Coq, nous avons besoin des prédicats suivants, qui permettent respectivement d'affirmer qu'un point P est sur, à l'intérieur, à l'extérieur, strictement à l'intérieur ou strictement à l'extérieur d'un cercle de centre A et de rayon AB.

```
Definition OnCircle P A B := Cong A P A B.
Definition InCircle P A B := Le A P A B.
Definition OutCircle P A B := Le A B A P.
Definition InCircleS P A B := Lt A P A B.
Definition OutCircleS P A B := Lt A B A P.
```

À présent, nous pouvons formaliser les axiomes et leurs variantes. Commençons par l'axiome de continuité cercle-cercle. Sa variante `circle_circle_two` affirme qu'il existe deux points d'intersection. Nous avons prouvé qu'elle est équivalente à la première formulation, le deuxième point d'intersection pouvant être construit à partir du premier. Ce n'est pas complètement trivial : pour procéder ainsi, il est nécessaire de prouver qu'il est possible de prendre l'image d'un point par la réflexion par rapport à une droite sans utiliser aucun axiome de continuité. Les preuves requises ont été développées par Haragauri Nayaran Gupta [Gup65], reprises dans [SST83] et formalisées précédemment [BN12].

```
Definition circle_circle := forall A B C D P Q,
  OnCircle P C D -> OnCircle Q C D ->
  InCircle P A B -> OutCircle Q A B ->
  exists Z, OnCircle Z A B /\ OnCircle Z C D.

Definition circle_circle_two := forall A B C D P Q,
  OnCircle P C D -> OnCircle Q C D ->
  InCircle P A B -> OutCircle Q A B ->
  exists Z1 Z2,
    OnCircle Z1 A B /\ OnCircle Z1 C D /\
    OnCircle Z2 A B /\ OnCircle Z2 C D /\
    (InCircleS P A B -> OutCircleS Q A B -> Z1<>Z2).
```

Une deuxième variante équivalente présente le double avantage d'être symétrique et d'être exactement la propriété requise pour prouver la première proposition des *Éléments* d'Euclide :

```
Definition circle_circle_bis := forall A B C D P Q,
  OnCircle P C D ->
  InCircle P A B ->
  OnCircle Q A B ->
  InCircle Q C D ->
  exists Z, OnCircle Z A B /\ OnCircle Z C D.
```

Une autre proposition du premier livre des *Éléments* d'Euclide est quant à elle équivalente à l'axiome de continuité cercle-cercle :

Proposition 22 Avec trois droites égales à trois droites données construire un triangle ; il faut que deux de ces trois droites, de quelque manière qu'elles soient prises, soient plus grandes que la troisième.

Ici, le mot "droites" désigne en fait des longueurs, et prendre deux longueurs signifie considérer leur somme. En d'autres termes, cette proposition stipule qu'un triangle peut être construit à partir de trois longueurs données à condition que les inégalités triangulaires associées soient respectées. Pour la définir, nous utilisons un prédicat décrivant la somme de longueurs : `SumS A B C D E F` exprime le fait que la longueur du segment EF est égale à la somme des longueurs des segments AB et CD.

```
Definition SumS A B C D E F := exists P Q R,
  Bet P Q R /\ Cong P Q A B /\ Cong Q R C D /\ Cong P R E F.

Definition euclid_s_prop_1_22 := forall A B C D E F A' B' C' D' E' F',
  SumS A B C D E' F' -> SumS A B E F C' D' -> SumS C D E F A' B' ->
  Le E F E' F' -> Le C D C' D' -> Le A B A' B' ->
  exists P Q R, Cong P Q A B /\ Cong P R C D /\ Cong Q R E F.
```

En nous inspirant de `circle_circle_bis`, nous avons défini une autre variante équivalente utilisant uniquement les prédicats de base du système de Tarski, afin de faciliter la définition d'une *type class* décrivant le contexte de la géométrie à la règle et au compas :

```
Definition circle_circle_axiom := forall A B C D B' D',
  Cong A B' A B -> Cong C D' C D ->
  Bet A D' B -> Bet C B' D ->
  exists Z, Cong A Z A B /\ Cong C Z C D.
```

Définissons à présent deux versions de l'axiome de continuité cercle-droite affirmant respectivement l'existence d'un ou deux points d'intersection. L'équivalence entre les deux formulations se démontre de façon analogue à l'équivalence entre les deux premières versions de l'axiome de continuité cercle-cercle.

```
Definition one_point_line_circle := forall A B U V P,
  Col U V P -> U <> V -> Bet A P B ->
  exists Z, Col U V Z /\ OnCircle Z A B.

Definition two_points_line_circle := forall A B U V P,
  Col U V P -> U <> V -> Bet A P B ->
```

```
  exists Z1 Z2, Col U V Z1 /\ OnCircle Z1 A B /\
                Col U V Z2 /\ OnCircle Z2 A B /\
                Bet Z1 P Z2 /\ (P <> B -> Z1 <> Z2).
```

Enfin, la propriété cercle-segment affirme que si un segment a ses extrémités respectivement à l'intérieur et à l'extérieur du cercle, alors il traverse le cercle.

```
Definition segment_circle := forall A B P Q,
  InCircle P A B ->
  OutCircle Q A B ->
  exists Z, Bet P Z Q /\ OnCircle Z A B.
```

2 Liens entre les axiomes de continuité

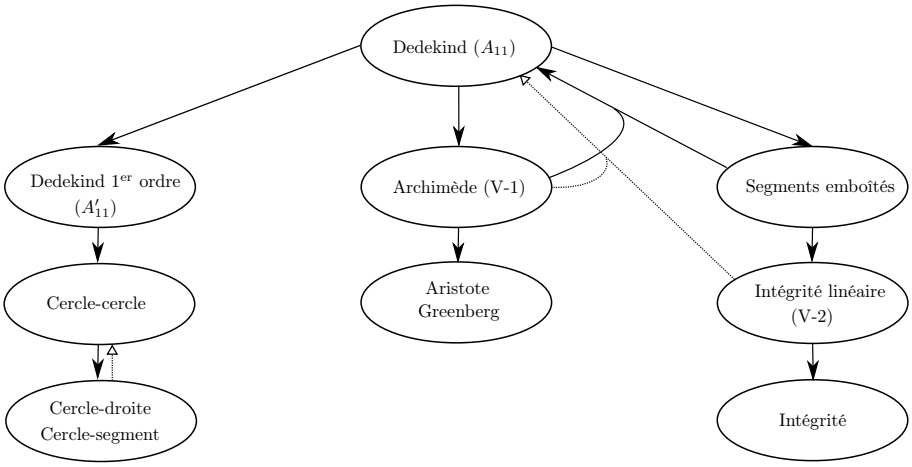


FIGURE 5 – Aperçu des liens entre les axiomes de continuité

La figure 5 présente la hiérarchie des axiomes de continuité en géométrie neutre. Les ovales représentent chacun un axiome ou un groupe d'axiomes dont nous avons formalisé l'équivalence, à l'exception de l'ovale "Intégrité", dont la définition dépend de la dimension, comme nous l'avons vu dans le paragraphe 1.2.2. Les implications que nous avons formalisées sont désignées par des flèches en trait plein. Notons que notre formalisation de la preuve de l'implication entre Cercle-cercle et Cercle-droite résout un des quatre défis proposé par Beeson en 2012 [Bee13] ; cette implication n'est pas triviale à obtenir si on n'a pas formalisé d'abord les résultats de Gupta [Gup65] et Tarski [SST83] permettant d'utiliser la symétrie axiale sans supposer d'axiome de continuité. Les flèches pointillées décrivent des implications connues dans la littérature mais absentes de notre bibliothèque. Ainsi, l'implication entre Cercle-droite et Cercle-cercle a été prouvée par Gyula Strommer [Str73]. Remarquons au passage qu'il n'est pas inutile de formaliser ces résultats, car par exemple l'implication entre la notion de continuité et l'axiome d'Archimède a fait l'objet de controverses entre Georg Cantor, Giuseppe Veronese et Otto Stolz [Ehr06].

Les axiomes présentés et les preuves associées ont été formalisés en logique intuitionniste, à quelques exceptions près : les formalisations des implications Dedekind $\Rightarrow$ Archimède et

Segments emboîtés ⇒ Intégrité linéaire, ainsi que celle de l'équivalence des deux versions de Dedekind, recourent à la logique classique.

La plupart des implications sont strictes, mais nous n'avons pas formalisé en Coq les preuves d'indépendance. Notamment, l'axiome d'Archimède n'est pas une conséquence de l'axiome d'Aristote [Gre88, Gre10].

2.1 La conjonction Segments emboîtés + Archimède implique Dedekind

Nous avons formalisé une preuve par dichotomie que la conjonction des axiomes des segments emboîtés et d'Archimède implique celui de Dedekind. En d'autres termes, notre preuve repose sur une suite de segments dont chacun est une des deux moitiés du précédent, et qui converge vers le point que l'on cherche à construire, à savoir le point de résolution d'une coupure de Dedekind. Nous devons donc associer à toute coupure deux prédicats décrivant les suites des deux extrémités de tels segments emboîtés. Pour cela, il suffit de s'assurer que chacun des segments traverse la coupure, c'est-à-dire que les deux sous-ensembles concernés sont représentés chacun par une des extrémités du segment. À chaque étape, on utilisera ce critère pour décider de définir un des segments emboîtés comme étant l'une ou l'autre moitié du précédent [12] :

```
Inductive cX A C (Alpha Beta : Tpoint -> Prop) : nat -> Tpoint -> Prop :=
| cX_init : cX A C Alpha Beta O A
| cX_same : forall n X Y M, cX A C Alpha Beta n X -> cY A C Alpha Beta n Y ->
                     Midpoint M X Y -> Beta M -> cX A C Alpha Beta (S n) X
| cX_other : forall n X Y M, cX A C Alpha Beta n X -> cY A C Alpha Beta n Y ->
                     Midpoint M X Y -> Alpha M -> cX A C Alpha Beta (S n) M
with cY A C (Alpha Beta : Tpoint -> Prop) : nat -> Tpoint -> Prop :=
| cY_init : cY A C Alpha Beta O C
| cY_same : forall n X Y M, cX A C Alpha Beta n X -> cY A C Alpha Beta n Y ->
                    Midpoint M X Y -> Alpha M -> cY A C Alpha Beta (S n) Y
| cY_other : forall n X Y M, cX A C Alpha Beta n X -> cY A C Alpha Beta n Y ->
                      Midpoint M X Y -> Beta M -> cY A C Alpha Beta (S n) M.
```

Ici, `Alpha` et `Beta` désignent les deux prédicats caractérisant la coupure de Dedekind, tandis que A est l'origine de la demi-droite coupée et C un témoin de `Beta`. Notre définition mutuellement inductive permet de prouver, en raisonnant par induction mutuelle, que les prédicats `cX A C Alpha Beta` et `cY A C Alpha Beta` décrivent des segments emboîtés. L'axiome des segments emboîtés garantit donc l'existence d'un point B appartenant à chacun des segments. Comme chaque segment est deux fois plus petit que le précédent, l'axiome d'Archimède permet de prouver qu'ils convergent vers le point B, dont on démontre alors qu'il résout la coupure.

2.2 Dedekind implique l'intégrité linéaire

L'axiome d'intégrité linéaire est une conséquence de l'axiome de Dedekind [13]. Pour formaliser la preuve de cette implication entre des axiomes géométriques, nous nous sommes inspirés de la preuve de James Forsythe Hall concernant les énoncés analogues dans les corps ordonnés [Hal11].

Cette preuve repose sur le fait que l'image de toute extension f dans un corps archimédien est dense dans ce corps, c'est-à-dire qu'entre deux éléments distincts A et B du corps d'arrivée,

12. La définition des moitiés d'un segment repose sur l'existence de son milieu, qui a été démontrée par Gupta [Gup65].

13. Cette implication n'est pas directement représentée sur la figure 5, car le lecteur peut la déduire des autres implications.

il existe toujours un élément X tel que $f(X)$ est situé strictement entre A et B. Notre traduction de cette propriété en termes géométriques est la suivante :

```
Lemma extension_image_density : forall {Tm: Tarski_neutral_dimensionless}
  {Tm2 : Tarski_neutral_dimensionless_with_decidable_point_equality Tm}
  P Q (f : @Tpoint Tn -> @Tpoint Tm),
  @archimedes_axiom Tm ->
  line_extension f P Q ->
  forall A B, Col (f P) (f Q) A -> Col (f P) (f Q) B -> A <> B ->
  exists X, Col P Q X /\ Bet A (f X) B /\ f X <> A /\ f X <> B.
```

Cependant, la preuve originale établit cette propriété en utilisant explicitement la structure de corps : elle s'appuie sur la densité de $\mathbb{Q}$ dans tout corps archimédien. Or, les axiomes de la géométrie neutre ne nous permettent pas de définir de structure de corps au sein d'une droite de notre espace : l'axiome des parallèles (caractérisant la géométrie euclidienne) est requis.

Nous avons donc dû trouver un autre moyen de prouver cette propriété. Nous l'avons fait en introduisant principalement deux lemmes intermédiaires. Le premier établit que pour toute extension de droite f, tous points A et B dans la droite de départ, et tout entier n, il existe un point C tel que $f(A)f(C) = f(A)f(B)\times n$ (il s'agit en fait du point C tel que $AC = AB\times n$). En effet, multiplier une longueur par un entier n'utilise pas de structure de corps, mais le prédicat `Grad` que nous avons défini dans le paragraphe 1.2.1 :

```
Lemma extension_grad : forall {Tm: Tarski_neutral_dimensionless}
  {Tm2 : Tarski_neutral_dimensionless_with_decidable_point_equality Tm}
  P Q (f : @Tpoint Tn -> @Tpoint Tm),
  line_extension f P Q ->
  forall A B X, Col P Q A -> Col P Q B -> Grad (f A) (f B) X ->
  exists C, Col P Q C /\ Grad A B C /\ f C = X.
```

Comme l'axiome d'intégrité comporte un espace archimédien, il était prévisible que notre démonstration fasse appel au prédicat `Grad`, sur lequel repose notre définition de l'axiome d'Archimède. Notre deuxième lemme intermédiaire utilise une de ses variantes : `GradExp A B C` signifie que le segment AC est le produit du segment AB par une puissance de 2. Cela revient à dire que le segment AB est le quotient de AC par un entier de la forme 2^n, ou encore qu'il existe un entier n tel que le segment AB est obtenu après n bissections successives de AC [14]. Ici, il nous permet de formuler le fait que pour toute extension de droite f, tous points A et B dans la droite de départ, et tout entier naturel n, il existe un point C tel que $f(A)f(C) = f(A)f(B)/2^n$ (il s'agit en fait du point C tel que $AC = AB/2^n$) [15] :

```
Inductive GradExp : Tpoint -> Tpoint -> Tpoint -> Prop :=
  | gradexp_init : forall A B, GradExp A B B
  | gradexp_stab : forall A B C C',
                     GradExp A B C ->
                     Bet A C C' -> Cong A C C C' ->
                     GradExp A B C'.

Lemma extension_gradexp : forall {Tm: Tarski_neutral_dimensionless}
```

14. On peut remarquer que ce prédicat est utilisé pour raisonner par dichotomie dans la formalisation de la sous-partie 2.1.

15. Remarquons qu'entre la définition du prédicat `GradExp` et celle de notre lemme, nous avons interverti les rôles des points B et C, afin de valoriser le fait que le point C est défini en fonction de A et B.

```
{Tm2 : Tarski_neutral_dimensionless_with_decidable_point_equality Tm}
P Q (f : @Tpoint Tn -> @Tpoint Tm),
line_extension f P Q ->
forall A B X, Col P Q A -> Col P Q B -> GradExp (f A) X (f B) ->
exists C, Col P Q C /\ GradExp A C B /\ f C = X.
```

La combinaison de ces deux lemmes intermédiaires nous permet d'affirmer que pour toute extension de droite f et tous points A et B de la droite de départ, l'image de f comprend tout point X de la droite d'arrivée issu d'une construction de la forme $f(A)X = f(A)f(B) \times n/2^m$, où n et m désignent des entiers naturels. Si la droite d'arrivée est archimédienne, on peut en déduire que l'image de f est dense dans cette droite, ce qui constitue le principal argument que l'axiome de Dedekind implique celui d'intégrité linéaire :

Démonstration. Étant donnés une extension f d'une droite PQ et un point A sur la droite $f(P)f(Q)$, nous devons montrer qu'il existe un point B tel que $f(B) = A$. Sans perte de généralité on peut supposer que A est différent de $f(P)$. Soit A_1 le symétrique de $f(P)$ par rapport à A. Le lemme de densité de l'image nous permet de construire un point R tel que $f(R)$ est situé strictement entre A et A_1. Le point A est donc situé entre $f(P)$ et $f(R)$. On considère ensuite la coupure de la demi-droite $]PR$ distinguant les points dont l'image par f est située strictement entre $f(P)$ et A et les points dont l'image est située sur la demi-droite $[Af(R)$. L'axiome de Dedekind nous permet d'obtenir un point B résolvant cette coupure. Si $f(B)$ était différent de A, on pourrait construire par densité un point X dont l'image se situerait strictement entre A et $f(B)$, ce qui contredirait la définition de B. On a donc $f(B) = A$. □

Conclusion

Nos précédents résultats sur la continuité avaient contribué à la classification de différentes versions du postulat des parallèles [BGNS17]. Dans cet article, nous avons étendu cette étude pour y intégrer des axiomes de continuité historiquement importants et reconstruire une hiérarchie dans les axiomes de continuité. Nous avons aussi exhibé un chemin permettant de remonter à l'axiome le plus fort, à savoir celui de Dedekind.

Suite à nos précédents travaux, la bibliothèque *GeoCoq* permettait de choisir parmi 34 axiomes des parallèles. Dorénavant, l'utilisateur peut aussi choisir le ou les axiomes de continuité les plus adaptés à ses besoins. Ainsi, l'utilisateur dérouté par l'axiome d'intégrité pourra lui préférer l'axiome des segments emboîtés de Cantor. Outre sa définition géométriquement intuitive, l'axiome des segments emboîtés présente à la fois l'avantage d'être indépendant de l'axiome d'Archimède et de suffire à obtenir la complétude en présence d'Archimède.

L'originalité de nos démonstrations réside dans le fait qu'elles s'inscrivent dans la géométrie neutre de Tarski et que nous travaillons majoritairement en logique intuitionniste.

Ce travail peut avoir plusieurs extensions pouvant se recouper entre elles. D'abord, certains des axiomes de continuité pour les corps ordonnés listés dans [Hal11] (par exemple la version indénombrable de l'axiome des segments emboîtés) pourraient être adaptés à la géométrie et intégrés à notre étude. Ensuite, nous pourrions nous restreindre à la géométrie euclidienne et utiliser son arithmétisation [BBN19] pour prouver l'équivalence entre certains axiomes de continuité géométriques et leurs traductions respectives au sein des corps ordonnés. Nous pourrions aussi intégrer à notre étude une version faible de l'axiome d'Archimède exprimable comme un schéma d'axiomes au premier ordre [Pam18]. Enfin, il serait intéressant de clarifier notre hiérarchie en formalisant les résultats d'indépendance ou d'implication entre nos différents axiomes.

Disponibilité Le développement Coq décrit dans cet article est disponible dans la bibliothèque *GeoCoq* : http://geocoq.github.io/GeoCoq/

Références

[Bal28] Richard Baldus. Zur Axiomatik der Geometrie. I : Über Hilberts Vollständigkeitsaxiom. *Mathematische Annalen*, 100(1) :321–333, 1928.

[Bal30] Richard Baldus. Zur Axiomatik der Geometrie. III : Über das Archimedische und das Cantorsche Axiom. *Sitzungsberichte Heidelberg*, 5(12), 1930.

[BBN16] Gabriel Braun, Pierre Boutry, and Julien Narboux. From Hilbert to Tarski. In *Eleventh International Workshop on Automated Deduction in Geometry*, Proceedings of ADG 2016, page 19, Strasbourg, France, June 2016.

[BBN19] Pierre Boutry, Gabriel Braun, and Julien Narboux. Formalization of the Arithmetization of Euclidean Plane Geometry and Applications. *Journal of Symbolic Computation*, 98 :149–168, 2019.

[Bee13] Michael Beeson. Proof and Computation in Geometry. In Tetsuo Ida and Jacques Fleuriot, editors, *Automated Deduction in Geometry (ADG 2012)*, volume 7993 of *Springer Lecture Notes in Artificial Intelligence*, pages 1–30, Heidelberg, 2013. Springer.

[BGNS17] Pierre Boutry, Charly Gries, Julien Narboux, and Pascal Schreck. Parallel postulates and continuity axioms : a mechanized study in intuitionistic logic using Coq. *Journal of Automated Reasoning*, page 68, 2017.

[BN12] Gabriel Braun and Julien Narboux. From Tarski to Hilbert. In Tetsuo Ida and Jacques Fleuriot, editors, *Post-proceedings of Automated Deduction in Geometry 2012*, volume 7993 of *LNCS*, pages 89–109, Edinburgh, United Kingdom, September 2012. Springer.

[BNSB14] Pierre Boutry, Julien Narboux, Pascal Schreck, and Gabriel Braun. A short note about case distinctions in Tarski's geometry. In Francisco Botana and Pedro Quaresma, editors, *Proceedings of the 10th Int. Workshop on Automated Deduction in Geometry*, volume TR 2014/01 of *Proceedings of ADG 2014*, pages 51–65, Coimbra, Portugal, July 2014. University of Coimbra.

[Can74] Georg Cantor. Uber eine Eigenschaft des Inbegriffs aller reellen algebraischen Zahlen. *Journal für die reine und angewandte Mathematik*, 77 :258–262, 1874.

[Deh00] M. Dehn. Die Legendre'schen Sätze über die Winkelsumme im Dreieck. *Mathematische Annalen*, 53(3) :404–439, 1900.

[Dup10] Jean Duprat. Fondements de géométrie euclidienne. 2010.

[Ehr97] Philip Ehrlich. From Completeness to Archimedean Completeness : An Essay in the Foundations of Euclidean Geometry. *A Symposium on David Hilbert*, 110 :57–76, 1997.

[Ehr06] Philip Ehrlich. The Rise of non-Archimedean Mathematics and the Roots of a Misconception I : The Emergence of non-Archimedean Systems of Magnitudes. *Archive for History of Exact Sciences*, 60(1) :1–121, January 2006.

[GBN16] Charly Gries, Pierre Boutry, and Julien Narboux. Somme des angles d'un triangle et unicité de la parallèle : une preuve d'équivalence formalisée en Coq. In *Les vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016)*, Actes des Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA 2016), page 15, Saint Malo, France, January 2016. Jade Algave and Julien Signoles.

[Gio13] Eduardo Giovannini. Completitud y continuidad en Fundamentos de la Geometría de Hilbert : acerca del Vollständigkeitsaxiom. *THEORIA. An International Journal for Theory, History and Foundations of Science*, 28(1) :139–163, 2013.

[Gre88] Marvin Jay Greenberg. Aristotle's axiom in the foundations of geometry. *Journal of Geometry*, 33(1) :53–57, 1988.

[Gre10] Marvin J. Greenberg. Old and New Results in the Foundations of Elementary Plane Euclidean and Non-Euclidean Geometries. *The American Mathematical Monthly*, 117(3) :198–219, March 2010.

[Gup65] Haragauri Narayan Gupta. *Contributions to the axiomatic foundations of geometry.* PhD thesis, University of California, Berkley, 1965.

[Hal11] James Forsythe Hall. Completeness of ordered fields. *arXiv preprint arXiv :1101.5652*, 2011. Citation Key : hall_completeness_2011.

[Har00] Robin Hartshorne. *Geometry : Euclid and beyond.* Undergraduate texts in mathematics. Springer, 2000.

[Hil99] David Hilbert. *Grundlagen der Geometrie.* Leipzig, 1899.

[Hil77] David Hilbert. *Foundations of Geometry.* Paul Bernays, open court publishing edition, 1977. 10th Revised edition.

[Mar98] G. E. Martin. *The Foundations of Geometry and the Non-Euclidean Plane.* Undergraduate Texts in Mathematics. Springer, 1998.

[Pam18] Victor Pambuccian. The elementary Archimedean axiom in absolute geometry. *Submitted*, 2018.

[Pas82] Moritz Pasch. *Vorlesungen über neuere Geometrie.* Teubner, Leipzig, 1882.

[Rot14] Franz Rothe. *Several Topics from Geometry.* unpublished, 2014.

[SST83] Wolfram Schwabhäuser, Wanda Szmielew, and Alfred Tarski. *Metamathematische Methoden in der Geometrie.* Springer-Verlag, Berlin, 1983.

[Str73] J. Strommer. Über die Kreisaxiome. *Periodica Mathematica Hungarica*, 4 :3–16, 1973.

[Tar51] Alfred Tarski. *A decision method for elementary algebra and geometry.* University of California Press, 1951.

[Web38] Werner Weber. Über die Widersprüche in gewissen linearen Vollständigkeitsaxiomen der Geometrie. *Sitzungsber. Preuß. Akad. Wiss., Phys.-Math. Kl.*, 1938 :376–382, 1938.

Unboxing Mutually Recursive Type Definitions in OCaml

Simon Colin, Rodolphe Lepigre[1], and Gabriel Scherer[2]

[1] Inria, LSV, CNRS, ENS Paris-Saclay, France – rodolphe.lepigre@inria.fr
[2] Inria, France – gabriel.scherer@inria.fr

October 2018

Abstract

In modern OCaml, single-argument datatype declarations (variants with a single constructor, records with a single immutable field) can sometimes be "unboxed". This means that their memory representation is the same as their single argument, omitting an indirection through the variant or record constructor, thus achieving better memory efficiency. However, in the case of generalized/guarded algebraic datatypes (GADTs), unboxing is not always possible due to a subtle assumption about the runtime representation of OCaml values. The current correctness check is incomplete, rejecting many valid definitions, in particular those involving mutually-recursive datatype declarations. In this paper, we explain the notion of *separability* as a semantic for the unboxing criterion, and propose a set of inference rules to check separability. From these inference rules, we derive a new implementation of the unboxing check that properly supports mutually-recursive definitions.

1 Introduction

Version 4.04 of the OCaml programming language, released in November 2016, introduced the possibility to *unbox* single-constructor variants and single-immutable-field records. In other words, a value inhabiting such a datatype is exactly represented at runtime as the value that it contains, rather than as a pointer to a *block* containing a tag for the constructor and the contained value. The removal of this indirection is called *constructor unboxing*, or *unboxing* for short, and it allows for a slight improvement in speed and memory usage.

In the current version of OCaml, unboxing must be explicitly requested with `[@@unboxed]` as shown in the following example:

```
type uid = UId of int [@@unboxed]
```

One of the main interests of unboxing is that it allows the incorporation of semantic type distinctions without losing runtime efficiency – the mythical zero-cost abstraction. For example, the elements of `uid` are distinct from those of `int` for the type checker, but they have the same runtime representation. Unboxing resolves a tension between software engineering and performance.

Unboxing becomes even more interesting when it is combined with advanced features such as existential types, with GADTs, or higher-rank polymorphism, with polymorphic record fields, which are otherwise only accessible in boxed form.

```
type 'a data = { name : string ; data : 'a }
type any_data = Any_data : 'a data -> any_data [@@unboxed]

type proj = { proj : 'a. 'a -> 'a -> 'a } [@@unboxed]
```

1.1 Unboxing and dynamic floating-point checks

OCaml uses a uniform memory representation, one machine word for all values. Multi-word data such as floating-point numbers, records or arrays are represented by a word-sized pointer to a block on the heap.

For local computations involving floating-point numbers, the compiler tries to optimize by storing short-lived floating-point values directly, without the pointer indirection; this is called *floating-point unboxing*, a different form of unboxing optimization. The boxing indirection needs to be kept for values passed across function boundaries, or passed through data structures that expect generic OCaml values.

To avoid indirection costs on typical numeric computations, a specific representation exists for floating-point arrays, in which floating-point numbers are stored unboxed, as two consecutive words in memory. The compiler uses this optimized representation when an array is statically known to have type `float array`. It also performs a dynamic optimization when creating a new non-empty array: it checks whether its first element is a (boxed) floating-point value, and in that case uses the special floating-point array representation. As a consequence, all writes to such an array need to be unboxed first, and reads produce data in already-unboxed form, which meshes well with the local floating-point unboxing optimizations.

However, this optimization crucially relies on an underlying assumption: the inhabitants, the values of any given type are either all boxed floating-point values, or none of them are. We don't know of a standard name for this property, so we call it *separability*.

Indeed, if there were non-separable types, containing both floating-point and non-floating-point values, then this optimization would be unsound. This is demonstrated by the following example, which relies on the unsafe/forbidden casting function `Obj.magic`.

```
let despicable : float array = [| 0.0 ; Obj.magic 42 |]
(* Produces: "segmentation fault (core dumped)" *)
```

The array of floating point numbers that is constructed here is stored using the optimized representation. As a consequence, its elements must be unboxed prior to being inserted into the actual array. Here, the segmentation fault is precisely triggered when attempting to unbox the value `42`, which is not stored in a block but as an immediate memory word: dereferencing it as a float pointer accesses forbidden memory.

Although the above example requires unsafe features, a similar situation also arises when using a single-constructor GADT whose parameter is existentially quantified, and can hence contain either floating-point or non-floating-point values.

```
type any = Any : 'a -> any
```

The above datatype is the GADT formulation of the existential type $\exists\alpha.\,\alpha$: it can contain any OCaml value. However, those values are "boxed" under the `Any` constructor. In particular, the value `Any 0.0` is *not* directly represented as a floating-point value, but as pointer to a block stored on the heap that is tagged with the `Any` constructor, following by (the OCaml representation of) the floating-point number.

If the `any` datatype were allowed to be unboxed, we would have constructed a type breaking the separability assumption. The above example of segmentation fault could then be reproduced by a well-typed program as follows.

```
type any = Any : 'a -> any [@@unboxed]
(* The above type is rightfully rejected by OCaml, it cannot be unboxed. *)

let array = [| Any 0.0 ; Any 42 |]
```

The current implementation correctly rejects this `any` datatype, but it also rejects valid unboxed definitions that would not introduce non-separable types, when mutually-recursive datatypes are involved. The following real-world example of an interesting definition that is wrongly rejected was given by Markus Mottl.[1]

```
type (_, _) tree =
  | Root  : { mutable value : 'a; mutable rank : int } -> ('a, [`root ]) tree
  | Inner : { mutable parent : 'a node }                -> ('a, [`inner]) tree

and _ node = Node : ('a, _) tree -> 'a node [@@ocaml.unboxed]
(* The above type is incorrectly rejected by OCaml, it could be unboxed. *)
```

Yet another example of such wrongful rejection was encountered by the second author, in the context of the Bindlib library (Lepigre and Raffalli, 2018). It resembles the definitions of `any_type` and `'a data` that were given earlier, but it is rejected as the two datatypes are defined mutually-recursively (for more details, see Lepigre and Raffalli, 2018, Section 3.1).

1.2 The existing check

The existing implementation of the compiler check for unboxing was implemented by Damien Doligez in 2016, when constructor unboxing was introduced. It proceeds by inspecting the parameter of every unboxed GADT constructor. If it is an existential variable `'a`, the definition is rejected. If it is a type expression whose value representation is known, such as a function `foo -> bar` or a product `foo * bar`, then it is separable and the definition is accepted. The difficult case arises when the parameter is of the form `(foo, bar, ...) t`. For example, if the parameter has type `'a t`, where `'a` is an existential variable, then this definition must be rejected if `t` is defined as `type 'a t = 'a`, but it can be accepted if it is defined as `type 'a t = int * 'a`, for example.

In the current check, `(foo, bar, ...) t` is expanded according to the definition of `t`, and its expansion is checked recursively. In the case where `t` is an abstract datatype, the check correctly fails as soon as one of the parameter is an existential. If `t` is part of the same block of mutually-recursive definitions, its definition may not be known yet, and the check fails although it could have succeeded had the definition been known. Finally, it is worth noting that because of recursive datatypes, the expansion process may not terminate. As a consequence, there is a hard limit on the number of expansions.[2]

1.3 Our approach: inference rules for separability

We propose to replace the current check using a "type system" for separability. In other words, we introduce inference rules to approximate the semantic notion of separability: a type is separable if the values it contains are all floating-point numbers, or if none of them are. This

[1] https://github.com/ocaml/ocaml/pull/606#issuecomment-248656482
[2] This limit was originally set too high, and it had to be reduced to avoid type-checking slowdowns.

approach is very similar to how the variance of datatype parameters is handled in languages with subtyping, including OCaml, both in theory and in practice. In the case of variance, the "types" are annotations such as `covariant` or `contravariant`, and the "terms" that are being checked are types and datatype definitions.

Here, the "types" are *separability modes* indicating whether the corresponding type parameters need to be separable for the whole datatype to be separable. Modes include `Sep` (separable), the mode of types or parameters that must be separable, and `Ind` (indifferent), the mode of types or parameters on which no separability constraint is imposed.[3] For example, in the case of `type 'a t = 'a * int`, the parameter `'a` has mode `Ind` since the values it contains are all pairs, independently of the type used to instantiate `'a`. For `type ('a, 'b) second = 'b`, the parameter `'a` has mode `Ind`, but the parameter `'b` has mode `Sep`. Indeed, if `'b` is not instantiated with a separable type, then the whole definition cannot be separable either. The separability behavior of a parametrized datatype is characterized by a *mode signature*, which gives a choice of mode for the type parameters that guarantees that the whole datatype will be separable. For instance, the previous two examples would have mode signatures `('a:Ind) t` and `('a:Ind, 'b:Sep) second`.

If a type that is being checked is defined in terms of a type constructor `(foo, bar, ...) t`, and if the mode signature of `t` is known, then its definition does not need to be unfolded as in the legacy implementation. Indeed, it is enough to simply check the parameter instances (`foo`, `bar`, ...) against the modes of the signature. For example, if we encounter `(foo, bar) second`, then we only need to check that `bar` is separable.

In the case of a block of mutually-recursive datatype definitions, a mode signature must be constructed for each definition of the block. However, this cannot be done separately for each definition due to dependencies. As a consequence, we proceed by computing a fixpoint, as for variance: we iteratively refine an approximation of the mode signatures for the block, updating when encountering conflicts, until a mode signature that requires no update is found: it is a valid signature for the block. The number of possible modes assignments for each parameter is finite, so this fixpoint computation always terminates. Note that our inference rules do not talk about the fixpoint computation, or about algorithmic aspects in general. It is a simpler, declarative specification for the correctness of mode assignments, from which the checking algorithm can be derived. It can also be used to reason about the semantic correctness of our check.

1.4 Contributions

We claim the following three contributions:

- A clear explanation of separability. The notion of separability evidently existed in Damien Doligez's mind when unboxed datatypes where implemented in 2016, but we had to rediscover it to understand the check, and we took this opportunity to document separability and to give it a precise semantic definition.
- A set of inference rules for separability of type expressions and datatype definitions.
- An implementation of a separability check derived from these inference rules, within the OCaml compiler. It is compatible with mutually-recursive definitions, which were previously always rejected.

A preliminary version of this work was produced during an internship of the first author (Colin, 2018). We give here a full treatment of GADTs, while the internship report used first-class existential types, without equations. We also moved from a prototype implementation, separate

[3] There is actually a third mode `Deepsep` (deeply separable) that will be explained in the next section.

from the OCaml type-checker and defined on simpler data-structures, to a production-ready implementation of the separability check in (an experimental fork of) the OCaml type-checker.

2 Type language and separability modes

As mentioned in the introduction, our approach to unboxability checking relies on the notion of *separability* of a type, which was already introduced in an intuitive way.

Definition 1 (Separability)**.** *A type is said to be separable if and only if its inhabitant contain either only floating-point values, or only non-floating point values.*

Intuitively, our final goal is to make sure that all type definitions have a separable body. That is, for all definition `type ('a, 'b, ...) t = <type_body>`, we want to check that the type `<type_body>` is separable under some assumptions on the separability of parameters. Before going into formal definitions, two potential sources of difficulties must be discussed: GADTs, which are the only possible source of non-separability, and the (related) typing constraints, which can be used to extract possibly non-separable subcomponents of a type.

2.1 GADTs using type equalities and existential quantifiers

Generalized/guarded algebraic datatypes, GADTs for short extend the usual variant datatypes using a slightly different syntax. They are parametrized datatypes `(_, _, ...) t` in which the typing constraints on the parameters may vary depending on the variant constructor. Moreover, existentially quantified type variables may appear in GADT constructors.[4] Examples of GADTs illustrating these features are given below.

```
type _ data =
  | Char : char -> char data
  | Bool : bool -> bool data

type any_function = Fun : ('a -> 'b) -> any_function

type _ first = First : 'c -> ('c * 'd) first
```

As it turns out, GADTs can be decomposed into more primitive components: non-GADT algebraic datatypes, *type equalities* and *existential types*. For instance, the above examples can be encoded as follows, using an imaginary extension of the OCaml syntax explained below.[5]

```
type 'a data =
  | Char of char with ('a = char)
  | Bool of bool with ('a = bool)

type any_function = Fun of exists 'a 'b. 'a -> 'b
```

[4] See the OCaml manual (https://caml.inria.fr/pub/docs/manual-ocaml-4.07/extn.html#sec252) for a more thorough introduction to GADTs.
We are intentionally unclear about whether the "G" means "generalized" or "guarded", because both words have been used in the literature. In the present article, "guarded" makes more sense as we concentrate on the pains introduced by equality constraints, or equality *guards*. "Generalized" is also a rather empty name, as there are many other ways to generalize algebraic datatypes.

[5] The translation can be defined in a systematic way, see for example Simonet and Pottier (2007).

```
type 'a first = First of exists 'b 'c. 'b with ('a = 'b * 'c)
```

In the above, we use the new syntax `exists 'a. <type_expr>` for first-class existential quantification over one or several type variables, and `<type_expr> with ('a = <type_expr>)` for guarding a type expression with an equality constraint. In our formal description of the separability check, we will use the syntax $\exists\alpha.\tau$ for existentials, and the syntax $\tau\restriction(\alpha = \kappa)$ for equality guards, where α is a type variable, and τ and κ are type expressions.

The semantic intuition behind these two type-formers is the following. An existential type $\exists\alpha.\tau$ can be understood as the union over all types κ of $\tau[\alpha := \kappa]$. For example, the values of the type `exists 'a. 'a -> 'a` contains the values of `int -> int`, but also the values of `bool -> bool` and the values of `(char -> bool) -> (char -> bool)`. [6] An equality guard $\tau\restriction(\alpha = \kappa)$ exactly corresponds to τ in the case where the constraint $(\alpha = \kappa)$ is satisfied, and it is empty otherwise. Note that guards whose left-hand-side is a type variable suffice to express GADTs: all guards equate a type parameter to its instance in the "return type" of the variant constructor.

2.2 Equality guards and deep separability

In the introduction, the parametrized type constructors `'a t` that were considered either had the mode signature `('a : Ind) t`, meaning that `'a t` is always separable no matter what `'a` is, or they had mode signature `('a : Sep) t`, meaning that `'a t` is separable only if `'a` is itself separable. However, these two modes are not always sufficient due to equality guards.

The equality guards used in GADTs[7] introduce the ability for parametrized types to "peek" into the definition of their parameters in ways that affect our separability check. Consider, for example, the unboxed version of the `_ first` datatype, which is accepted by the current constructor unboxing check.

```
type _ first =
  | First : 'b -> ('b * 'c) first [@@unboxed]
```

Using this definition, `('b * int) first` has the same memory representation as `'b`, so it is separable if and only if `'b` is separable. In other words, the separability of type `foo first` does not on the separability of type `foo`, but rather on the separability of a *sub-component*, in the sense of syntactic inclusion, of the type `foo`.

To account for this situation, we introduce a third separability mode `Deepsep` (deeply separable). For a closed type expression (a type expression with no free type variables) to be `Deepsep`, all its sub-components, including the type expression itself, must be separable.

2.3 Formal syntax of types

We define the syntax of the type expressions and datatypes that we consider in the top and middle parts of Figure 1. It is a representative subset of the OCaml grammar of types,[8] with first-class existential types $\exists\alpha.\tau$ and type restrictions $\tau\restriction(\alpha = \kappa)$ to represent GADTs with

[6] The interpretation of existential quantifiers as unions (and dually, of universal quantifiers as intersections) is very common in realizability semantics, for example.

[7] Constrained datatype definitions such as `type 'a t = 'b constraint 'a = 'b * int` may also involve equality guards, but we chose to ignore this feature here since it poses exactly the same problem.

[8] For instance, we omit object types and polymorphic variants, but they could be handled just like products.

Syntax of type expressions:

$$
\begin{array}{lll}
\tau, \kappa ::= & \alpha, \beta & \text{type variable} \\
& \mid \texttt{float} \mid \texttt{int} \mid \texttt{bool} & \text{builtin types} \\
& \mid t(\tau_1, \ldots, \tau_n) & \text{(parametrized) type constructor} \\
& \mid \tau \to \kappa & \text{function type} \\
& \mid \tau_1 \times \cdots \times \tau_n & \text{product/record type} \\
& \mid \forall \alpha.\tau & \text{polymorphic type} \\
& \mid \exists \alpha.\tau & \text{existential type} \\
& \mid \tau \restriction (\alpha = \kappa) & \text{equality guard}
\end{array}
$$

Syntax of datatypes:

$$
\begin{array}{lll}
A, B ::= & C_1 \text{ of } \tau_1 \mid \ldots \mid C_n \text{ of } \tau_n & \text{boxed variant} \\
& \mid C \text{ of } \tau \ [@@\texttt{unboxed}] & \text{unboxed variant} \\
& \mid \{\texttt{mutable? } l_1 : \tau_1 \ ; \ \ldots \ ; \ \texttt{mutable? } l_n : \tau_n\} & \text{boxed record} \\
& \mid \{l : \tau\} \ [@@\texttt{unboxed}] & \text{unboxed record} \\
& \mid \tau & \text{type synonym}
\end{array}
$$

Sub-component relation on type expressions:

$$
\frac{}{\tau \lhd \tau} \qquad \frac{\tau_1 \lhd \tau_2 \qquad \tau_2 \lhd \tau_3}{\tau_1 \lhd \tau_3}
$$

$$
\frac{i \in [1; n]}{\tau_i \lhd t(\tau_1, \ldots, \tau_n)} \qquad \frac{1 \in \{1, 2\}}{\tau_i \lhd \tau_1 \to \tau_2} \qquad \frac{i \in [1; n]}{\tau_i \lhd \tau_1 \times \cdots \times \tau_n}
$$

$$
\frac{\tau \lhd \kappa \qquad \alpha \notin \tau}{\tau \lhd \forall \alpha.\kappa} \qquad \frac{\tau \lhd \kappa \qquad \alpha \notin \tau}{\tau \lhd \exists \alpha.\kappa} \qquad \frac{i \in \{1, 2\}}{\tau_i \lhd (\tau_2 \restriction (\alpha = \tau_1))}
$$

Figure 1: Syntax of type expressions, datatypes, and sub-component relation.

finer-grained rules, as explained in Section 2.1. We also include first-class universal types $\forall \alpha.\tau$, although they are only allowed in record or method fields in OCaml.

To define deep separability, we first need to define the syntactic sub-components of a type (see Section 2.2). We define a sub-component relation $\tau \lhd \kappa$ ("τ is a sub-component of κ") at the bottom of Figure 1. The first two rules make the relation reflexive and transitive, and the others give the immediate sub-components for each type-former.

The definition of the sub-component relation is careful to preserve the scoping of variables. For example, $\tau \lhd \forall \alpha.\kappa$ holds only if α does not occur free in τ, which we write $\alpha \notin \tau$. In particular, whenever $\tau \lhd \kappa$ holds, the free type variables of τ are included in those of κ; the sub-components of a closed type expression are also closed types.

2.4 Separability modes

Separability modes m, n, their order structure $m < n$ and mode composition $m \circ n$ are defined in Figure 2. Modes are totally ordered from less to more demanding. We may use derived notations such as $m \leqslant n$, $\min(m, n)$ or $\max(m, n)$ for the non-strict ordering, the minimum or

Separability modes (or simply, modes):

$$m, n ::= \texttt{Ind} \mid \texttt{Sep} \mid \texttt{Deepsep}$$

`Ind`	indifferent
`Sep`	separable
`Deepsep`	deeply separable

Mode composition $m \circ n$:

$$\begin{aligned} \texttt{Ind} \circ m &:= \texttt{Ind} \\ \texttt{Sep} \circ m &:= m \\ \texttt{Deepsep} \circ m &:= \texttt{Deepsep} \end{aligned}$$

Order structure:
$\texttt{Ind} < \texttt{Sep} < \texttt{Deepsep}$

Figure 2: Separability modes and mode operations.

the maximum of two modes.

A mode m expresses a requirement on a type expression, which comes from its context: for the whole expression to be valid, some of its sub-components must have the separability property m. The operation $m \circ n$ expresses composition of those requirements in our inference rules (given in the next section). For example, if $t(\alpha)$ requires its parameter α to have mode m for the whole expression to be separable, and $u(\beta)$ requires its parameter β to have mode n for the whole expression to be separable, then $t(u(\beta))$ is separable when β has mode $m \circ n$.

2.5 Contexts and mode signatures

Our separability judgments in Section 3 make use of contexts Γ, representing separability assumptions on the type variables $(\alpha, \beta \dots)$ that are in scope. Moreover, we also rely on mode signatures Σ, representing the separability requirements on the datatype constructors $t(\overline{\alpha})$ that are available in the scope. Contexts and mode signatures are defined in Figure 3.

$$\Gamma ::= \varnothing \mid \Gamma, \alpha : m \qquad \Sigma ::= \varnothing \mid \Sigma, t(\overline{\alpha : m})$$

$$\frac{\Gamma \leqslant \Gamma' \qquad m \leqslant m'}{\Gamma, \alpha : m \leqslant \Gamma', \alpha : m'} \qquad \frac{\Sigma \leqslant \Sigma' \qquad \overline{\alpha} = \overline{\alpha'} \qquad \overline{m \geqslant m'}}{\Sigma, t(\overline{\alpha : m}) \leqslant \Sigma', t(\overline{\alpha' : m'})}$$

Figure 3: Contexts, mode signatures, and their order

This figure also defines the extension of the order on modes to an order on contexts and on mode signatures. The order on contexts is just a point-wise extension of the mode order, imposing that two comparable contexts have the same type variables. The order on signatures imposes the same datatype constructors on both sides, but requires their parameter modes to be pointwise comparable in the opposite order, $(\geqslant)$ rather than $(\leqslant)$: parameters are in contravariant position. For example, we have $t(\alpha : \texttt{Sep}, \beta : \texttt{Sep}) \leqslant t(\alpha : \texttt{Ind}, \beta : \texttt{Ind})$.

3 Separability inference

The formal deduction rules that we will use to assess the separability of datatype definitions are defined in Figure 4. Type expressions are checked by judgments of the form $\Sigma; \Gamma \vdash \tau : m$, which can intuitively be read in either direction, from Γ to m or conversely:

- If the type variables respect the modes in Γ, then the type expression τ has the mode m.

- For the type expression τ to be safe at mode m, then its type variables need to have at least the modes given in Γ.

Inference rules at type-expression level:

$$\frac{(\alpha : m) \in \Gamma}{\Sigma; \Gamma \vdash \alpha : m} \qquad \frac{t(\alpha_1 : m_1, \ldots, \alpha_n : m_n) \in \Sigma \qquad (\Sigma; \Gamma \vdash \tau_i : m \circ m_i)_{1 \leqslant i \leqslant n}}{\Sigma; \Gamma \vdash t(\tau_1, \ldots, \tau_n) : m}$$

$$\frac{\Sigma; \Gamma \vdash \tau : m \circ \texttt{Ind} \qquad \Sigma; \Gamma \vdash \kappa : m \circ \texttt{Ind}}{\Sigma; \Gamma \vdash \tau \to \kappa : m} \qquad \frac{(\Sigma; \Gamma \vdash \tau_i : m \circ \texttt{Ind})_{1 \leqslant i \leqslant n}}{\Sigma; \Gamma \vdash \tau_1 \times \cdots \times \tau_n : m}$$

$$\frac{\Sigma; \Gamma, \alpha : n \vdash \tau : m}{\Sigma; \Gamma \vdash \forall \alpha . \tau : m} \qquad \frac{\Sigma; \Gamma, \alpha : \texttt{Ind} \vdash \tau : m}{\Sigma; \Gamma \vdash \exists \alpha . \tau : m} \qquad \frac{\Sigma; \Gamma \vdash \tau : m \qquad m \geqslant n}{\Sigma; \Gamma \vdash \tau : n}$$

$$\frac{\forall \Gamma' \geqslant \Gamma, \quad \Sigma; \Gamma' \vdash (\kappa_1 = \kappa_2) \implies \Sigma; \Gamma' \vdash \tau : m}{\Sigma; \Gamma \vdash \tau \restriction (\kappa_1 = \kappa_2) : m} \qquad \frac{\forall m, \quad \Sigma; \Gamma \vdash \tau_1 : m \iff \Sigma; \Gamma \vdash \tau_2 : m}{\Sigma; \Gamma \vdash (\tau_1 = \tau_2)}$$

Inference rules at datatype level:

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{decl}} C_1 \text{ of } \tau_1 \mid \ldots \mid C_n \text{ of } \tau_n} \qquad \frac{\Sigma; \Gamma \vdash \tau : \texttt{Sep}}{\Sigma; \Gamma \vdash_{\mathsf{decl}} C \text{ of } \tau \ [@@\texttt{unboxed}]}$$

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{decl}} \{l_1 : \tau_1 \ ; \ \ldots \ ; \ l_n : \tau_n\}} \qquad \frac{\Sigma; \Gamma \vdash \tau : \texttt{Sep}}{\Sigma; \Gamma \vdash_{\mathsf{decl}} \{l : \tau\} \ [@@\texttt{unboxed}]} \qquad \frac{\Sigma; \Gamma \vdash \tau : \texttt{Sep}}{\Sigma; \Gamma \vdash_{\mathsf{decl}} \tau}$$

Inference rule at the datatype declaration block level:

$$\frac{\exists (\overline{m_i})_{1 \leqslant i \leqslant n}, \ \Sigma_{\mathsf{block}} := t_1(\overline{\alpha_1} : \overline{m_1}), \ldots, t_n(\overline{\alpha_n} : \overline{m_n}) \qquad (\Sigma_{\mathsf{env}}, \Sigma_{\mathsf{block}}; \overline{\alpha_i} : \overline{m_i} \vdash_{\mathsf{decl}} A_i)_{1 \leqslant i \leqslant n}}{\Sigma_{\mathsf{env}} \vdash (t_i(\overline{\alpha_i}) := A_i)_{1 \leqslant i \leqslant n} \dashv \Sigma_{\mathsf{block}}}$$

Figure 4: Inference rules for separability.

Type expressions. The rule for parametrized datatypes $t(\tau_1, \ldots, \tau_m)$ uses mode composition; for example, if t is a one-argument type constructor with signature $t(\alpha : n)$ and we want $t(\tau)$ to have mode m, then τ needs to have the mode $m \circ n$.

The rules for concrete datatypes (functions and products, but also most other OCaml type-formers if we were to add them) use $m \circ \texttt{Ind}$ on their arguments. If m is `Sep` or `Ind`, then $m \circ \texttt{Ind}$ is `Ind`, which corresponds to not requiring anything of the sub-components: the values at type $\tau_1 \to \tau_2$ are all functions, never floats, regardless of τ_i. If m is `Deepsep`, then we do need to check the sub-components, and indeed $\texttt{Deepsep} \circ \texttt{Ind}$ is `Deepsep`.

A value is at the universal type $\forall \alpha . \tau$ only if it belongs to all the $\tau[\kappa/\alpha]$ for all κ. In particular, it is enough to prove the separability at just one of these $\tau[\kappa/\alpha]$, the universal type has even less values, so we can assume the arbitrary mode n of this particular κ by adding $\alpha : n$ in the context. Conversely, $\exists \alpha . \tau$ is inhabited by all the $\tau[\kappa/\alpha]$, so τ has to have the desired mode for all possible modes of κ. Instead of requiring the premise to hold for all possible modes n, we equivalently ask for the most demanding mode `Ind`.

The conversion rule allows to forget some information about a type $\tau : m$ by exporting it at a smaller mode $n \leqslant m$. For example, all `Sep` types are also `Ind` types.

Equality constraints. The rule for equality constraints is the most complex rule in the system. A first remark is that $\kappa\restriction(\tau_1 = \tau_2)$ always has less elements than κ: it has the same elements when the equality holds, or no elements otherwise. In particular, if $\kappa : m$ holds in the current context Γ, then $\kappa\restriction(\tau_1 = \tau_2) : m$ should also hold in Γ.

When we see an equality constraint, we gain more information, which should allow us to mode-check more types. The way our rules represent this information gain is by moving from the current mode context Γ to a stronger mode context Γ'. More precisely, to check $\kappa\restriction(\tau_1 = \tau_2) : m$ in Γ, the rule asks to check $\kappa : m$ in *any* context $\Gamma' \geqslant \Gamma$ that is consistent with the assumption $(\tau_1 = \tau_2)$. This corresponds to the $\Gamma' \vdash (\tau_1 = \tau_2)$ hypothesis, which will be explained shortly. For example, if Γ is $\alpha : \texttt{Ind}, \beta : \texttt{Sep}$, and we observe the equality $(\alpha = \beta)$, then in particular we know that $\alpha : \texttt{Sep}$ also holds: if the two types are equal, they must have the same mode. Our rule will type-check the body type κ in stronger contexts Γ' with $\alpha : \texttt{Sep}, \beta : \texttt{Sep}$.

A context Γ is valid for a type equality, written $\Gamma \vdash (\tau_1 = \tau_2)$, if the two types τ_1 and τ_2 have exactly the same mode in Γ. Remark that it is not enough to ask that, for a given mode m, both types have mode m; for example, all types trivially have mode `Ind`. Instead, we ask that for *any* mode m, either τ_1 and τ_2 have mode m, or neither of them have it. This is equivalent to requiring τ_1 and τ_2 to have the same "best", maximal mode.

In our example where Γ is $\alpha : \texttt{Ind}, \beta : \texttt{Sep}$, note that we do not have $\Gamma \vdash (\alpha = \beta)$: β has mode `Sep` but α does not. The modes $\Gamma' \geqslant \Gamma$ that satisfy $\Gamma' \vdash (\alpha = \beta)$ are $\Gamma_{\texttt{Sep}} := \alpha : \texttt{Sep}, \beta : \texttt{Sep}$, and $\Gamma_{\texttt{Deepsep}} := \alpha : \texttt{Deepsep}, \beta : \texttt{Deepsep}$. To check that $\kappa\restriction(\alpha = \beta) : m$ holds in Γ, our rule asks us to check that $\kappa : m$ holds in both $\Gamma_{\texttt{Sep}}$ and $\Gamma_{\texttt{Deepsep}}$. But in fact we, the implementers of the checking algorithm, know that making stronger assumptions in the context makes more mode-checks pass, so it suffices to check in context $\Gamma_{\texttt{Sep}}$.[9]

Finally, it is interesting to consider the derivation of the following judgment, which represents in our system the key ingredient of the `_ first` GADT that led to the introduction of `Deepsep` in Section 2.2.

$$\alpha : m \vdash \exists \beta\gamma.\, \beta\restriction(\alpha = \beta \times \gamma) : \texttt{Sep}$$

We expect to accept this type declaration only in the case where $\alpha : \texttt{Deepsep}$, this assumption guaranteeing that β will be separable. Opening the existentials puts β, γ in the context at `Ind`:

$$\alpha : m, \beta : \texttt{Ind}, \gamma : \texttt{Ind} \vdash \beta\restriction(\alpha = \beta \times \gamma) : \texttt{Sep}$$

Let us now reason by case analysis on m to show that only $m = \texttt{Deepsep}$ has a valid derivation of this judgment.

- If m is `Deepsep`, then any $\Gamma' \geqslant \Gamma$ with $\Gamma' \vdash (\alpha = \beta \times \gamma)$ has $\beta : \texttt{Deepsep}, \gamma : \texttt{Deepsep}$ since otherwise $\beta \times \gamma : \texttt{Deepsep}$ cannot hold. In particular, $\Gamma' \vdash \beta : \texttt{Sep}$ holds as expected, so the judgment is derivable.
- If m is `Sep` or `Ind`, then $\Gamma' := \alpha : \texttt{Sep}, \beta : \texttt{Ind}, \gamma : \texttt{Ind}$ satisfies $\Gamma' \geqslant \Gamma$ and $\Gamma' \vdash (\alpha = \beta \times \gamma)$, but we do not have $\Gamma' \vdash \beta : \texttt{Sep}$: the judgment is not derivable.

Datatypes. The judgment for datatypes is simple: a datatype is accepted if its set of values is separable. Boxed variant or record definitions are always separable, so no premises are required. Unboxed variants/records with parameter type τ, or type synonyms for τ require $\tau : \texttt{Sep}$.

Definition blocks. The judgment for definition blocks has an input signature, $\Sigma_{\texttt{env}}$ in the rule, which lists assumptions that we make on the datatype constructors provided by the typing

[9] We could formulate the rule to only check the unique minimal context Γ', but Fact 2 in Section 4 suggests that such a unique context may not always exist.

environment, and an output signature, Σ_{block} in the rule, which is a valid signature for the current block of mutually-recursive definitions. A definition block is valid if each datatype definition it contains is valid. Note that each definition is checked with the full Σ_{block} in the signature context: all mutually-recursive datatype constructors are available in scope.

Meta-theory. The following two lemmas can be easily proved by structural induction on typing derivations.

Lemma 1 (Cut elimination)**.** *A separability derivation can always be rewritten so that all occurrences of the conversion rule only have axiom rules as their premises or, equivalently, using the following more primitive rule.*

$$\frac{(\alpha : m) \in \Gamma \qquad m \geqslant n}{\Sigma; \Gamma \vdash \alpha : n}$$

Lemma 2 (Monotonicity)**.** *The following rule is admissible.*

$$\frac{\Sigma \leqslant \Sigma' \qquad \Gamma \leqslant \Gamma' \qquad \Sigma; \Gamma \vdash \tau : m \qquad m \geqslant m'}{\Sigma'; \Gamma' \vdash \tau : m'}$$

4 Semantics

Due to space limitations, we moved our presentation of separability semantics to Appendix A. It contains a precise semantic characterization of separability judgments in the flavor of set-theoretic or realizability models, and a discussion of soundness, principality and completeness. Many results are left as conjectures – we explain why some of them are surprisingly delicate.

5 Integration into OCaml

We are now going to highlight some important points of our implementation in the OCaml type-checker. The corresponding code has been proposed for integration through a GitHub pull request, that is visible at the following URL.

https://github.com/ocaml/ocaml/pull/2188

Our implementation is derived from the type system given in Section 3 by inferring mode signatures for type definitions. It only assigns mode signatures that can be justified by our syntactic type system. For example, if `('a : Sep, 'b : Ind) t` is assigned to a type definition `type ('a, 'b) t = A`, then $\Sigma_{\mathsf{env}} \vdash t(\alpha, \beta) := A \dashv t(\alpha : \texttt{Sep}, \beta : \texttt{Ind})$ must be derivable using the inference rules of Figure 4.

5.1 Inferring block signatures with a fixpoint

Our main checking function constructs a block signature Σ_{block} given an environment Σ_{env} and a block of type definitions $(t_i(\overline{\alpha_i}) := A_i)_{1 \leqslant i \leqslant n}$:

```
val check : Env.t -> type_definition ConstrMap.t -> mode_signature ConstrMap.t
```

In this signature, `type_definition ConstrMap.t` maps datatype constructors $t_i(\overline{\alpha})$ to datatype definitions A_i, and `mode_signature ConstrMap.t` them to a mode signature $t_i(\overline{\alpha : m})$.

It is not possible to directly compute mode signatures for mode definitions, due to recursive types: we need to know the mode signature of the type constructors in order to assign them a mode signature. This circularity is solved by using a fixpoint computation: we iteratively refine an approximation of the mode signatures.

The fixpoint computation starts with the most permissive mode signature, which only requires mode `Ind` for the variables of every type constructor of the block. At each iteration, the separability of every type of the block is checked against the current approximation of the mode signatures, accumulating more precise constraints whenever required. If the mode signatures coming from these constraints are more demanding than those of the current approximation, we define them as the new approximation and continue. Otherwise, we have found a mode signature that validates the judgment – it is sound. In fact, it is the most permissive mode signature that we can find by iteration in this way.

5.2 Management of GADTs

As explained in Section 2.1, our type system for separability does not directly account for GADTs: they are encoded using existential quantifiers and equality guards. Our implementation handles only GADTs, which correspond to a very specific mode of use of existentials and guards within type declarations.

Consider, for example, `type 'a fun = Fun : ('b -> 'c) -> ('b -> 'c) fun`. The return type `('b -> 'c) fun` determines the equality guard ($'\mathtt{a} = '\mathtt{b} \to '\mathtt{c}$), and the existential types are the free type variables $'\mathtt{b}$, $'\mathtt{c}$ of the declaration. In general, unboxable GADT types contain existential quantifiers only at the toplevel, immediately followed by one equation for each type parameter, with finally a concrete parameter type τ.

The first thing to note is that all existential quantifiers occurs at the top level: they exactly correspond to the free variables of the parameter type. We can infer a mode signature for the parameter type τ and, following our rule for existentials, check that the modes inferred for the free existential variables in τ are `Ind` and fail otherwise. Finally, the mode signature for the definition of the GADT is obtained by removing the existential variables from the inferred mode signature.

The delicate matter with in handling of GADTs is unsurprisingly the management of equality guards. Recall that an unboxed GADT $\mathtt{type}\ t(\overline{\alpha}) = \mathtt{K} : \tau \to t(\overline{\kappa})$ is encoded as $\exists\overline{\beta}.\tau\restriction(\overline{\alpha = \kappa})$, where the $\overline{\beta}$ are the type variables free in $\tau, \overline{\kappa}$ and the $\overline{\kappa}$ do not contain any α_i. The idea of the implementation is to first infer a mode context Γ such that $\Sigma; \Gamma \vdash \tau : \mathtt{Sep}$, assigning modes to the variables in $\overline{\alpha}$ and $\overline{\beta}$. Equations are then discharged one by one, refining Γ in the process, before the existential variables $\overline{\beta}$ are checked to be `Ind` and eliminated to create the final context for the GADT parameters only – the mode signature. Every equation, taken in any order, is managed according to the following three cases:

- An equation of the form $(\alpha_i = \beta_j)$ leads to Γ being updated with $\{\alpha_i \mapsto \Gamma(\beta_j), \beta_j \mapsto \mathtt{Ind}\}$.
- An equation of the form $(\alpha_i = \kappa)$ with $\Gamma \cap FV(\kappa)$ only containing `Ind` leads to the equation being discarded without any update to Γ.
- Any other equation (i.e., equations $(\alpha_i = \kappa)$ with $\Gamma \cap FV(\kappa)$ not only containing `Ind`) leads to Γ being updated with $\{\alpha_i \mapsto \mathtt{Deepsep}, FV(\kappa) \mapsto \mathtt{Ind}\}$.

These transformations respect the following invariant: if the equation $(\alpha = \kappa)$ updates Γ into Γ_0, then any $\Gamma' \geqslant \Gamma_0$ that satisfies $\Gamma' \vdash (\alpha = \kappa)$ is above the original Γ ($\Gamma' \geqslant \Gamma$). In other words, strengthening the resulting context Γ_0 with the equation we just handled would give a context as permissive or more than the original context Γ.

5.3 Cyclic types

The OCaml type system accepts equi-recursive types. By default, any cycle in types must go through a polymorphic variant or object type constructor, but the `-rectypes` option generalizes this to any ground type constructor. A type such as `('a -> 'b) as 'b`, for example, gives a cyclic representation to the infinite type `'a -> 'a -> 'a -> ...`, and must be supported by our implementation.

To support cyclic types, we extend our checking rules to support a form of coinduction: each time we reduce a judgment to simpler premises (for example from $\Gamma \vdash \tau_1 \to \tau_2 : m$ to $\Gamma \vdash \tau_i : m \circ \texttt{Ind}$), we record in the premises that we have previously encountered the judgment $\tau_1 \to \tau_2 : m$. If, later, we encounter the same judgment to prove, we terminate immediately with a success.

With this rule, it is possible to prove both

$$\alpha : \texttt{Ind} \vdash ((\alpha \to \beta)\ \texttt{as}\ \beta) : \texttt{Sep} \qquad\qquad \alpha : \texttt{Deepsep} \vdash ((\alpha \to \beta)\ \texttt{as}\ \beta) : \texttt{Deepsep}$$

Informally, the reason why recursive occurrences of the same judgment can be considered an immediate success is that we "made progress" between the first occurrence of the cyclic type β and its second occurrence in $\alpha \to \beta$: the second occurrence is "guarded" under a value constructor, and the set of values of β we are classifying as separable, or deeply separable, is not the one we started from – that would be an invalid cyclic reasoning – but a copy of it occurring deeper in the type structure.

However, some cyclic types such as $t(\alpha)\ \texttt{as}\ \alpha$ have a less clear status, as the recursive occurrence is not guarded under a computational type constructor (arrow, product...), but under an abbreviation. Is this type well-defined if, for example, $t(\alpha)$ is defined as $t(\alpha) := \alpha$? OCaml rejects some of these dubious-looking circular types, but instead of trusting our fate to the rest of the type-checker we decided to account for them in our theory. We split our set of co-inductive hypotheses (the list of judgments that we are trying to prove) into a set of "safe" hypotheses Θ_{safe}, which can be used immediately, and a set of "unsafe" hypotheses, which can only be used after a computation type constructor has been traversed.

Here are some rules of the corresponding formal system, extended with coinductive hypotheses, which guided our OCaml implementation:

$$\frac{\forall i \in \{1,2\}, \quad \Sigma; \Gamma; \Theta_{\mathsf{safe}}, \Theta_{\mathsf{unsafe}}, (\tau_1 \to \tau_2 : m); \varnothing \vdash \tau_i : m \circ \texttt{Ind}}{\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}} \vdash \tau_1 \to \tau_2 : m}$$

$$\frac{t(\alpha_1 : m_1, \ldots, \alpha_n : m_n) \in \Sigma \qquad (\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}}, (t(\tau_1, \ldots, \tau_n) : m) \vdash \tau_i : m \circ m_i)_{1 \leqslant i \leqslant n}}{\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}} \vdash t(\tau_1, \ldots, \tau_n) : m}$$

$$\frac{(\tau : m') \in \Theta_{\mathsf{safe}} \qquad m' \geqslant m}{\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}} \vdash \tau : m} \qquad\qquad \frac{(\tau : m) \in \Theta_{\mathsf{unsafe}}}{\Sigma; \Gamma; \Theta_{\mathsf{safe}}; \Theta_{\mathsf{unsafe}} \not\vdash \tau : m}$$

The arrow rule has its premises under a computational type constructor, so it passes all coinductive hypotheses, including the new assumption on $\tau_1 \to \tau_2$, to its safe set. A datatype constructor may be just an abbreviation, so it adds the new hypothesis to the unsafe set.

Finally, whenever a judgment needs to be proved, it immediately succeeds if a stronger hypothesis is in the safe set. On the other hand, if our hypothesis is already in the unsafe set, then we know that it cannot be proven without recursively assuming itself, and we can in fact fail with an error.

5.4 Non-conservativity

Section A.2, Fact 2 shows that our modes do not admit principal judgments – in particular our judgments are non-principal. In particular, the following OCaml declaration can be given either signatures $t(\alpha : \texttt{Ind}, \beta : \texttt{Sep})$ and $t(\alpha : \texttt{Sep}, \beta : \texttt{Ind})$.

```
type ('a, 'b) t = K : 'c -> ('c, 'c) t
```

Our implementation will choose one of the two minimum signatures, depending on the order in which constraints are handled – see Section 5.2. This means that some correct uses of the type will be disallowed: no matter which one is chosen, one of the two following declarations will be rejected:

```
type t1 = T1 : (int, 'a) t -> t1
type t2 = T2 : ('a, int) t -> t2
```

On the other hand, the current implementation, which expands the definition of `t`, accepts both definitions.

We have decided to accept these completeness regressions. Our implementation is cleaner, accepts important examples rejected by the current implementation, and is safer in its handling of cycles, without fuel. In contrast, the counter-examples we could build are fairly esoteric, and we have not found any of them in the current testsuite or user programs. Only time will tell if this assumption is reasonable; we discuss ideas to recover principality (and accept both `t1` and `t2`) in Future Work Section 6.1.

6 Related and future work

6.1 Future work

Richer modes The implementation of our unboxability check is satisfactory in the sense that it accepts most valid (unboxed) definitions. There is however room for improvement, especially in the handling of type equality guards, be they in GADTs or in toplevel equality constraints.

For instance, the way we handle equations of GADTS (see the previous section) is in some sense incomplete, and our language of modes is not principal. We considered extending our modes with modes of the form $\exists(\beta : m).\, m'$ and $m \restriction (\alpha = \tau)$. However, this would significantly increase the complexity of the theory and the implementation, only to handle corner cases that may not be worth it.

Another simpler approach to regain an impression of principality would be to support disjunctions of modes. In the problematic example $t(\alpha, \beta) := \beta \restriction (\alpha = \beta)$, there are two minimum modes $t(\alpha : \texttt{Ind}, \beta : \texttt{Sep})$ and $t(\alpha : \texttt{Sep}, \beta : \texttt{Ind})$, so this type could be given the principal mode $t(\alpha : \texttt{Ind}, \beta : \texttt{Sep}) \vee (\alpha : \texttt{Sep}, \beta : \texttt{Ind})$.

Automatic unboxing Currently, unboxable type declarations are never unboxed automatically, the user has to explicitly ask for it. Automatic unboxing has been considered, but it could break existing code using the foreign function interface. For example, a C function receiving an OCaml value from an unboxable but non-unboxed type currently needs to unbox it explicitly. The same action on an (automatically) unboxed type would fail if the C code is not changed accordingly.

Disjoint GADT unboxing One could wish to unbox multi-constructors GADTs in the case where typing constraints lead to the mutual exclusion of the different constructors.

```
type _ value =
  | Int : int -> int value
  | Bool : bool -> bool value
  | Pair : ('b * 'c) -> ('b * 'c) value
```

While each ground instance of this value type has a single possible constructor and could be unboxed, pattern-matching on an `'a value` would then have to be disallowed: pattern-matching learns the value of `'a` from inspecting the GADT constructor, which is not present anymore in the unboxed representation. It seems fishy to only allow pattern-matching on partially-determined instances of the type, and we did not investigate further.

6.2 Just get rid of the damn float thing

The fairly elaborate sufferings we just went through are caused by the dynamic unboxing optimization for arrays of floating point numbers. If this dynamic optimization were removed, we would not need separability anymore and the unboxing check could also be removed. This dynamic check also has consequences on other features: by making a $\top$ type (our $\exists\alpha.\,\alpha$) unsound, it prevents extending the relaxed value restriction to generalize contravariant variables (Garrigue, 2004, page 11).

There is an ongoing debate in the OCaml community on this idea. An experimental configure flag `-no-flat-float-array` can be set to disable dynamic flat representation optimizations in the implementation, and benefit from the simpler type theory. Since 4.06 (November 2017), a new primitive monomorphic type `floatarray` exists that is specialized for unboxed float arrays, and can be used by users intending to use this optimization, but it lacks library support and convenient array-indexing notations. The problem is with generic code, written against parametric `'a array` value and then applied in numeric programs on float array, whose performance would be silently degraded with an important slowdown. In other terms, removing the dynamic optimization would be acceptable for experts authors of numerical code willing to modify their codebase, but hurt the performance of programs written naively by beginners.

6.3 Related work

We discussed the existing implementation of the unboxability check in Section 1.2.

The approach presented in our work is largely inspired from the way the variance of type declarations is handled in languages with subtyping (Abel, 2008; Scherer and Rémy, 2013).

The memory representation of values used in OCaml and similar languages finds its origin in Lisp-like languages (Leroy, 1990). Despite the advantage of allowing every value to have the same, single-word representation, this approach also has obvious limitations in terms of performances due to the introduction of indirections. As a consequence, ways of lowering this overhead in certain scenarios have been investigated, one possibility being to mix tagged and untagged representations (Peterson, 1989; Leroy, 1990, 1992). Another idea that has been investigated is to consider unboxed values as first-class citizens, although distinguished by their types (Peyton Jones and Launchbury, 1991).

References

Andreas Abel. Polarised subtyping for sized types. *Mathematical Structures in Computer Science*, 2008.

Simon Colin. Specifying the unboxability check on mutually recursive datatypes in OCaml, 2018. Internship report (under the supervision of Gabriel Scherer).

Jacques Garrigue. Relaxing the value restriction. In *FLOPS*, 2004.

Rodolphe Lepigre and Christophe Raffalli. Abstract representation of binders in ocaml using the bindlib library. In *LFMTP*, 2018.

Xavier Leroy. Efficient data representation in polymorphic languages. In *PLILP*, 1990.

Xavier Leroy. Unboxed objects and polymorphic typing. In *POPL*, 1992.

John Peterson. Untagged data in tagged environments: Choosing optimal representations at compile time. In *FPCA*, 1989.

Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *FPCA*, 1991.

Gabriel Scherer and Didier Rémy. Gadts meet subtyping. In *ESOP*, 2013.

Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *TOPLAS*, 2007.

A Semantics

We give a semantic model of types, datatypes and modes in Figure 5. The general idea is to be able to interpret the judgment $\Gamma \vdash \tau : m$ as follows: "if we replace each type variable $(\alpha : m_\alpha)$ of Γ by a closed/ground type with the correct separability mode m_α, then τ will really have the separability mode m".

Ode to semantics. In Section 3 we have given a set of inference rules to prove judgments of the form $\Sigma; \Gamma \vdash \tau : m$, guided by the intuition that this *should* provide evidence that τ is separable. Inference rules are our key contribution as they can easily be turned into a checking or inference algorithm, but they are also subtle, and may very well be wrong. They are hard to audit by someone else who does not trust us, the authors: there are a lot of details to check.

The point of a semantics is to provide an "obvious" counterpart to inference rules. A formal definition of separability, or whatever property an inference system is trying to capture, that is self-evident, can be easily checked and trusted by other people – in particular, it does not depend on the previous inference rules in any way. It serves as a specification, can use arbitrary mathematical operations, and does not need to be computable or close to an algorithm. Finally, one wishes to prove that the syntactic inference rules and the semantics coincide in some sense; this implies that we can trust the inference rules as much as we trust the semantics.

Our semantics can also be understood as an idealized "model" of the programming language we are studying: it contains the assumptions that we make about the language and type system, and can be compared to OCaml to ensure that those assumptions are correct.

Ground value semantics. To define what it means for a closed type $\underline{\tau}$ to have the mode m, we use the set-theoretic intuition introduced earlier: "its values are all floating-point numbers, or none of them are". To do so formally, we introduce a syntax of closed/ground values, and specify which ground values inhabit which ground types.

Our notion of ground/closed values is idealized; in particular, we represent all values at a function type as an opaque `function` blob, as if all functions were represented in the same way in the programming languages we model. They are not, but the differences in representation are irrelevant to reason about separability.

In the figure we define ground values $\underline{v}$ (just data, no term variables), ground types $\underline{\tau}$ (no free type variables), ground datatypes (no free type variables or parameters), and blocks of datatype definitions σ, which assign a datatype to each datatype constructor of a signature Σ.

Finally, we define a series of semantic judgments using the symbol $(\models)$. The judgment $\underline{v} \models^\sigma \underline{\tau}$ specifies when a value inhabits a type, relative to a block of definitions σ to interpret type constructors. The judgment $\underline{v} \models^\sigma \underline{A}$ specifies when a value inhabits a type parameter; note in particular how the judgments for `unboxed` datatypes reflect what happens in an implementation.

The judgments $\underline{\tau} \models^\sigma m$ and $\underline{A} \models^\sigma m$ specify when a type expression or datatype respect the separability mode m. They are defined in terms of our set-theoretic characterizations, separable(X). Finally, $A \models^\sigma t(\overline{\alpha : m})$ specifies when a parametrized datatype A respects a mode signature, and $\sigma \models \Sigma$ specifies that a definition block respects a signature block.

In Figure 6, we use these specifications to build a semantic counterpart for each of our syntactic judgments. This lets us easily formulate soundness and partial completeness results. For example, $\Sigma; \Gamma \models \tau : m$ is the semantic counterpart of the judgment $\Sigma; \Gamma \vdash \tau : m$. It captures what it means for the judgment to hold: for any valid definitions $\sigma \models \Sigma$, and any choice of ground types $\underline{\gamma}$ valid for Γ ($\underline{\gamma} \models^\sigma \Gamma$), replacing the variables in τ by the ground types in $\underline{\gamma}$ gives a ground type $\underline{\gamma(\tau)}$ at mode m, that is, $\underline{\gamma(\tau)} \models^\sigma$ holds.

Ground/closed values $\underline{v}$, type expressions $\underline{\tau}$, datatypes $\underline{A}$,
context valuations $\underline{\gamma}$, datatype signature valuations σ

$$
\begin{array}{lll}
\underline{v} ::= & \texttt{true}, \texttt{false} & \text{booleans} \\
| & \text{int}(n \in \mathbb{N}) & \text{integers} \\
| & \text{float}(x \in \mathbb{R}) & \text{float} \\
| & (\underline{v_1}, \ldots, \underline{v_n}) & \text{tuple} \\
| & \texttt{function} & \text{function} \\
| & \{l_1 : \underline{v_1}; \ldots; l_n : \underline{v_n}\} & \text{record} \\
| & C_i\ \underline{v} & \text{variant}
\end{array}
$$

$$
\begin{array}{rcl}
\text{closed}(\tau) & := & \forall \alpha,\ \alpha \notin \tau \\
\mathsf{GType} & := & \{\underline{\tau} \mid \text{closed}(\tau)\} \\
\\
\text{closed}(A) & := & \forall \tau \in A,\ \text{closed}(\tau) \\
\mathsf{GDatatype} & := & \{\underline{A} \mid \text{closed}(A)\}
\end{array}
$$

$$
\begin{array}{rcl}
\underline{\gamma} & \in & \mathcal{P}(\mathsf{TypeVar} \to \mathsf{GType}) \\
\underline{\sigma} & \in & \mathcal{P}(\mathsf{TypeConstructor} \to \mathsf{GDatatype})
\end{array}
$$

Values at a ground type expression $\underline{v} \vDash^{\sigma} \underline{\tau}$

$$
\frac{}{\texttt{true}, \texttt{false} \vDash^{\sigma} \underline{\texttt{bool}}} \qquad \frac{}{\text{int}(n) \vDash^{\sigma} \underline{\texttt{int}}} \qquad \frac{}{\text{float}(x) \vDash^{\sigma} \underline{\texttt{float}}} \qquad \frac{}{\texttt{function} \vDash^{\sigma} \underline{\tau_1 \to \tau_2}}
$$

$$
\frac{\left(\underline{v_i} \vDash^{\sigma} \underline{\tau_i}\right)_{1 \leqslant i \leqslant n}}{(\underline{v_1}, \ldots, \underline{v_n}) \vDash^{\sigma} \underline{(\tau_1 \times \cdots \times \tau_n)}} \qquad \frac{\exists \underline{\tau} \in \mathsf{GType}, \quad \underline{v} \vDash^{\sigma} \underline{\kappa[\tau/\alpha]}}{\underline{v} \vDash^{\sigma} \underline{\exists \alpha.\kappa}} \qquad \frac{\forall \underline{\tau} \in \mathsf{GType}, \quad \underline{v} \vDash^{\sigma} \underline{\kappa[\tau/\alpha]}}{\underline{v} \vDash^{\sigma} \underline{\forall \alpha.\kappa}}
$$

$$
\frac{(t(\overline{\alpha}) := A) \in \sigma \qquad \underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{A[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]}}{\underline{v} \vDash^{\sigma} \underline{t(\tau_1, \ldots, \tau_n)}} \qquad \frac{(\underline{\tau_1} = \underline{\tau_2}) \implies \underline{v} \vDash^{\sigma} \underline{\kappa}}{\underline{v} \vDash^{\sigma} \underline{\kappa \restriction (\tau_1 = \tau_2)}}
$$

Values at a ground datatype $\underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{A}$

$$
\frac{\underline{v} \vDash^{\sigma} \underline{\tau_i} \qquad 1 \leqslant i \leqslant n}{C_i\ \underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{C_1 \text{ of } \tau_1 \mid \cdots \mid C_n \text{ of } \tau_1}} \qquad \frac{\underline{v} \vDash^{\sigma} \underline{\tau}}{\underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{C \text{ of } \tau\ [@@\texttt{unboxed}]}}
$$

$$
\frac{\left(\underline{v_i} \vDash^{\sigma} \underline{\tau_i}\right)_{1 \leqslant i \leqslant n}}{\{l_1 : \underline{v_1}; \ldots; l_n : \underline{v_n}\} \vDash^{\sigma}_{\mathsf{decl}} \underline{\{l_1 : \tau_1; \ldots; l_n : \tau_n\}}} \qquad \frac{\underline{v} \vDash^{\sigma} \underline{\tau}}{\underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{\{l : \tau\}\ [@@\texttt{unboxed}]}} \qquad \frac{\underline{v} \vDash^{\sigma} \underline{\tau}}{\underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{\tau}}
$$

Ground types at a mode $\underline{\tau} \vDash^{\sigma} m$, ground valuations at a context $\underline{\gamma} \vDash^{\sigma} \Gamma$

$$
\begin{array}{rcl}
\text{isfloat}(\underline{v}) & := & \exists x,\ \underline{v} = \text{float}(x) \\
\text{separable}(X) & := & (\forall \underline{v} \in X,\ \text{isfloat}(v)) \vee (\forall \underline{v} \in X,\ \neg\, \text{isfloat}(v))
\end{array}
$$

$$
\frac{}{\underline{\tau} \vDash^{\sigma} \texttt{Ind}} \qquad \frac{\text{separable}(\{\underline{v} \mid \underline{v} \vDash^{\sigma} \underline{\tau}\})}{\underline{\tau} \vDash^{\sigma} \texttt{Sep}} \qquad \frac{\forall \tau' \lhd \tau, \quad \underline{\tau'} \vDash^{\sigma} \texttt{Sep}}{\underline{\tau} \vDash^{\sigma} \texttt{Deepsep}} \qquad \frac{\forall (\alpha : m) \in \Gamma, \quad \underline{\gamma(\alpha)} \vDash^{\sigma} m}{\underline{\gamma} \vDash^{\sigma} \Gamma}
$$

Remark: we have $\underline{\exists \alpha.\, \alpha} \nvDash^{\sigma} \texttt{Sep}$, which means that not all ground types are separable.

Ground datatypes at a mode $\underline{A} \vDash^{\sigma}_{\mathsf{decl}} m$,
parametrized datatypes at a mode signature $A \vDash^{\sigma} t(\overline{\alpha : m})$,
datatype definitions at a block signature $\sigma \vDash^{\sigma} \Sigma$

$$
\frac{\text{separable}(\{\underline{v} \mid \underline{v} \vDash^{\sigma}_{\mathsf{decl}} \underline{A}\})}{\underline{A} \vDash^{\sigma}_{\mathsf{decl}} \texttt{Sep}} \qquad \frac{\forall \underline{\gamma} \vDash^{\sigma} \overline{\alpha : m}, \quad \underline{\gamma(A)} \vDash^{\sigma}_{\mathsf{decl}} \texttt{Sep}}{A \vDash^{\sigma} t(\overline{\alpha : m})} \qquad \frac{\forall (t(\overline{\alpha}) := A) \in \sigma, \quad A \vDash^{\sigma} \Sigma(t)}{\sigma \vDash \Sigma}
$$

Figure 5: Ground semantics

$$\frac{\forall \sigma \vDash \Sigma, \forall \gamma \vDash^{\sigma} \Gamma, \quad \gamma(\tau) \vDash^{\sigma} m}{\Sigma; \Gamma \vDash \tau : m} \qquad \frac{\forall \sigma \vDash \Sigma, \forall \gamma \vDash^{\sigma} \Gamma, \quad \gamma(A) \vDash^{\sigma}_{\mathsf{decl}}}{\Sigma; \Gamma \vDash_{\mathsf{decl}} A}$$

$$\frac{\forall \sigma_0 \vDash \Sigma_0, \quad \sigma_0, \sigma \vDash \Sigma_0, \Sigma}{\Sigma_0 \vDash \sigma \dashv \Sigma}$$

Figure 6: Judgment semantics

One does not have to trust the inference rules of our syntactic judgment $\Sigma; \Gamma \vdash \tau : m$, which may very well be wrong, to trust that this semantic judgment $\Sigma; \Gamma \vDash \tau : m$ captures a good notion of separability. One need to read the declarative rules of figures 5 and 6, which we designed to be obvious. It is now evident what the right statements should be for soundness and completeness of our syntactic inference system: it is sound if the syntactic judgment implies the semantic judgment (all judgments with a syntactic derivation are semantically true), and complete if the converse holds (all true judgments can be established by a syntactic derivation).

A.1 (Maybe-)Soundness and (in-)completeness

In a research paper, the perfect way to evaluate a syntactic system of inference rules is to provide an independent declarative semantics for it, state the corresponding soundness/completeness results, and prove them.

In real-world implementations of a programming language, the syntactic system is rarely written down (often, only the algorithm inspired from the rules is kept), the declarative semantics are only vague intuitions in the mind of the rule designers, and the statements are never formulated precisely enough to dream of a proof.

In this imperfect research paper, we have precise rules, and an independent declarative semantics, and precise statements... but most proofs are missing. In fact, it is fairly non-obvious that those proofs exist: as we were doing this precise formalization work we have discovered that completeness does not hold in presence of constraints, and even that the system is not principal – not only the inference rules, but even the semantics.

This subsection contains the best state of our knowledge and hopes about the formal status of our inference rules.

Conjectured Lemma 3 (Type expression soundness)**.** $\Sigma; \Gamma \vdash \tau : m \implies \Sigma; \Gamma \vDash \tau : m$

Remark: it is not obvious that the existential rule is sound, for example. It proves $\exists \alpha. \tau : m$ under a premise $\alpha : \texttt{Ind} \vdash \tau : m$. The values of $\exists \alpha. \tau$ are the union of all the $\underline{\tau[\sigma/\alpha]}$, and our induction hypothesis tells us that each $\underline{\tau[\sigma/\alpha]}$ indeed has the mode m. But separability is not at all stable by union: both `int` and `float` are separable, but their union is not. One needs to argue that either all $\underline{\tau[\sigma/\alpha]}$ are inhabited by floating-point numbers only, or that none of them contain floating-point numbers, and this is a subtle property of the structure of algebraic datatypes.

Conjectured Theorem 1 (Soundness)**.** $\Sigma_0 \vdash \sigma \dashv \Sigma \implies \Sigma_0 \vDash \sigma \dashv \Sigma$

Fact 1 (Incompleteness)**.** *There exists a τ such that $\varnothing; \varnothing \vDash \tau : m$ but $\varnothing; \varnothing \nvDash \tau : m$.*

Proof. Consider the two types $\tau_1 := \exists \beta. \beta \restriction (\texttt{int} = \texttt{bool})$ and $\tau_2 := \exists \alpha \beta. \beta \restriction (\alpha = \alpha \times \texttt{int})$. They are semantically separable, but our type system cannot prove it. The reason why they are semantically separable is that the equations never hold in our semantic model with finite ground types, so the constrained type is empty and thus trivially separable.

We could make our syntactic inference rules more complete by adding more equality-reasoning power to the judgment $\Sigma; \Gamma \vdash (\tau_1 = \tau_2)$. For the first case ($\texttt{int} = \texttt{bool}$), incompatible type constructors should not be considered equal. In the second case, ($\alpha = \alpha \times \texttt{int}$), one could add an occurs-check to rule out such recursive types, but note that this property, true in our model, may not hold in the real world – it does not in OCaml when `-rectypes` is used. □

This result suggests that equality-reasoning can be fairly subtle and that we should not expect syntactic completeness on types with constraints. We can still hope to have completeness in their absence.

Definition 2 (Constraint-free)**.** *A type τ, datatype A or definition block σ is* constraint-free *if it does not contain any type equality constraint ($\kappa \restriction (\alpha = \tau')$). We write $\mathsf{CF}(\tau)$, $\mathsf{CF}(A)$ or $\mathsf{CF}(\sigma)$.*

Conjectured Lemma 4 (Completeness on constraint-free type expressions)**.**

$$\mathsf{CF}(\tau) \wedge \Sigma; \Gamma \vDash \tau : m \implies \Sigma; \Gamma \vdash \tau : m$$

Conjectured Theorem 2 (Completeness on constraint-free signatures)**.**

$$\mathsf{CF}(\sigma) \wedge \Sigma_0 \vdash \sigma \dashv \Sigma \implies \Sigma_0 \vDash \sigma \dashv\!\vDash \Sigma$$

A.2 Constraint-free principality

Unfortunately, there is more bad news to come for equality constraints.

Fact 2 (Semantic non-principality)**.** *There exists a parametrized datatype A with $A \vDash^{\varnothing} t(\Gamma_1)$ and $A \vDash^{\varnothing} t(\Gamma_2)$, but $A \nvDash^{\varnothing} t(\min(\Gamma_1, \Gamma_2))$.*

Proof. Over two parameters α, β, take $A := \alpha \restriction (\alpha = \beta)$. Both $\Gamma_1 := \alpha : \texttt{Ind}, \beta : \texttt{Sep}$ and $\Gamma_2 := \alpha : \texttt{Sep}, \beta : \texttt{Ind}$ are admissible signatures for A. In particular, our inference rules can verify them: any Γ' with $\Gamma' \vdash \alpha = \beta$ has $(\alpha : \texttt{Sep}, \beta : \texttt{Sep})$. However, the minimum signature $\alpha : \texttt{Ind}, \beta : \texttt{Ind}$ is not valid for A. □

Note the example in the above proof can be represented as an OCaml datatype as follows.

```
type (_, _) strange_eq =
  | Strange_refl : 'a -> ('a, 'a) strange_eq [@@unboxed]
```

This results shows that some limitations of the system are not due to our typing rules, but a fundamental lack of expressiveness of the current modes and mode signatures as a specification/reasoning language. We would need a richer language of modes, keeping track of equalities between types, to hope to get a principal system.

Lemma 5. *If $\Sigma; \Gamma_1 \vdash \tau : m$ and $\Sigma; \Gamma_2 \vdash \tau : m$ and $\mathsf{CF}(\tau)$ then $\Sigma; \min(\Gamma_1, \Gamma_2) \vdash \tau : m$*

Proof. This proof is done by induction on the two derivations at once, but we need to use the Cut Elimination Lemma 1 first to be able to assume, by inversion/syntax-directedness, that the rules on both sides are the same. □

Corollary 1 (Constraint-free principality)**.**
If $\Sigma; \Gamma \vdash \tau : m$ with $\mathsf{CF}(\tau)$, then there exists a minimal context Γ_{min} such that $\Sigma; \Gamma_{\mathsf{min}} \vdash \tau : m$.
If $\Sigma_0 \vdash \sigma \dashv \Sigma$ with $\mathsf{CF}(\sigma)$, then there exists a minimal signature Σ_{min} such that $\Sigma_0 \vdash \sigma \dashv \Sigma_{\mathsf{min}}$.

Typer: ML Boosted with Type Theory and Scheme

Stefan Monnier

Université de Montréal - DIRO
monnier@iro.umontreal.ca

Abstract

We present the language Typer which is a programming language in the ML/Haskell family. Its name is an homage to Scheme(r) with which it shares the design of a minimal core language combined with powerful metaprogramming facilities, pushing as much functionality as possible into libraries. Contrary to Scheme, its syntax includes traditional infix notation, and its core language is very much statically typed. More specifically the core language is a variant of the implicit calculus of constructions (ICC). We present the main elements of the language, including its Lisp-style syntactic structure, its elaboration phase which combines macro-expansion and Hindley-Milner type inference, and its treatment of implicit arguments.

1 Introduction

Typer is an experimental programming language born from a desire to have a programming language as convenient as OCaml for everyday tasks, but with a type system as powerful as that of Coq and with a meta-programming system like that of Lisp. While Scheme with its dynamic typing may appear diametrically opposed to systems like Coq where the static typing can be extremely rigid, they both share the desire to make "everything" first class. In this sense, Coq shares some of Scheme's design, with a minimal but very powerful language at its core (Gallina) and layers of meta-programming added on top to make it convenient for the programmer. Yet, Coq's meta-programming does not follow the same minimalist approach: the meta-programs are split into tactics written in the Ltac or OCaml language and proof scripts, also written in Ltac, that call those different tactics. As a first approximation, those correspond respectively to Lisp macros and macro calls, except they don't reuse the same language and syntax as Gallina.

Typer starts with a core language similar to that of Coq, but combines it with a macro facility where macros are written directly in and called directly from Typer, so the language can be seamlessly extended via meta-programming as in Lisp. The syntactic extensibility is made possible by the use of a very primitive parsing technique, which is just flexible enough to support a fairly familiar infix syntax, yet simple enough that it maps straightforwardly to the equivalent of Lisp's S-expressions.

While the core language lets us manipulate proofs, Typer is mostly meant to be used as a programming language. So we wanted the power of fully dependent types to not unduly get in the way of programs that do not make use of them. Concretely, we tried to design Typer in such a way that programs can be written with just as few extra annotations as in any other ML/Haskell-family language.

The paper presents the following contributions of the design of Typer:

- A syntactic structure that shares many of the features of Lisp but uses operator precedence grammars to apply it to a more familiar infix syntax.

- An elaboration phase which combines HM-style type inference and macro-expansion, relying on the inferred type information to distinguish macro calls.
- An extension of ICC* [BB08] with inductive types.

2 Typer primer

Before getting to Typer's internals, we'll give a short overview of what the language looks like. To a first approximation Typer is very similar to other languages in the ML/Haskell family. It is a statically typed (pure) functional language, with basically two core elements: functions and datatypes. To define a function which adds 1 to its argument, you can write:

```
add1 : Int -> Int;
add1 = lambda x -> x + 1;
```

The definition above could have used a bit of a syntactic sugar to become:

```
add1 x = x + 1;
```

To define a new datatype to represent singly linked lists we can write:

```
type List (a : Type)
  | nil
  | cons (hd : a) (tl : List a);
```

where *hd* and *tl* are optional field names: we could have written just `cons a (List a)` instead. This *type* syntax is actually syntactic sugar provided by a predefined macro which rewrites the above to something of the form *List* = *typecons* ... where *typecons* is the low-level construct to define inductive types. Functions and data constructors are curried. You can define the *map* function as follows:

```
map : (a : Type) ≡> (b : Type) ≡>
      (a -> b) -> List a -> List b;
map f xs = case xs
  | nil => nil
  | cons x xs => cons (f x) (map f xs);
```

The type declaration is generally optional, although we currently require it for recursive definitions. The triple arrow `≡>` is used for functions whose argument is *implicit*, which is actually called *erasable* in Typer.

There is just one remaining important construct in Typer, which is *let*: This allows you to introduce new locally scoped definitions. The shape of this construct is "`let` *decls* `in` *exp*" where *decls* is a sequence of declarations such as the ones shown above, separated by semicolons. For example, we could have defined the above map function as follows:

```
map f =
  let map' xs = case xs
        | nil => nil
        | cons x xs => cons (f x) (map' xs)
  in map'
```

Typer has fundamentally two syntactic categories: expressions and declarations. A Typer file is defined as a sequence of declarations, separated by semicolons.

If these are all the constructs, you might wonder how macros are defined. They're defined simply as values with a dedicated type *Macro*, which can be constructed with the *macro* constructor. For example, the following code defines a new macro *declare_is_within_*:

```
diw : List Sexp -> ME Sexp;
diw args =
  let e1 = nth 0 args error_sexp;
      e2 = nth 1 args error_sexp;
      e3 = nth 2 args error_sexp
  in return (quote
             ((lambda (uquote e1) -> (uquote e3))
              (uquote e2)));

declare_is_within_ : Macro;
declare_is_within_ = macro diw;
```

quote is a macro similar to the backquote/quasiquote in Lisp macros (and *uquote* corresponds to the "unquoting" comma in those systems). Wherever the above macro definition is in scope, the following:

```
... declare_is_within_ x 2 (x + 4) ...
```

will be taken as a macro call which expands to:

```
... ((lambda x -> x + 4) 2) ...
```

The intention is likely to let the programmer call the macro with the syntax "declare e_1 is e_2 within e_3", but that requires a change to the grammar, which we show in the next section.

Being purely functional, Typer resorts to the usual monadic technique to get access to a side effecting world, just as is done in Haskell. In the above code, *ME* is the macro-expansion monad (like that of Template Haskell [SP02]) and *return* is the unit of that monad. This is used to allow side effects during macro expansion, such as emitting warnings, generating fresh variable names, and reading files.

Some aspects of Typer's base language are purposefully minimalist, leaving it to macros to provide a more convenient surface language. For example, the above definition of *declare_is_within_* can be defined using the define-macro macro:

```
define-macro (declare_is_within_ e1 e2 e3)
  return (quote
          ((lambda (uquote e1) -> (uquote e3))
           (uquote e2)));
```

As a sad note, these macros are currently not hygienic and the programmer needs to rely on *gensym* to avoid variable capture. This will hopefully be addressed soon.

```
type Sexp
  | node (head : Sexp) (args : List Sexp)
  | symbol (name : String)
  | number Int
  | string String
```

Figure 1: Definition of Typer's S-expressions

3 Syntactic structure

Typer's parser is divided into a *reader* which turns the source code into a generic tree structure called S-expression, and an *elaborator* which handles scoping, macro expansion, and type inference. The *reader* handles the lexical analysis, which requires all tokens to be separated by whitespace, except for a few single-char tokens, such as '`(`', '`;`', and a few more.

After lexical analysis is performed, the reader parses the stream of tokens according to a simple grammar to extracts the S-expression. The shape of S-expressions can be described with the datatype shown in Figure 1. Note that contrary to the Lisp S-expression syntax, parentheses are only used for grouping purposes, so `(x)` will produce the same *Sexp* as just `x`. And we use `()` as the printed representation of the zero-length *symbol* (which we call *espilon*).

Note how, at this stage, the representation of the code has no notion of scoping, functions, types, or function calls. It's only during elaboration that S-expressions will be analyzed to distinguish the various constructs such as macro calls, function calls, let bindings, variable references, etc.

Any S-expression written using an infix or mixfix operator can also be written some other way, following the underscore convention of Agda's mixfix [DN08]. In the case of Typer, instead of:

```
let x = a * b + c in x
```

the user can write

```
let_in_ (_=_ x (_+_ (_*_ a b) c)) x
```

and these two notations result in identical S-expressions. Note that contrary to "let", "=", and "in" which are keywords, *let_in_* is a normal identifier, with no special meaning in the *reader*'s grammar, so the following:

```
let_in_ g x = f x + a
```

is read in the same way as:

```
_=_ (let_in_ g x) (_+_ (f x) a)
```

3.1 Operator precedence grammar

Typer's external notion of S-expression is more flexible than Lisp's, since it allows infix notation. The *reader* relies on operator precedence grammars (OPG) [Flo63] for that. An OPG is a very restrictive subset of context free grammars, much more restrictive than LALR, for example.

You can think of the job of an OPG parser from the point of view of someone trying to add parentheses to render the document's structure explicit: whenever the parser sees something of

the form "kw_1 e kw_2" (where kw_1 and kw_2 are keywords and e is a possibly empty sequence of non-keyword tokens or fully parenthesized sub-trees), it just needs to decide whether that should be parenthesized as "kw_1 (e kw_2" or "kw_1 e) kw_2". For example, when starting with:

```
... g + f(5) * 6 - x ...
```

The parser will look at "`+ f(5) *`" and add an open parenthesis because it decides that the "`f(5)`" should be attached to the "`*`" keyword:

```
... g + (f(5) * 6 - x ...
```

then it will see "`* 6 -`" and add a close parenthesis this time:

```
... g + (f(5) * 6) - x ...
```

Then it will consider "`+ (f(5) * 6) -`" and add an open parenthesis:

```
... g + ((f(5) * 6) - x ...
```

and so on and so forth. What sets OPG apart here is that it makes these choices without considering e nor the surrounding context: it bases its decision only on the pair of keywords.

In Typer, the grammar is represented by a table which lists every token that should be considered as a keyword, along with its two precedence levels: one for its left side and another for its right side. Then parsing uses the following rule: when we see "kw_1 e kw_2", we lookup the right precedence of kw_1 and the left precedence of kw_2, and we then attach e to whichever is higher. If the precedences are equal, then we consider those keywords as part of a mixfix whose name it constructs by concatenating them as $kw_1\_kw_2$.

For example, given the default grammar, we can define the new syntax "declare e_1 is e_2 within e_3" by setting the precedences as follows:

```
define-operator "declare" () 2;
define-operator "is" 2 1;
define-operator "within" 1 66;
```

After that, such a form gets parsed identically to "*declare_is_within_* e_1 e_2 e_3". Note that the modification of the grammar is independent from the definition of *declare_is_within_* as a macro: the grammar changes only affect the parsing of source code into an S-expression. The resulting S-expression can then correspond to a macro call (as here), but it can just as well correspond to a function call (as is the case typically for infix mathematical operators like "+"), or it may even result in an invalid chunk of code (for example if the above operator definitions are used without any *declare_is_within_* in scope).

Clearly, hardcoding levels of precedence as in the example above is messy, but this is only the low-level interface provided by the *reader*. It can be supplemented with macros that provide a more flexible interface, for example letting the programmer specify precedence levels relative to existing ones, or using fragments of BNF grammar.

While it enjoys a simple and efficient implementationthe motivation behind the choice of an OPG parser was not efficiency but rather the following aspects:

- Extensible grammars suffer from an inherent lack of modularity, since the combination of two extensions can always lead to conflicts or ambiguities, sometimes in ways that can be difficult to understand. While OPG's simplicity means that conflicts are more frequent, they also tend to be much more superficial and hence easier to understand.

```
Ltype = Lexp
type Lexp
  | var    (name : Id) (pos : Int)
  | imm    (value : Limmediate)
  | prim   (id : String);
  | fix    (bindings : List Lbinding) (body : Lexp)
  | arw    (arg : Id) (atype : Ltype) (rtype : Ltype)
  | fun    (arg : Id) (atype : Ltype) (body : Lexp)
  | app    (f : Lexp) (arg : Lexp)
  | ind    (params : List Id) (cases : List LindCase)
  | con    (typ : Ltype) (name : Id)
  | case   (val : Lexp) (cases : List Lbranch)
```

Figure 2: Sketch of the definition of core λ-expressions

- More importantly, OPG grammars are "strongly context free": a given stream of tokens will be parsed in the same way regardless of the surrounding context.

The second point is what makes them particularly suitable for S-expressions, since at the time of parsing, we do not know yet what role a given S-expression will play: we do not yet know if it is a type, a pattern, a declaration, an expression, an lvalue, or anything else for that matter since it depends on the definition of the macros in which it appears.

For example, in OCaml the following piece of code:

```
let x = a; b in x = a; b
```

is parsed as

```
let (x = (a; b)) in ((x = a); b)
```

Notice that the '`;`' has higher precedence than '`=`' before the "*in*" keyword (i.e. in a declaration context) but it's the opposite after the "*in*" (i.e. in an expression context). So if Typer supported a similar grammar, when faced with a macro call of the form

```
mymacro (x = a; b)
```

the parser would need to know if the macro's argument should be parsed as a declaration or as an expression. We did not want to make the parser depend so tightly on the semantics of the language: by restricting ourselves to an OPG grammar we disallow these context-dependent parsing rules, such that we do not need to know what is a macro call, let alone figure out what is that macro's definition.

4 Elaboration

Elaboration is the phase in Typer's compiler which turns an S-expression into an expression in Typer's core λ-calculus. We want most of this phase to be itself implemented in Typer so that we can prove properties such as the correctness of the compilation of pattern matching [CA18, CB16].

Figure 2 shows a sketch of the definition of the datatype *Lexp* to represent core expressions and *Ltype* to represent their types. Notice that the there is really only one datatype used for both: the name *Ltype* is only used as a hint that this expression is expected to be a type, since *Lexp* is used both to represent what is usually considered *expressions* (such as *let*, *fun*, ...) as well as what is usually considered as *types* (e.g. *arw* x t_1 t_2 which represents the function type `(x : t1) -> t2`, or *ind* which is the representation of an inductive data type) since this core language is a kind of Pure Type System (PTS) [Bar91].

The different constructors of *Lexp* are as follows: *var* is a variable reference represented as a De Bruijn index together with the original name of the variable, used for error messages and debugging purposes; *imm* represents an immediate value such as a constant string or integer; *prim* represents built-in primitives such as the string concatenation and the string type; *fix* represents a set of local bindings which can be mutually recursive; *arw*, *fun*, and *app* represent respectively, the type of a function, a function, and a function call; finally *ind* represent an inductive data type, *con* represent a particular constructor of an inductive data type, and *case* represents the eliminator by case analysis.

Elaboration performs the following tasks:

- Finish the syntactic analysis: decide what is a function call, a macro call, a *let* definition, a *case* analysis, ...
- Macro expand the macro calls.
- Infer and verify the types.

This is the heart of Typer's front-end and requires a fair bit of supporting functionality: type inference needs to perform unification between arbitrary *Lexp* terms as well as compare them for equality, which involves normalizing them; macro expansion requires evaluation of arbitrary Typer code, either via a small interpreter, or via the complete compiler and runtime system.

4.1 Final syntactic analysis

Elaboration has to distinguish the different constructs of the language. Figure 3 shows a pseudocode of how it works. In the case of Typer, just as is the case of Scheme, we do it without hard coding the meaning of any identifier. More specifically, elaboration will not check to see if an *Sexp node* has a symbol named for example *let_in_* as its head. Instead, when it encounters a *node*, it does the following:

1. Elaborate the head, which will also return the inferred type of that expression.
2. If the returned type is *Special-Form*, it means this expression is one of the "built-in" ones and we want to call the corresponding special form's elaboration function, found in a global table. There is a special form for each core syntactic construct, such as `let_in_` and `case_`.
3. If the returned type is *Macro*, then it is a macro call, and we expand it, as detailed below.
4. Otherwise, it should be a function call and we recurse on each element of the node.

Note that at step 2 above, we have to double check that the elaborated head is of the form "*prim* ..", because the source code could be

```
elaborate : Ctx → Sexp → Pair Lexp Ltype;
elaborate c sexp =
 case sexp
  | symbol s ⇒ elab_variable_reference c s
  | number n ⇒ elab_immediate_value n
  | node head args ⇒
   let (e, t) = elaborate c head in
   case t
    | "Macro" ⇒ elaborate c (macroexpand e c args)
    | "Special-Form" ⇒ elab_special_form c e args
    | _ ⇒ elab_funcall c t e args;
```

Figure 3: Sketch of the elaboration

```
(if x then let_in_ else case_) 42
```

in which case the head is a valid expression of type *Special-Form* but we do not know which special form it is, so we have to reject such meaningless code.

The way keywords like *let_in_* get their special meaning is simply by binding them to the corresponding special form primitive in the initial environment. The programmers are free to rebind those identifiers if they want, or to bind the corresponding primitive to other identifiers.

4.2 Macro expansion

As explained above, a macro call is recognized simply by the fact that the head of the *node* has type *Macro*. As before with *Special-Form*, the mere fact that the head has type *Macro* does not guarantee that this is a valid macro. Again we may just be looking at a source code of the form:

```
(if x then mymacro else yourmacro) 42
```

where the head may be a valid expression of type *Macro* but is not really a macro we can expand because x will only be known at runtime. So, to make sure we do have a macro, we additionally need to verify that the head expression is *let-closed*. We say that a term is *let-closed* if all its free variables are bound by *fix* and that their definitions are themselves *let-closed*. Once established that it is *let-closed*, we can reduce it (by regular evaluation) to a value out of which we can finally extract the Typer function to call to perform the expansion. Since Typer is pure, the evaluation of the head has no visible side-effects and its result can be cached (contrary to the macroexpansion itself, which is done in the *ME* monad which is allowed to perform arbitrary side-effects).

This approach naturally supports lexically scoped macros or even higher-order macros. Its downside is that it needs to know the type of the head of a *node* to detect a macro call. This makes it virtually impossible to expand all macros in a separate phase before we infer types. And we can't infer types before we have expanded the macros either, so we are forced to interleave macro expansion and type inference within one big elaboration phase. While this can be a significant downside, performing macro expansion from inside the type inference phase

has the advantage that macros can get access to the complete type context and the expected return type.

In the context of macros that provide syntax extensions, this is currently of little benefit, but we intend to make it possible in the future to write other kinds of macros which act more like *proof tactics*, where the type environment represents the set of valid hypotheses, and the expected return type is the proposition one wants to prove. While Typer is a programming language at heart, its core language is very similar to that of proof assistants, so it can also be used to write and manipulate propositions and proofs.

4.3 Type checking

Bidirectional type checking [PT00], compared to the previously traditional way to check types, is a more principled method of checking types that propagates type information more effectively while keeping the type checking code simple, if not even simpler. Type checking of the various language constructs is split into two mutually recursive functions:

- *infer* takes a type environment and an *Sexp* and returns the corresponding elaborated *Lexp* along with its *Ltype*. If it is called on a construct it does not handle, it means type checking fails for lack of type annotation.
- *check* takes a type environment, an *Sexp*, and its expected type (an *Ltype*), and returns the elaborated form, of type *Lexp*. If it is called on a construct it does not handle, it delegates to *infer* and then checks equality with the inferred type (which is the only place where types are compared).

For example, typically function calls are handled in *infer* as follows:

1. When a function call is encountered, *infer* is called on the function part.
2. The returned type is verified to be that of a function and then split into the argument type and the return type.
3. Then *check* is called on the argument since we now know its expected type.
4. Finally we can construct the *app* node and return it along with its type.

The previously traditional way to check types basically only relied on *infer*, so the advantage of bidirectional type checking is in *check* which both lets us confine the place where type equality needs to be checked, and lets us propagate type information available from the context. E.g. traditionally functions are handled by *check* which makes it unnecessary to annotate the formal argument with its type when the context already gives this information.

In our case, this division of labor would result in too much code duplication since both *infer* and *check* would need to look for special forms and macro calls, so we want to have just a single *elaborate* function which plays both roles. To that end, we took the *elaborate* shown in Figure 3, whose type corresponds to what *infer* would need, and changed it to also cover the needs of *check*. Its resulting type is:

$$elaborate : Ctx \to Sexp \to Maybe\ Ltype \to Pair\ Lexp\ Ltype;$$

So *elaborate* can receive the expected type from the context when it is available. This information can be used not only to reduce the need for type annotations, but also to adapt the elaborated code to what the context expects, as we will see below.

4.4 Type inference

Bidirectional type checking is helpful to propagate existing type information, such as that given in top-level type annotations, but it is no substitute for real type inference. In order to provide the same kind of experience as in other members of the ML/Haskell family, Typer refines the type checking presented above with a form of Hindley-Milner (HM) unification-based type inference with let-polymorphism.

HM is designed for a much restricted language than Typer, so it requires some adjustments. For lack of a nice inference algorithm with principal types in ICC, Typer uses an ad-hoc algorithm which tries to be simple enough to be understandable for the user and works about as well as HM on those programs that could be written in ML.

HM is based on 3 elements: unification, automatic generalization, and automatic specialization. Typer's inference uses unification in the same way as HM: when *elaborate* is not provided with an expected type yet sees a construct for which the type cannot be inferred, rather than signaling an error it creates a new meta-variable which stands for the expected type; and in the reverse case where *elaborate* both receives an expected type and can infer the type of the expression, it tries to unify the two types, which will try to instantiate meta-variables such that the types become equal.

HM performs generalization (i.e. introduction of implicit (type) abstractions) whenever a value is defined in a let-binding. Typer does the same when the variable has no type annotation, but with the following caveat: if a free meta-variable is used in a non-erasable way, we signal an error since generalizing it with an erasable abstraction would lead to invalid code.

HM performs specialization (i.e. introduction of implicit (type) applications) every time an identifier is used, by instantiating all its type arguments with fresh meta-variables. In the context of Typer this is sometimes too eager and at the same time it is insufficient: it can be too eager because an identifier can be passed to a function which expects an erasable ("polymorphic") function argument, in which case the identifier should be passed as-is without specializing it. And it can be insufficient because a normal function can return an erasable function so specialization can be needed also at other places than where identifiers are used. So, instead Typer performs specialization as a kind of coercion: whenever an expression with erasable function type is used in a context which does not expect an erasable function, that expression is specialized (i.e. Typer inserts a type application with a fresh meta-variable). This is an example where we take advantage of the bidirectional type checking's ability to propagate the type expected from the context.

For definitions that come with a type annotation, Typer also provides a form of generalization: first, when elaborating a *type annotation*, all remaining free meta variables are generalized into erasable arrows, so the user can write:

```
map : (?a -> ?b) -> List ?a -> List ?b;
```

where we use the "`?`" prefix for user-written meta-variables, so Typer will turn `?a` and `?b` into two additional erasable arguments. This reproduces the same behavior as that used in systems such as Twelf [PS99].

Second, when a λ abstraction is elaborated in a context that expects an erasable function, we wrap it into an additional erasable λ. So if the previous type annotation is followed by:

```
map f x = ...;
```

the elaboration will automatically add the two additional (erasable) λ corresponding to `?a` and `?b`.

Lemma 4.1 (HM equivalence)**.** *For all expressions e in context Γ in the subset of Typer corresponding to HM, Typer's inference elaborate e to $\Gamma \vdash e : \tau$ iff HM elaborates it to $\Gamma \vdash e : \tau$.*

The proof is straightforward induction on e: the only significant difference is the conditions under which specialization happens, but in the language handled by HM all contexts expect monotypes and the only polymorphically typed elements are variables, so the conditions used by Typer degenerate to those of HM.

We do not know whether this inference algorithm is guaranteed to terminate in theory, but it seems to perform well in practice. Given that macro-expansion is allowed to perform arbitrary side-effects, we have already given up the idea of guaranteeing termination of elaboration anyway.

5 Core language

Typer's core language is based on ICC* [BB08, MLS08]. We start with a λ-calculus that is a sort of pure type system (PTS) [Bar91] extended with annotations to indicate which arguments are 'n'ormal and which are 'e'rasable:

$$\begin{array}{llll}
\textit{(level)} & \ell & \in & \mathbb{N} \\
\textit{(var)} & x, y, t & \in & \mathcal{V} \\
\textit{(sort)} & s & ::= & \mathsf{Type}\ \ell \\
\textit{(argkind)} & k & ::= & \mathsf{n} \mid \mathsf{e} \\
\textit{(exp)} & e, \tau & ::= & s \mid x \mid (x : \tau_1) \xrightarrow{k} \tau_2 \mid \lambda x : \tau \xrightarrow{k} e \mid e_1 @^k e_2
\end{array}$$

Those annotations are similar to those of Bernardy et al.'s colored PTS (CPTS) [BJP12], in that the annotation on a function or function call has to match the annotation of the function's type. The rules of the CPTS corresponding to Typer's core calculus are the following:

$$\begin{array}{lll}
\mathcal{S} & = & \{\ \mathsf{Type}\ \ell \mid \ell \in \mathbb{N}\ \} \\
\mathcal{A} & = & \{\ (\mathsf{Type}\ \ell : \mathsf{Type}\ (\ell + 1)) \mid \ell \in \mathbb{N}\ \} \\
\mathcal{R} & = & \{\ (\mathsf{e}, s, \mathsf{Type}\ \ell, \mathsf{Type}\ \ell);\ \mid s \in \mathcal{S}, \ell \in \mathbb{N}\ \} \\
 & & \cup\ \{\ (\mathsf{n}, \mathsf{Type}\ \ell_1, \mathsf{Type}\ \ell_2, \mathsf{Type}\ (\ell_1 \sqcup \ell_2)) \mid \ell_1, \ell_2 \in \mathbb{N}\ \}
\end{array}$$

Where $\mathcal{S}$ is the set of possible sorts (i.e. types of types), $\mathcal{A}$ is the set of axioms, and $\mathcal{R}$ specifies the set of allowed abstractions: (k, s_1, s_2, s_3) means that an arrow of color k can go from an argument in sort s_1 to a result in sort s_2, and that this arrow will live in sort s_3.

Compared to a normal CPTS, we use a slightly different typing rule for erasable functions:

$$\frac{\Gamma \vdash \tau_1 : s\Gamma, x : \tau_1 \vdash e : \tau_2 x \notin \mathsf{fv}(e*)}{\Gamma \vdash \lambda x : \tau_1 \xrightarrow{\mathsf{e}} e : (x : \tau_1) \xrightarrow{\mathsf{e}} \tau_2}$$

The difference compared to the rule for *normal* functions is the addition of the test that x doesn't appear in $e*$ which is the *erasure* of e. The erasure function $(\cdot)*$ erases type annotations as

well as all erasable arguments:

$$\begin{array}{lcl} s* & = & s \\ x* & = & x \\ ((x:\tau_1) \xrightarrow{k} \tau_2)* & = & (x:\tau_1*) \xrightarrow{k} \tau_2* \\ (\lambda x:\tau \xrightarrow{\mathsf{n}} e)* & = & \lambda x \to e* \\ (\lambda x:\tau \xrightarrow{\mathsf{e}} e)* & = & e* \\ (e_1 @^{\mathsf{n}} e_2)* & = & e_1* @ e_2* \\ (e_1 @^{\mathsf{e}} e_2)* & = & e_1* \end{array}$$

This expresses the fact that erasable arguments do not influence evaluation. So far, this is exactly like ICC*. But Typer extends this with inductive types.

5.1 Inductive types

Rather than decompose inductive types into separate unions and products as suggested by Bernardo in [Ber09], Typer keeps inductive types as a "hardcoded" combination of a sum of products:

$$\begin{array}{llcl} \textit{(label)} & l & \in & \mathcal{L} \\ \textit{(exp)} & e,t & ::= & \ldots \mid \mathsf{ind}\ l \mapsto \overrightarrow{\overrightarrow{x:^k \tau}} \mid \mathsf{con}(e,l) \mid \mathsf{case}\ e\ \mathsf{in}\ \overrightarrow{l\ y\ \vec{x^k} \Rightarrow e_l} \\ & & & \mid \mathsf{fix}\ \overrightarrow{x:\tau = e}\ \mathsf{in}\ e \end{array}$$

$$\begin{array}{lcll} (\mathsf{con}(e,l))* & = & \mathsf{con}(l) & \\ (\mathsf{ind}\ l \mapsto \overrightarrow{\overrightarrow{x:^k \tau}})* & = & \mathsf{ind}\ l \mapsto \overrightarrow{\overrightarrow{x:^k \tau*}} & \\ (\mathsf{fix}\ \overrightarrow{x:\tau = e}\ \mathsf{in}\ e)* & = & \mathsf{fix}\ \overrightarrow{x = e*}\ \mathsf{in}\ e* & \\ (\mathsf{case}\ e\ \mathsf{in}\ \overrightarrow{l\ y\ \vec{x^k} \Rightarrow e_l})* & = & \mathsf{case}\ e*\ \mathsf{in}\ \overrightarrow{l\ \vec{x}' \Rightarrow e_l*} & \text{where } \vec{x}' \text{ only includes the } x_i^{\mathsf{n}} \end{array}$$

This follows the approach used in Giménez [Gim94]. $\mathsf{con}(e,l)$ is a reference to the l constructor of inductive type e; arguments to the constructor are then passed via curried function calls. The differences with [Gim94] are: constructors are selected by labels instead of by position; fields can be annotated as erasable; inductive types cannot have *indices*; in ind each alternative is specified by the list of its field types rather than by the curried type of the constructor; ind does not take a variable x to be able to refer to itself, because we let fix play this role instead; case does not specify its own return type. For example, List can be defined as:

$$\begin{array}{ll} \mathit{List} & = \mathsf{fix}\ \mathit{List} = \lambda t : \mathit{Type}\ 0 \xrightarrow{\mathsf{n}} \mathsf{ind}\ \{\mathit{nil} \mapsto \cdot; \mathit{cons} \mapsto \{\mathit{car} :^{\mathsf{n}} t; \mathit{cdr} :^{\mathsf{n}} \mathit{List} @^{\mathsf{n}} t\}\}\ \mathsf{in}\ \mathit{List} \\ \mathit{nil} & = \lambda t : \mathit{Type}\ 0 \xrightarrow{\mathsf{e}} \mathsf{con}(\mathit{List} @^{\mathsf{n}} t, \mathit{nil}) \\ \mathit{cons} & = \lambda t : \mathit{Type}\ 0 \xrightarrow{\mathsf{e}} \mathsf{con}(\mathit{List} @^{\mathsf{n}} t, \mathit{cons}) \\ \mathit{singleton} & = \lambda t : \mathit{Type}\ 0 \xrightarrow{\mathsf{e}} \lambda x : t \xrightarrow{\mathsf{n}} \mathit{cons} @^{\mathsf{e}} t @^{\mathsf{n}} x @^{\mathsf{n}} (\mathit{nil} @^{\mathsf{e}} t) \end{array}$$

The typing rule for fix is out of the scope of this article, but it combines the usual syntactic checks for structural recursion when the definition is a function, and for strictly positive occurrences when the definition is that of an inductive type.

5.2 Equality type

An important pragmatic issue with inductive types and GADTs is how to provide type refinement in the branches of the case statement. Following the de Bruijn principle we want our core language's typing rule for case to be fairly primitive and leave it up to the elaboration phase to provide a more transparent form of refinement.

The careful reader has probably noted how we solved this problem: our inductive types simply cannot have *indices*, in other words our inductive types are like plain old algebraic data types rather than GADTs, so there is simply no refinement to be had in case branches.

This means that Typer's core language cannot directly express the obligatory length-indexed list type:

```
type NList (a : Type) : (n : Nat) -> Type
  | nil  : NList a zero
  | cons : a -> NList a n -> NList a (succ n)
```

To make up for it, Typer provides an additional built-in equality type "*Eq* e_1 e_2", with its customary *Eq_refl* constructor and *Eq_cast* eliminator, the eliminator encoding Leibniz equality. Armed with this equality type, Typer can now define the obligatory length-indexed list type as follows:

```
type NList (a : Type) (n : Nat)
  | nil (P ::: Eq n zero)
  | cons (n':::Nat) a (NList a n') (P ::: Eq (succ n') n)
```

where ::: is used to denote an erasable field. The two ways to define such a type are basically equivalent, and while the approach we chose was fairly common back when GADTs started to appear (e.g. in [SP04]), it is the exception rather than the norm nowadays. The factors that made us choose this approach are the following:

- It eliminates the subtle distinction between *parameters* and *indices* to inductive types: Notice how in the first definition of *NList* above, the two arguments a and n are presented differently because a is a parameter while n is an index. Here, it's pretty clear which argument will be a parameter and which an index, but this is not always the case. Furthermore the typing rules can be slightly different for the two cases in terms of universe constraints.
- As mentioned, it eliminates the need for case to provide a form of refinement for type indices, simplifying the typing rule of the case construct as well as its syntax: there is no need for the syntax of case to specify a return type for type checking to be decidable.
- Instead, the constructors directly carry the equality witnesses we need to implement the same refinement.

This last point means that when we perform a case on an object of type *NList* defined as above:

```
case (e : NList a n)
 | nil => <en>
 | cons x xs => <ec>
```

the branch *en* receives an (erasable and by default invisible) witness that "*Eq n zero*". We can then use it to adjust our return type according to the needed refinement. Compared to Coq's

match construct, this also eliminates the need to resort to the *convoy pattern* [Chl13] when this equality proof is needed for other reasons than to refine the return type of the branch. Of course, the downside is the need to manipulate those equality proofs, but we hope to hide (or auto-generate) these manipulations in the majority of cases by meta-programming.

In order for the return type of case to be able to depend on the value of the object analyzed, each branch additionally receives another equality witness. For instance the branch *en* above also receives an (erasable and invisible) witness that "*Eq e nil*".

Erasable arguments are usually not visible in the source code: they are "invisible" (or anonymous) variables. But they are very much present in the core language and the elaboration phase can make use of them just like normal variables. When needed, the source code can also make them visible, for example the pattern `nil (P := iszero)` would let *iszero* refer to the (still erasable) proof that "*Eq n zero*".

6 Related work

Like all languages, Typer takes inspiration from too many of its predecessors to be able to list them all. We will try and limit ourselves to some recent systems which share enough of their design or their goals here.

Honu [RF12] is a programming language in the Racket system which provides an extensible infix/prefix syntax integrated with Racket's metaprogramming facilities. Contrary to Typer, it delays some of the infix parsing to the macro-expansion phase, where grammar extensions are tied to macros. It can also handle more complex grammars than Typer, which can in turn lead to unexpected interactions between macros. The **Star** language [MS13] is a statically typed programming language which also makes it easy to define embedded DSLs via syntactic and macro expansion facilities, relying also on an OPG grammar, but contrary to Typer, it expands macros in a separate phase, which hence can't get access to the typing context.

Template Haskell [SP02] is an extension of Haskell to allow compile time metaprogramming. Typer's interleaving of type inference and macro expansion is very similar to that of Template Haskell. But Typer and Template Haskell differ in how the macros are used by the programmer: in Template Haskell, macro calls are made explicit in the source file by preceding them with a '`$`' sign rather than being determined by their type. Also Template Haskell is not meant to add new binding forms to the language: arguments to the macro are type checked before being passed to the macro.

Many languages like **Scala** [Bur13], **OCaml** [dR03], and **Coq** [HPM+00] offer extensible syntax and metaprogramming facilities in various forms, but these are bolted on pre-existing languages so they are less flexible and more complex than what we wanted in Typer.

Idris [Bra13] and **F-Star** [SHK+16] are programming languages with dependent types that share many of Typer's goals and also offers metaprogramming facilities, but these facilities are more aimed at writing proofs, so they do not contribute as much as they could to simplifying the design of the language.

Agda [BDN09] is a proof assistant with a syntax similar to Haskell's but with the possibility of adding mixfix and not just infix operators. Their use of mixfix operators like *if_then_else_* as

a way to add new syntactic forms is what gave us the idea of adding mixfix to S-expressions in Typer using operator precedence grammar [DN08].

Template-Coq [Mal14, ABC+18] reflects Coq's internal language so as to allow advanced meta-programming, mostly used to generate propositions and proofs. While its goals are similar to ours, it needs to preserve the existing Coq design, whereas the availability of these facilities permeates the design of Typer. Also Typer's meta-programming facilities currently only offer access to syntax and not to the reflected semantics like in Typed Template-Coq.

7 Conclusion

Typer is a new experimental language in the family of dependently typed functional programming languages. Its design is generally conservative in that it mostly uses existing solutions, but tries to streamline them and combine them is ways which hopefully simplify the overall system while making it more flexible at the same time. Its strength relies mainly in its syntactic flexibility and meta-programming facilities, although these are currently mainly focused on writing syntax extensions, and we plan to extend them so as to be able to manipulate also the *Lexp* representation of the code, more convenient when generating proofs. Its current main weakness is lack of hygiene, which needs to be addressed urgently.

While it has not been officially released yet, its code can be found at `https://gitlab.com/monnier/typer`.

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) grant N° 298311/2012. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

References

[ABC+18] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with Typed Template-Coq. In *Interactive Theorem Proving*, pages 20–39, 2018.

[Bar91] Henk P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):121–154, April 1991.

[BB08] Bruno Barras and Bruno Bernardo. Implicit calculus of constructions as a programming language with dependent types. In *Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *Lecture Notes in Computer Science*, Budapest, Hungary, April 2008.

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In *International Conference on Theorem Proving in Higher-Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78, August 2009.

[Ber09] Bruno Bernardo. Towards an implicit calculus of inductive constructions. extending the implicit calculus of constructions with union and subset types. In *International Conference*

on Theorem Proving in Higher-Order Logics, volume 5674 of *Lecture Notes in Computer Science*, August 2009.

[BJP12] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free: Parametricity for dependent types. *Journal of Functional Programming*, 22(2):1–46, 2012.

[Bra13] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[Bur13] Eugene Burmako. Scala macros: let our powers combine!: on how rich syntax and static types work with metaprogramming. In *Workshop on Scala*, pages 3:1–3:10, 2013.

[CA18] Jesper Cockx and Andreas Abel. Elaborating dependent (co)pattern matching. *Proceedings of the ACM on Programming Languages*, 2(ICFP):75:1–75:30, 2018.

[CB16] David Christiansen and Edwin Brady. Elaborator reflection: Extending Idris in Idris. In *International Conference on Functional Programming*, pages 284–297, September 2016.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013.

[DN08] Nils Anders Danielsson and Ulf Norell. Parsing mixfix operators. In *Implementation and Application of Functional Languages*, volume 6836 of *Lecture Notes in Computer Science*, pages 80–99, September 2008.

[dR03] Daniel de Rauglaudre. Camlp4 reference manual, 2003.

[Flo63] Robert W. Floyd. Syntactic analysis and operator precedence. *Journal of the ACM*, 10(3):316–333, July 1963.

[Gim94] Eduardo Giménez. Codifying guarded definitions with recursive schemes. Technical Report RR1995-07, École Normale Supérieure de Lyon, 1994.

[HPM+00] Gérard P. Huet, Christine Paulin-Mohring, et al. The Coq proof assistant reference manual. Part of the Coq system version 6.3.1, May 2000.

[Mal14] Gregory Malecha. *Extensible Proof Engineering in Intensional Type Theory*. PhD thesis, Harvard University, 2014.

[MLS08] Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In *Conference on Foundations of Software Science and Computation Structures*, volume 4962 of *Lecture Notes in Computer Science*, pages 350–364, Budapest, Hungary, April 2008.

[MS13] Frank McCabe and Michael Sperber. Feel different on the Java platform: The Star programming language. In *International Conference on Principles and Practices of Programming on the Java Platform*, pages 89–100, September 2013.

[PS99] Frank Pfenning and Carsten Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In *International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 202–206, July 1999.

[PT00] Benjamin C. Pierce and David M. Turner. Local type inference. *Transactions on Programming Languages and Systems*, 22(1):1–44, Jan 2000.

[RF12] Jon Rafkind and Matthew Flatt. Honu: A syntactically extensible language. In *Proceedings of Generative Programming and Component Engineering*, 2012.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Symposium on Principles of Programming Languages*, pages 256–270. ACM Press, January 2016.

[SP02] Tim Sheard and Simon Peyton Jones. Template metaprogramming for Haskell. In *Haskell Workshop*, pages 1–16, Pittsburgh, Pennsylvania, October 2002. ACM Press.

[SP04] Tim Sheard and Emir Pašalić. Meta-programming with built-in type equality. In *Logical Frameworks and Meta-Languages*, Cork, July 2004.

De l'Assembleur sur la Ligne ? Appelez TInA !

Frédéric Recoules[1], Sébastien Bardin[1], Richard Bonichon[1], Laurent Mounier[2], and Marie-Laure Potet[2]

[1] CEA LIST, Laboratoire de Sûreté et Sécurité du Logiciel, Paris-Saclay, France
prenom.nom@cea.fr
[2] Univ. Grenoble Alpes. VERIMAG, Grenoble, France
prenom.nom@univ-grenoble-alpes.fr

Résumé

Les méthodes formelles appliquées au logiciel ont fait des progrès importants lors des deux dernières décennies. Leur application à l'embarqué est ainsi un succès indéniable. Parmi les prochains défis réside la question de leur application à des domaines moins contraints. Par exemple, le développement en C de logiciels non-critiques utilise régulièrement l'insertion d'assembleur « en ligne », que ce soit pour optimiser certaines opérations ou pour accéder à des primitives systèmes autrement inaccessibles. Ceci empêche complètement l'utilisation des méthodes développés pour le C pur. Nous proposons ici TInA, une méthode générale, automatique, sûre et orientée vérification pour pouvoir porter le code assembleur en ligne vers un code C sémantiquement équivalent, et profiter en retour des analyses pré-existantes pour ce langage. Nos expérimentations sur du code C d'envergure montre la faisabilité et l'intérêt de l'approche.

1 Introduction

Contexte. Les méthodes formelles utilisées dans le développement de logiciel ont fait des progrès conséquents lors des deux dernières décennies [3, 12, 34, 22, 28, 41], avec des succès notables dans des domaines industriels critiques, souvent régulés, comme l'avionique, le transport ferroviaire ou l'énergie. Cependant, l'application de ces méthodes pour le logiciel disons « courant », moins contraint, que ce soit pour la sûreté ou la sécurité, reste un défi scientifique. En particulier, l'extension des méthodes issues de domaines critiques, avec des règles de codages strict et des processus de validation obligatoires, à des domaines généraux, plus variés et laxes dans les méthodes de développement, est une tâche difficile.

Problème. Nous considérons ici le problème de l'analyse de code « mixte », combinant assembleur en ligne et code C/C++. Cette fonctionnalité, présente dans les compilateurs GCC, clang et Visual Studio, permet d'intégrer des instructions assembleur au sein d'un programme C/C++. Elle est utilisée en général pour des raisons d'efficacité ou d'accès à des fonctionnalités bas niveau qui ne peuvent être déclenchées depuis le langage hôte. L'usage d'assembleur en ligne est plus fréquent qu'on n'imagine : ainsi **11%** des paquets Debian 8.11 écrits en C/C++ utilisent de l'assembleur en ligne, directement ou à travers des bibliothèques, avec des blocs allant jusqu'à 500 instructions, et **28%** des projets C les plus populaires sur Github en contiennent, d'après Rigger et coll. [44]. En vérité, l'assembleur en ligne est utilisé fréquemment dans des domaines comme la cryptographie, le multimédia ou les pilotes matériel.

Cependant, les analyseurs de programme C/C++ gèrent mal cette fonctionnalité. Certains comme Frama-C [34] la gèrent peu, d'autres comme KLEE [12] ne le gèrent pas du tout. Dans ce cas, l'analyse produira des résultats incorrects ou incomplets. Ceci correspond à un réel problème d'applicabilité des techniques avancées d'analyse de code.

Étant donné qu'il est très coûteux de redévelopper des analyseurs dédiés, la façon usuelle de traiter ces morceaux de code assembleur est de proposer un code équivalent, dans le langage hôte (C/C++) ou bien sous forme de spécification logique — deux méthodes que Frama-C [34] permet. Dans les deux cas, cette tâche est effectuée manuellement : cela met d'emblée hors de portée l'analyse de larges bases de code. Il est en effet chronophage d'annoter ou de traduire manuellement de gros bouts d'assembleur, et cela est aussi une source d'erreur non négligeable. Plus le morceau d'assembleur est gros, plus ces problèmes peuvent devenir importants.

Objectifs. Nous souhaitons concevoir et développer une technique automatique, générique, sûre et orientée vérification pour porter le code assembleur en ligne en code C sémantiquement équivalent, afin de réutiliser les analyses de vérification C/C++. Cette méthode se veut :

Largement applicable Elle ne doit pas être liée à une architecture matérielle, un dialecte d'assembleur ou un compilateur particulier, tout en gérant un large sous-ensemble de l'assembleur en ligne qui se trouve dans les applications courantes ;

Correcte Le processus de traduction doit maintenir exactement tous les comportements du code mixte d'origine, et proposer une façon de démontrer ladite correction ;

Compatible avec la vérification formelle La traduction se doit d'être agnostique vis-à-vis des techniques de vérifications classiques (e.g., exécution symbolique [33], vérification déductive [27, 29] ou interprétation abstraite [20]), tout en permettant une qualité d'analyse post-traduction suffisamment bonne en pratique (nous utiliserons pour décrire cette propriété le terme de *vérifiabilité*).

Les travaux menés jusqu'à présent en vérification de code « mixte » ne remplissent pas tous ces objectifs. Ainsi, Vx86 [37] est ciblé `x86` (architecture Intel 32-bits) et vérification déductive mais ne fournit aucun moyen de montrer la correction de la traduction. Au premier abord, les techniques de décompilation [15, 16, 17] peuvent sembler bien adaptées. Mais leur but premier est l'aide à la rétro-ingénierie. De fait, cette famille de techniques peut remettre en cause la correction du processus, à tel point que « les décompilateurs existants produisent fréquemment un code décompilé qui n'est pas complètement fonctionnellement équivalent au programme d'origine » [48]. Certains travaux récents [10, 48] ciblent ce critère de correction mais Schwartz et. al [10] n'étudient pas la question de savoir si le code produit peut être vérifié par les outils actuels et ne démontrent pas la correction de leur approche — bien qu'à leur crédit, ils montrent un certain degré de correction via du test intensif. Schulte et coll. [48] démontre sans équivoque la correction de leur méthode mais l'approche *search-based* qu'ils proposent ne termine pas toujours et ne parle pas de vérifiabilité.

Proposition. Nous proposons TInA (*Taming Inline Assembly*), une technique automatique, générique, sûre, orientée vérification pour porter du code assembleur en ligne vers un C équivalent.

Un élément clé de TInA est qu'en nous concentrant sur l'assembleur en ligne, plutôt que sur le problème général de la décompilation, nous attaquons un problème plus restreint (taille de code, portée bien définie, type d'instructions et de constructions), mieux défini (grâce à l'interface avec le code C), ouvrant ainsi la porte à une technique ciblée plus puissante.

En particulier, TInA est fondé sur les principes suivants :

— La réutilisation de plateformes d'analyse de code exécutable [9, 23, 31, 51] qui permettent de porter du code binaire vers un langage intermédiaire, indépendant de l'architecture matérielle, éprouvé et concis.

— La validation de la traduction, pour la sûreté de notre méthode, plutôt que la validation du traducteur. Ce problème est en général plus aisé et réduit la base de confiance à un vérificateur testé intensivement, et qui pourrait, lui, être validé.

— Un algorithme de vérification d'équivalence de programmes dédié [1], ramené ici à la résolution d'un problème plus réduit, ciblé pour notre processus.

— Des passes de transformations dédiées, pour améliorer la vérifiabilité de notre résultat, par raffinements successifs du langage intermédiaire brut vers des abstractions le rapprochant de C (flot de contrôle haut niveau, types de données, opérations).

Contributions. En résumé, cet article décrit les contributions suivantes :

— Une chaîne outillée permettant la vérification de programme mélangeant de l'assembleur en ligne et du C (Sec. 3.1) ;

— Une méthode (Sec. 3.2) pour ramener l'assembleur en ligne à du code C, améliorée par l'usage du contexte dans lequel l'assembleur se trouve inclus, et validée automatiquement afin de pouvoir faire confiance au processus de traduction mais également de ne pas mettre en péril les critères de correction des analyses formelles considérées ;

— Le prototypage de notre méthode atteste de son intérêt pratique (Sec. 4) : nous portons et validons 100% des blocs assembleur de 3 projets libres importants, et nous confirmons aussi la vérifiabilité du code produit.

2 Contexte et motivation

Principe. Concentrons-nous sur un extrait de code assembleur en ligne (x86) décrit en Fig. 1a, provenant des sources du logiciel UDPCast. Le compilateur va traiter ce morceau quasiment en aveugle. En réalité, il n'est concerné que par la spécification des entrées, sorties et éléments modifiés (en anglais, *clobbers*), donnée par le programmeur et contenant des contraintes qui pourront par exemple être utilisées lors de l'allocation de registres. Ce mode d'utilisation avec spécification, représentatif de l'assembleur en ligne dit « étendu », est celui que le manuel de GCC conseille. Mais le code assembleur en lui-même, notamment toutes ses mnémoniques, est opaque et est propagé *tel quel* — en tant que chaîne de caractères — jusqu'à l'émission de code.

Exemple. Le code en Fig. 1a est entouré d'une spécification, à partir d'un langage concis de description de contraintes, dans des zones séparées par ':'.

— Tout d'abord des contraintes d'allocations de variables pour les sorties :

0. `"=c" (__d0)` indique que la variable `__d0` doit être affectée au registre `ecx` ;
1. `"=D" (__d1)` indique que la variable `__d1` doit être affectée au registre `edi` ;

En fait ces contraintes sont présentes ici pour pouvoir y référer dans la partie suivante tout en explicitant au compilateur que la valeur contenue par le registre (par exemple `ecx`) à la fin du bloc peut être différente de celle passée en entrée.

— Nous avons ensuite la description des entrées : `"a"` (`eax`) contient la valeur `0`, le registre décrit en 0 (i.e. `ecx`) contient `sizeof(fd_set) / sizeof(__fd_mask)` et celui en 1 (`edi`) contient `(&((read_set)->__fds_bits)[0])`.

1. Les problèmes de vérification de logiciel sont généralement indécidables — c'est le cas de l'équivalence de programmes. Cela n'empêche bien évidemment pas les outils de vérification d'exister et d'être utiles en pratique.

```c
# 54 "/usr/include/i386-linux-gnu/sys/select.h"
typedef long int __fd_mask;

# 64 "/usr/include/i386-linux-gnu/sys/select.h"
typedef struct {
  __fd_mask __fds_bits[1024 / (8 * sizeof(__fd_mask))];
} fd_set;

# 1074 "socklib.c"
int udpc_prepareForSelect
(int *socks, int nr, fd_set *read_set)
{
  /* [...] */
  int maxFd;
  do {
    int __d0, __d1;
    __asm__ __volatile__
      ("cld; rep; " "stosl"
       : "=c" (__d0), "=D" (__d1)
       : "a" (0),
         "0" (sizeof(fd_set) / sizeof(__fd_mask)),
         "1" (&((read_set)->__fds_bits)[0])
       : "memory");
  } while (0);
  /* [...] */
  return maxFd;
}
```

(a) Version originale (socklib.i)

```c
# 1074 "socklib.c"
int udpc_prepareForSelect
(int *socks, int nr, fd_set *read_set)
{
  /* [...] */
  int maxFd;
  {
    int __d0;
    int __d1;
    __fd_mask *__tina_4;
    unsigned int __tina_3;
    __tina_3 = sizeof(fd_set) / sizeof(__fd_mask);
    __tina_4 = & read_set->__fds_bits[0];
    {
      long *__tina_edi;
      unsigned int __tina_ecx;
      __TINA_BEGIN_1__: ;
      __tina_ecx = __tina_3;
      __tina_edi = __tina_4;
      while (0U != __tina_ecx) { // rep
        *__tina_edi = 0;         // stosl
        __tina_edi ++;           // rep
        __tina_ecx -;           // rep
      }
      __TINA_END_1__: ;
    }
  }
  /* [...] */
  return maxFd;
}
```

(b) Version C générée par TINA

FIGURE 1 – Exemple initial

— Enfin, il est spécifié que toute la mémoire peut être changée ("memory"). Cela indique au compilateur de sauver ce qu'il juge précieux de sa mémoire avant d'entrer dans ce bloc.

Cette spécification s'applique au code "cld ; rep ;" "stosl" dont la sémantique est qu'il met à zéro l'indicateur (*flag*) de direction df, puis remplit ecx double-mots à partir du pointeur edi avec la valeur contenue dans eax. Comme précisé dans le manuel de l'architecture [30], df dirige le signe de l'incrément : quand df est à zéro, l'incrément est positif. Le code produit par TINA est donné en Fig. 1b : il comporte bien une boucle, que la sémantique informelle annonçait, mais des éléments (cld, "a" (0)) ont été nettoyés du résultat.

Comportement des analyses. Si nous essayons d'analyser du code comportant ce genre de morceau d'assembleur en ligne, les outils peuvent avoir des comportements erratiques. Certains s'arrêtent tout bonnement avec une erreur — c'est après tout un comportement correct — d'autres, comme Frama-C, émettent un messages d'alarme. Ici Frama-C émet "Clobber list contain "memory" argument. Assuming no side-effect beyond those mentioned in output operands", message clair mais dont la correction est discutable puisque le mot-clé "memory" explicite justement que nous écrivons potentiellement dans toute la mémoire et Frama-C l'ignore en ne prenant en compte que les sorties — le seul choix correct, sans analyser l'assembleur, est de s'arrêter (ou de mettre toute la mémoire à $\top$ si nous étions en inteprétation abstraite). Une seule ligne d'assembleur a ainsi le pouvoir de mettre à mal les outils d'analyse. Bien entendu, on peut contourner le problème, par exemple en réécrivant manuellement le morceau en code C sémantiquement équivalent. Cependant, il est clair que la méthode manuelle aura du mal à passer à l'échelle et sera aussi source d'erreurs.

Propriétés de l'assembleur en ligne. Nous pouvons améliorer cet état de fait en exploitant les propriétés suivantes, spécifiques à l'assembleur en ligne :

P1. La taille des morceaux d'assembleur en ligne est généralement réduite : un tel morceau comporte en moyenne moins de 10 instructions, atteignant rarement plusieurs centaines

d'instructions ;

P2. La structure du flot de contrôle est relativement simple, avec seulement quelques conditionnelles et boucles, et aucun saut calculé dynamiquement ;

P3. L'interface des morceaux d'assembleur avec le code C est généralement spécifiée ;

P4. Enfin, le morceau d'assembleur est intégré à un contexte (ici en C), qui contient en particulier des informations de typage : c'est une information particulièrement recherchée en décompilation, souvent de manière heuristique, que nous n'avons dans notre cas plus qu'à propager.

En tirant parti de ces éléments, nous allons définir en Sec. 3 une méthode automatique (pour passer à l'échelle), sûre (par validation de la correction de la technique) et orientée vérification (pour pouvoir continuer à utiliser les outils avancés de vérification C déjà produits) pour obtenir le code C depuis l'assembleur en ligne. De surcroît, cette méthode est générique : elle fonctionne de la même façon quelle que soit l'architecture cible du code assembleur.

3 TInA : porter l'assembleur en ligne vers C

Nous allons décrire TInA tout d'abord dans ses grandes lignes (Sec. 3.1) avant de détailler les phases techniques internes du portage (Sec. 3.2).

3.1 Survol de la méthode

La méthode que nous proposons porte l'assembleur en ligne vers un code C sémantiquement équivalent, en utilisant les propriétés P1–P4 décrites en Sec. 2. Nous procédons en deux temps : tout d'abord, le code est *porté* de l'assembleur vers le C puis cette traduction est *validée*. Nous détaillerons quelques aspects importants de ces phases en Sec. 3.2. Mais concentrons-nous tout d'abord sur l'approche générale, schématisée en Fig. 2.

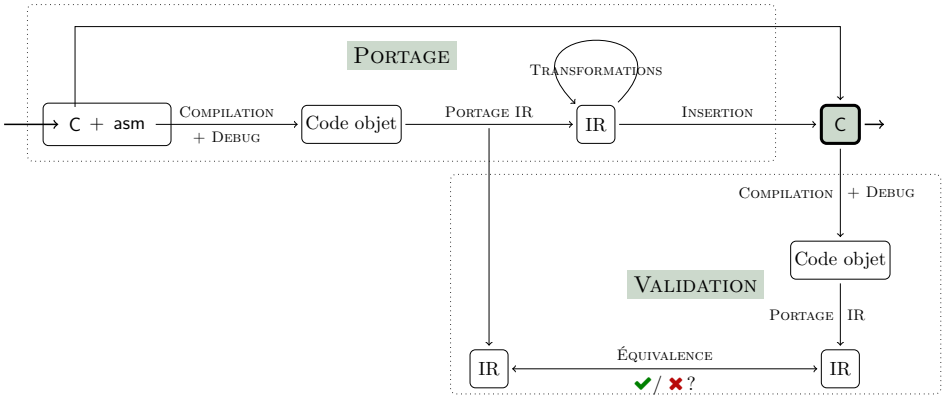


Figure 2 – TInA : une vue haut niveau

Notre méthode exploite les fonctionnalités d'outils existants comme Frama-C [34], pour son *front-end* C, et Binsec [23] pour traduire le code exécutable en représentation intermédiaire DBA [4], faire nos analyses et utiliser les solveurs SMT [6] pour la validation.

Localisation. Le processus commence par l'identification d'un code source C qui contient un ou plusieurs morceaux d'assembleur. Ce code initial nous permet de localiser les morceaux d'assembleur via Frama-C — c'est ce qui nous permettra finalement de substituer un code C équivalent au bon endroit.

Nous extrayons également la spécification décrite par le programmeur du morceau d'assembleur : entrées, sorties, zones modifiées. Nous l'utiliserons pour vérifier que le code écrit la respecte. Cette vérification est une passe que les compilateurs ne font pas : en effet, l'assembleur en ligne est propagé comme chaîne de caractères, dans laquelle les motifs identifiés seront remplacés lors de la phase d'allocation de registres (par des registres, des variables de piles ou de tas). Ainsi, *les mnémoniques assembleurs sont gardées intactes par le compilateur*. Par ailleurs, la spécification contient l'information nécessaire pour lier les noms de variable C aux noms de registres ou de zones mémoire dans notre représentation intermédiaire étendue.

Compilation. À la fin de cette phase d'extraction, le code source est compilé pour l'architecture cible *avec les informations de débogage*. Une fois encore, nous allons exploiter la présence du code source pour nous faciliter la tâche. En effet, nous avons la possibilité de contrôler la compilation du code. Par conséquent, nous choisirons bien entendu d'inclure toutes les données qui peuvent nous faciliter la tâche de reconstruction de code C. Par exemple, nous souhaitons avoir accès aux noms et types de variables du contexte. Ici , nous exploitons le format de débogage DWARF [25] pour faire passer cette information jusqu'au binaire duquel nous porterons le code vers du C.

$$
\begin{array}{rcl}
\mathsf{inst} & := & \mathsf{lv} \leftarrow \mathsf{e} \mid \mathsf{goto\ e} \mid \mathsf{if\ e\ then\ goto\ e} \\
\mathsf{lv} & := & \mathsf{var} \mid [\mathsf{e}]_{\mathsf{n}} \\
\mathsf{e} & := & \mathsf{cst} \mid \mathsf{var} \mid [\mathsf{e}]_{\mathsf{n}} \mid \mathsf{unop\ e} \mid \mathsf{binop\ e} \\
\mathsf{unop} & := & \neg \mid - \mid \mathsf{uext}_{\mathsf{n}} \mid \mathsf{sext}_{\mathsf{n}} \mid \mathsf{restrict}_{\mathsf{i}\ldots\mathsf{j}} \\
\mathsf{binop} & := & + \mid - \mid \times \mid /_{\mathsf{u,s}} \mid \%_{\mathsf{u,s}} \mid \wedge \mid \vee \mid \oplus \\
 & \mid & << \mid >>_{\mathsf{u,s}} \mid >_{\mathsf{u,s}} \mid <_{\mathsf{u,s}} \mid = \mid \mathsf{concat}
\end{array}
$$

FIGURE 3 – DBA : un langage intermédiaire bas niveau à base de bit-vecteurs

Portage vers IR. À ce stade, nous commençons le processus de traduction lui-même. D'abord, nous utilisons BINSEC pour revenir du code binaire vers une représentation intermédiaire DBA [4]. L'IR DBA (Fig. 3) est un langage minimaliste bas niveau à la sémantique est bien définie, utilisé dans BINSEC pour modéliser la sémantique des mnémoniques. Il est indépendant de l'architecture (BINSEC supporte x86 et ARM-v7) et ses opérations sont toutes réalisées sur des bit-vecteurs. L'IR DBA, étendu par l'information de débogage de la compilation, servira de support aux différentes passes d'analyses et de raffinements. La mémoire est gérée comme un tableau accédé par l'opérateur [ ].

L'utilisation directe du code binaire peut sembler une complication arbitraire à première vue. C'est en réalité l'unique endroit où le morceau d'assembleur se trouve totalement instancié et plongé dans son contexte — les noms de registre sont explicités, les positions dans la mémoire ont été résolues par le compilateur. Si nous nous étions positionnés dans une phase amont, nous aurions eu à résoudre au moins un problème similaire à l'allocation de registre — une fois par architecture supportée.

Portage vers C. À cet instant, nous sommes prêts à entamer le portage proprement dit à travers une succession de transformations détaillées en Sec. 3.2, dont l'architecture interne ressemble à celle d'un compilateur. Le résultat est un code C pur débarrassé de l'assembleur en ligne remplacé par un code C dont allons maintenant démontrer l'équivalence sémantique.

Recompilation et validation. La phase de validation démarre par une nouvelle compilation, *sans optimisation*, afin de préserver la structure du code, cette fois en partant du code C émis par le portage initial. D'une façon similaire, nous arrivons à localiser via les informations DWARF le code binaire correspondant à notre code C. Nous sommes à ce stade en possession de deux morceaux de code DBA, l'un venant de cette dernière localisation et l'autre venant de la première compilation. La phase de validation consiste ainsi à *démontrer l'équivalence sémantique des deux codes DBA*.

Pour ce faire, nous utilisons une méthode originale fondée sur un isomorphisme de graphe, facilité par la préservation de la structure du code, et une preuve d'équivalence bloc à bloc, démontrée par solveur SMT. Ceci permet de se débarrasser des boucles dans la traduction logique de la preuve d'équivalence. Cette méthode particulière est utilisable car nous avons la main sur toute la chaîne de compilation et de transformation de portage : ainsi nous faisons en sorte que la structure par blocs de base reste inchangée afin de pouvoir appliquer notre stratégie.

La phase de validation a posteriori de notre traduction permet ainsi à notre technique d'être sûre et par conséquent de s'insérer dans un contexte de vérification sans remettre en cause les propriétés de correction des outils d'analyse. Un autre aspect de cette orientation vers la vérification, concernant une mesure de l'impact pratique sur les outils d'analyse, sera étudié en Sec. 4.2. Revenons tout d'abord sur les détails des transformations qui vont en définitive permettre l'efficacité pratique de la méthode.

3.2 Détails du portage de l'IR vers C

Les détails du portage comportent 9 passes de transformation qui permettent de revenir d'un code assembleur à un code C sémantiquement équivalent. À la suite de ces passes de transformation, le code C est émis.

1. *(Renommage)* La première passe de renommage utilise les informations liées à l'interface, transmise par le DWARF. Ainsi les variables DBA liées aux registres ou zones mémoires sont remplacées par les noms idoines du contexte C. Le DWARF nous permet d'identifier exactement les variables, par exemple pour déterminer si une variable locale est affectée en registre, événement plutôt rare en x86 compilé sans optimisation, ou dans la pile, comme décalage par rapport au registre (ebp).

2. *(Contrôle)* Avant d'entamer des phases de transformations plus intensives, nous contrôlons que les spécifications sont correctes, par exemple qu'il n'y pas d'entrée non déclarée, pas de sortie non déclarée, ou encore que le code est bien invariant modulo allocation de registres pour ne pas être à la merci du compilateur.

3. *(Déballage)* La seconde étape déballe les sous-éléments accessibles des registres pour pouvoir les substituer directement dans le code DBA. En effet, une affectation d'un registre comme eax se verra décomposée en de multiples affectations restreintes dont celles de ax (les 16 bits de poids faibles d'eax) et al (resp. ah), c'est-à-dire les 8 bits de poids faibles (resp. forts) d'ax. Par la suite ces éléments seront directement substitués en lieu et place des lectures réduites aux mêmes sous-ensembles du registre eax. Cette phase est particulièrement adaptée aux registres xmm, de taille 128 à 512 bits, représentants généralement des vecteurs d'éléments indépendants et plus petits. De manière générale, cette phase permet de gagner en précision sur les registres utilisés pour stocker plusieurs entités indépendantes.

La substitution des sous-éléments, à laquelle succède une phase de filtrage des variables non utilisées permet de propager une information plus précise sur les opérations menées sur les sous-ensembles de registres.

4. *(Conditions de haut niveau)* Les conditionnelles en assembleurs sont des opérations faites sur des indicateurs (*flags*), par exemple de signe, de débordement ou de retenu et non sur une comparaison de haut niveau retournant un booléen. Cette phase réutilise ainsi la technique de Djoudi et coll. [24], fondée sur l´équivalence sémantique démontrée par solveurs SMT, pour recomposer des comparaisons booléennes de haut niveau, plus « lisibles »— qui favorisent également les analyseurs. Comme pour la passe 3, les instructions haut niveau retrouvées sont insérées dans le code DBA, puis une passe de filtrage de variables non utilisées nettoie le code pour ne laisser que les élément nécessaires au calcul de haut niveau.

5. *(Pré-structuration du code)* Nous souhaitons retrouver un code C aussi bien structuré que possible dans le but d'aider les analyseurs (et d'améliorer la lisibilité du code produit). Cette passe recherche ainsi des motifs correspondants aux structures de contrôle C comme les conditionnelles ou les boucles. Pour ce faire, le programme est décomposé en blocs de base, sur lesquels nous allons reconnaître des motifs de haut niveau — comme une conditionnelle avec deux branches et un postlude commun. Ce motif reconnu a lui-même une forme de bloc, sur lequel nous appliquons la même méthode, et ainsi de suite, récursivement. Cette méthode est ainsi appliquée de la base (blocs de base) au sommet pour structurer notre code.

6. *(Mise en forme SSA)* Avant d'appliquer d'autres transformations, le code est mis en forme SSA. Cela permet, comme d'habitude, de faciliter l'implémentation des phases postérieures.

7. *(Propagation de constantes et d'expressions & élimination des sous-expressions communes)* Pour aller plus loin dans la simplification du code, nous procédons ensuite à une phase classique de propagation de constante et d'expressions. Cette deuxième propagation est contrebalancée par une forme d'élimination de sous-expressions communes globale au morceau d'assembleur en ligne. Ici encore, les éléments nouveaux sont insérés puis le code est filtré.

8. *(Transformation d'opérations bit-à-bit en arithmétique)* Pour faciliter d'une part la lisibilité du code et d'autre part la gestion de celui-ci par certaines analyses, nous transformons autant que faire se peut les opérations bit-à-bit vers des opérations arithmétiques sémantiquement équivalentes. Le code assembleur en ligne comporte fréquemment ce type d'optimisation manuelle, qui peut parfois affecter les analyseurs de code.

9. *(Normalisation de boucle)* Enfin, notre dernière passe normalise la structure des boucles, en particulier les formes de compteur, afin d'en faciliter la gestion par les analyses ultérieures.

4 Expérimentations

Nous évaluons TInA (Sec. 3.1) via deux questions : 1) Est-elle applicable sur le type de code assembleur généralement rencontré dans les logiciels actuels ? 2) Comment se comportent les analyseurs de code lorsque nous portons le code assembleur vers du C ? A-t-on un effet bénéfique sur la qualité d'analyse via l'utilisation de notre méthode ?

Pour aborder ces questions, nous utiliserons trois projets libres qui contiennent un grand nombre de code assembleur en ligne : ALSA [2], pourvoyeur des fonctionnalités son et MIDI au système Linux, ffmpeg [3] le couteau suisse multi-plateforme pour la conversion de son et de vidéo et GMP [4] la bibliothèque GNU de manipulation d'arithmétique en précision arbitraire.

2. https://www.alsa-project.org/main/index.php/Main_Page
3. https://www.ffmpeg.org/
4. https://gmplib.org/

TABLE 1 – Applicabilité sur les morceaux asm de 3 projets open-source : ALSA, ffmpeg, GMP

	ALSA	ffmpeg	GMP	TOTAL	
Blocs asm	25	103	237	365	
Triviaux	0	6	13	19	
Hors-sujet	0	19	0	19	
Rejetés	0	55	1	56	
Pertinents	25	25	223	273	
Portés	25	25	223	273	100%
Validés	25	25	223	273	100%
Taille moyenne (instructions)	50	50	7	15	
Taille maximum (instructions)	70	341	31	341	

4.1 Généralité de l'applicabilité de la méthode

Pour évaluer l'applicabilité de notre méthode, nous l'employons sur tous les morceaux d'assembleur en ligne trouvé dans les sources des 3 projets sus-cités, ciblant l'architecture x86. Au total, ces projets contiennent **365** morceaux d'assembleur. Les résultats sont résumés en Table 1.

Nous écartons de l'évaluation les morceaux triviaux (vides ou non utilisés) ou hors-sujet, et rejetons ceux qui violent certaines hypothèses de sûreté (accès non-déclarés par exemple). Un morceau d'assembleur est ici considéré hors-sujet s'il utilise des flottants, non supportés par BINSEC, ou des primitives d'accès matériel, non modélisables en C. Le rejet des blocs est principalement issu de la non-conformité des déclarations des entrées/sorties/modifications. Certains morceaux de ffmpeg communiquent ainsi entre eux en séquence par registres xmm sans le déclarer dans leur interface. Nous signalons ceci dans notre prototype comme une erreur et rejetons ces morceaux.

Parmi les morceaux considérés, c'est-à-dire 75% (273/365) des morceaux initiaux, le portage est *réussi et validé à 100%*.

4.2 Adéquation aux analyses formelles de programme

Nous sélectionnons deux outils représentant des familles distinctes de techniques formelles utilisées dans l'industrie actuellement : l'interprétation abstraite [20, 21] et la vérification déductive [27, 29]. Nous prenons un représentant par catégorie dans Frama-C [34] : le greffon EVA [11] d'interprétation abstraite et le greffon WP [7] de vérification déductive. Frama-C a un support limité de l'assembleur en ligne fondé sur l'interface du morceau en question, traduit en annotations logiques assigns. Chaque greffon est responsable de la gestion de ces annotations en fonction de sa sémantique sous-jacente.

Interprétation abstraite. Le greffon EVA prend en compte les annotations logiques assigns générées, résultant de fait dans une sur-approximation des valeurs possibles pour lesdites variables. Ici, nous montrons comment notre traduction 1) améliore la précision de l'analyse ; 2) impacte les alarmes levées pré- et post-traduction.

Table 2 – Impact du portage sur Frama-C EVA

	ALSA	ffmpeg	GMP	Total	
Fonctions C	22	19	16	57	
Blocs asm	25	26	122	173	
Fonctions avec retour (non-void)	0	9	10	19	
Amélioration de la précision du retour	—	9	1	10	52%
Fonctions avec alarme initiale C	2	9	16	27	
Réduction alarmes C	2	9	13	24	89%
Nouvelles alarmes mémoires asm	12	1	0	13	23%
Impact positif	14	15	13	42	74%

La notion d'*impact positif* englobe ici l'amélioration de la précision, la réduction des alarmes C et l'ajout d'alarmes spécifiques au code porté depuis l'assembleur.

Pour ce faire, nous exécutons (abstraitement) **57** fonctions, comprenant 173 blocs assembleur, avec des valeurs abstraites d'entrées (intervalles). Cela nous permet de voir, par rapport à la situation initiale sans portage, si la valeur de retour est plus précise (quand elle existe) ou si certaines alarmes existantes ont pu être désactivées.

La Table 2 résume les résultats obtenus. Nous pouvons constater qu'il y a presque toujours (24/27) réduction d'alarmes post-portage dans le code C commun. Ceci est directement lié à une amélioration générale de la précision de l'analyse puisque les variables modifiées dans le code porté sont maintenant visibles par EVA. Dans la moitié des cas environ (10/19), nous arrivons également à gagner en précision sur le retour de fonction. Au total, 30/33 ($\approx 90\%$) fonctions avec retour ou alarme initiale exhibent une de ces deux améliorations de précision.

Le code C porté contient maintenant lui aussi des alarmes (13/57), auparavant indétectables, qui doivent être prises en compte. Ainsi, nos exemples exhibent d'ailleurs un cas relevant du bogue potentiel dans ffmpeg : le code peut ainsi accéder à l'index -1 du tampon d'entrée. En outre, ce code ne contient aucune documentation ou assertion qui montre que le programmeur avait perçu ce problème : c'est pourquoi nous parlons de bogue potentiel. Sans analyser l'ensemble du programme, nous ne pouvons cependant en déduire qu'il existe un chemin d'exécution qui déclenchera ce bogue potentiel. De manière générale, le code porté met au jour des alarmes additionnelles d'accès mémoire ou de débordement potentiel.

En résumé, sur 74% (42/57) des fonctions au total, nous arrivons à observer un impact positif (plus de précision, extinction des alarmes résultant de l'opacité du code assembleur ou alarmes mémoires dans le code porté) pour la qualité de l'analyse résultante.

Vérification déductive. Pour évaluer notre portage vis-à-vis de la vérification déductive, nous prenons 10 fonctions optimisées en assembleur, dont nous savons vérifier la version C initiale, et le greffon WP [13] de Frama-C. Cela nous permet d'avoir un squelette de preuve sur lequel bâtir la preuve de la version portée. Les 10 exemples détaillés en Table 3 consistent pour moitié d'exemples synthétiques, pour moitié provenant de sources extérieures.

Pour chacune de ces fonctions, nous essayons de prouver les mêmes propriétés fonctionnelles avec deux niveaux d'optimisation : le premier est *naïf* (passes 1, 2, 5 et 6 du portage vers C détaillé en Sec. 3.2), le second, TInA, comprend toutes les passes (1 à 9). Pour référence, nous essayons aussi de démontrer le programme sans portage — dans ce cas le contenu du morceau

Table 3 – Détail des fonctions testées pour Frama-C WP

	Fonction	Description	Origine
Calculs	saturated_sub	Maximum entre 0 et la soustraction entière (« grands entiers ») des 2 entrées	—
	saturated_add	Minimum entre MAX_UINT et l'addition entière (« grands entiers ») des 2 entrées	—
	log2	Plus grande puissance de 2 d'un entier	—
	mid_pred	Valeur médiane entre 3 entrées	ffmpeg
Tableaux	memset	Remplit le contenu du tableau via l'entrée	—
	count	Compte le nombre d'occurrences des entrées dans le tableau	ACSL by example
	max_element	Premier index du plus grand élément du tableau	ACSL by example
	sum_array	Somme des éléments du tableau	ACSL by example
Chaînes de caractères	streq	Teste l'égalité de deux chaînes de caractères	—
	strnlen	Calcule la taille de la chaîne (ou du tampon en l'absence de `'\0'`)	fast strlen

Table 4 – Impact du portage sur Frama-C WP

		Portage						
		Sans		Naïf			TInA	
	Instructions	OPs[a]		OPs		Invs.[b]	OPs	Invs.
saturated_sub	2	✗	3 / 6	✓	**29 / 29**	0	✓ **7 / 7**	0
saturated_add	2	✗	3 / 6	✗	27 / 29	0	✓ **7 / 7**	0
log2	1	✗	2 / 4	✗	24 / 28	5	✓ **16 / 16**	5
mid_pred	7	✗	3 / 12	✗	61 / 68	0	✓ **19 / 19**	0
memset	9	✗	1 / 3	✗	38 / 44	0	✓ **19 / 19**	0
count	8	✗	3 / 5	✗	56 / 61	4	✓ **17 / 17**	4
max_element	10	✗	3 / 7	✗	59 / 69	7	✓ **33 / 33**	7
sum_array	20	✗	3 / 5	✗	333 / 336	7	✓ **21 / 21**	7
streq	10	✗	3 / 6	✗	139 / 146	6	✓ **18 / 18**	6
strnlen	16	✗	3 / 8	✗	69 / 79	6	✓ **27 / 27**	6

[a] Obligations de Preuve [b] Invariants

d'assembleur est invisible. Les résultats sont résumés en Table 4. TInA, comparé à un portage naïf, permet à la fois de réduire le nombre d'obligations de preuve car le code comporte moins d'instructions, et de plus haut niveau, et de faire démontrer lesdites obligations (*100% de réussite* sur les 10 exemples). Ces résultats démontrent que les passes supplémentaires implémentées, potentiellement superflues concernant le critère de correction, sont en réalité nécessaires pour permettre l'intégration dans un processus de vérification.

4.3 Conclusion

Les expériences menées établissent que la technique proposée permet de traiter le code assembleur en ligne dans un contexte C. Ainsi, tous les codes assembleur en ligne à la portée de notre outil sont portés et validés (Sec. 4.1).

Par ailleurs, le code porté exhibe un bon degré de *vérifiabilité* (Sec. 4.2). Pour l'analyse par interprétation abstraite (EVA), le portage a un impact positif, soit en réduisant le nombre d'alarmes du code C commun, soit en améliorant la précision, soit encore en mettant au jour de nouvelles alarmes mémoire du code porté, qui sont parfois des bogues potentiels du programme.

Le cas de la vérification déductive (WP) montre que le portage a de plus besoin de passes de transformations pour permettre de totalement vérifier en pratique nos exemples.

5 Travaux connexes

Portage de code assembleur et vérification. Maus [37, 36] propose une méthode générique qui simule le comportement des instructions assembleur dans une machine virtuelle écrite en C. Ces travaux proviennent du projet Verisoft pour vérifier le code d'un hyperviseur qui contient du code mixte bas niveau. La technique de Maus utilise VCC [18] pour écrire et démontrer les conditions de vérification de l'état de la machine. Contrairement à notre approche, le code virtuel contient tous les détails (par exemple les indicateurs) du code bas niveau alors que nous nous efforçons d'obtenir un code le plus haut niveau possible.

Des travaux successifs de Schmaltz et Shadrin [47] visent à prouver l'adéquation au niveau de l'interface binaire-programm (*ABI*) de la partie assembleur. Cette méthode est cependant circonscrite à l'assembleur MASM (Microsoft Macro Assembler) et Windows. Notre méthode, bien qu'ici appliquée à l'assembleur en ligne de GCC (et clang), est effectivement indépendante du dialecte d'assembleur, car elle fonctionne à partir d'analyseurs de code binaire applicables à un plus grand nombre d'architectures.

Fehnker et coll. [26] traitent l'assembleur en ligne pour l'architecture ARM à travers l'usage de *model checking* pour analyser du code mixte. Cette solution est cependant purement syntaxique : ainsi, elle se limite à un type de dialecte assembleur pour une architecture, mais elle perd aussi la correction que nous visons. Cette perte peut être un compromis adéquat, mais pas dans le cadre d'analyses formelles correctes comme l'interprétation abstraite.

Corteggiani et coll. [19] utilise du portage de code. Cependant, leur but est de faire des analyses symboliques dynamiques sur le code porté — sans viser d'autres analyses. Or, certaines analyses vont fonctionner sans aucune technique avancée de traduction alors que d'autres vont nécessiter plus d'effort. Nous avons vu par exemple que le greffon WP de Frama-C nécessitait certaines passes d'optimisations pour être utilisé tel quel. Les analyses de type exécution symbolique sont a priori celles qui ont le moins besoin de transformation pour fonctionner. Par ailleurs, la correction de la démarche n'est pas abordée.

Décompilation. Cette technique [39] a pour but de recouvrer le code source d'origine (ou un très proche) d'un code exécutable. Cet objectif, très difficile, nécessite de rechercher l'information perdue durant la compilation [14, 39]. En dépit de progrès récents [10], la décompilation reste un défi scientifique ouvert. Cependant, cela permet tout de même d'améliorer la compréhension d'un programme, par exemple en rétro-ingénierie. Ce but n'est pas toujours en accord avec celui que nous avons de nous insérer dans un processus de vérification — par exemple ce n'est pas forcément nécessaire de produire un code source en tout point valide.

Schulte et coll. [48] utilisent pour garantir la correction une technique *search-based* afin de produire un code source qui se compile en code binaire égal octet pour octet au code binaire d'origine. Cette technique, lorsqu'elle termine, assure par construction la correction mais elle ne s'applique pour l'instant en pratique qu'à de petits exemples, avec un succès mitigé.

Nous réutilisons également des technique de rétro-ingénierie [35, 46] permettant d'inférer le type des registres et zones mémoires via la façon dont ils sont utilisés. Ceci nous aide à renforcer notre « système de types » bien que nous ne construisions aucun type non dérivé de celui des entrées, et donc connu par l'interface.

En décompilation, un grand défi consiste à récupérer la liste des instructions et le graphe de flot de contrôle du code [1, 38]. Bien que ce problème soit très difficile, notamment dans le

cas de code offensif de type malware, le code usuel a une bien plus grande régularité, avec un nombre conséquent de motifs, permettant en pratique une bonne reconstruction en utilisant des méthodes incorrectes — donc sans garantie. Notre cas est encore plus favorable : les morceaux d'assembleur sont en général encore plus limités au niveau structurel (flot de contrôle simple, pas de saut calculé dynamiquement) et nous avons accès à l'ensemble de la chaîne de compilation.

Analyse de programme au niveau binaire. Depuis plus d'une décennie maintenant, la communauté d'analyse de programme a consacré d'importants efforts pour traiter directement le code exécutable [2] soit pour pouvoir analyser les codes dont le source n'est pas disponible (composants sur étagère, code hérité, malware), soit pour vérifier le code tel qu'il s'exécute véritablement. Ces efforts se sont concentrés sur le recouvrement d'abstractions de haut niveau [5, 24, 32, 43, 49] et le calcul d'invariants.

En outre, plusieurs « porteurs » (*lifters*) ont été proposés récemment, réduisant ainsi les divers jeux d'instructions à un petit nombre de primitives bien définies d'un langage intermédiaire. Si ceux-ci sont éprouvés en pratique [31], nous pourrions en dériver encore plus de confiance s'ils étaient générés automatiquement, par exemple à partir de spécifications similaires à celles, récentes, de ARM [42].

Validation de traduction et équivalence de code. Pour garantir la sûreté de notre portage, nous nous inscrivons dans la mouvance de la validation de la traduction [40, 45, 50], technique utilisée par exemple pour l'allocation de registre de CompCert [8]. Ceci nous permet à moindre coût de faire reposer nos besoins formels de correction sur des outils disponibles et éprouvés (en l'occurrence les solveurs SMT), utilisables en boîte noire, plutôt de nous atteler à une démonstration formelle complète de la correction de l'ensemble de la chaîne.

6 Conclusion

Nous proposons une méthode sûre permettant l'analyse de code C/C++ comportant de l'assembleur en ligne. Cette méthode génère un code C bien structuré permettant la réutilisation de techniques de vérification existantes pour le C, en utilisant des transformations successives sur un langage intermédiaire extrait de l'exécutable. La correction de la méthode est assurée par validation de la traduction. Nos expérimentations sur des codes libres d'envergure écrits en C démontre l'applicabilité de la méthode et son intérêt pratique pour la vérification.

Références

[1] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos. An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries. In T. Holz and S. Savage, editors, *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.*, pages 583–600. USENIX Association, 2016.

[2] G. Balakrishnan and T. W. Reps. WYSINWYX : what you see is not what you execute. *ACM Trans. Program. Lang. Syst.*, 32(6):23:1–23:84, 2010.

[3] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The static driver verifier research platform. In T. Touili, B. Cook, and P. B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 119–122. Springer, 2010.

[4] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA Framework for Binary Code Analysis. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*

- 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, volume 6806 of *Lecture Notes in Computer Science*, pages 165–170. Springer, 2011.

[5] S. Bardin, P. Herrmann, and F. Védrine. Refinement-Based CFG Reconstruction from Unstructured Programs. In R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 2011.

[6] C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking.*, pages 305–343. Springer, 2018.

[7] P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye. *WP Manual*, Frama-C Chlorine-20180501 edition, 2018.

[8] S. Blazy, B. Robillard, and A. W. Appel. Formal Verification of Coalescing Graph-Coloring Register Allocation. In A. D. Gordon, editor, *Programming Languages and Systems, 19th European Symposium on Programming, ESOP 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6012 of *Lecture Notes in Computer Science*, pages 145–164. Springer, 2010.

[9] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. *BAP : A Binary Analysis Platform* , pages 463–469. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[10] D. Brumley, J. Lee, E. J. Schwartz, and M. Woo. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In S. T. King, editor, *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, pages 353–368. USENIX Association, 2013.

[11] D. Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions:EVA, an Evolved Value Analysis for Frama-C.* PhD thesis, University of Rennes 1, France, 2017.

[12] C. Cadar, D. Dunbar, and D. R. Engler. KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[13] N. Carvalho, C. da Silva Sousa, J. S. Pinto, and A. Tomb. Formal Verification of kLIBC with the WP Frama-C Plug-in. In J. M. Badger and K. Y. Rozier, editors, *NASA Formal Methods - 6th International Symposium, NFM 2014, Houston, TX, USA, April 29 - May 1, 2014. Proceedings*, volume 8430 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2014.

[14] B.-Y. E. Chang, M. Harren, and G. C. Necula. *Analysis of Low-Level Code Using Cooperating Decompilers*, pages 318–335. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[15] C. Cifuentes. Interprocedural data flow decompilation. *J. Prog. Lang.*, 4(2):77–99, 1996.

[16] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Softw., Pract. Exper.*, 25(7):811–829, 1995.

[17] C. Cifuentes, D. Simon, and A. Fraboulet. Assembly to High-Level Language Translation. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pages 228–237. IEEE Computer Society, 1998.

[18] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. *VCC : A Practical System for Verifying Concurrent C*, pages 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[19] N. Corteggiani, G. Camurati, and A. Francillon. Inception : System-Wide Security Testing of Real-World Embedded Systems Software. In W. Enck and A. P. Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018.*, pages 309–326. USENIX Association, 2018.

[20] P. Cousot and R. Cousot. Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[21] P. Cousot and R. Cousot. Abstract interpretation : past, present and future. In T. A. Henzinger

and D. Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 2:1–2:10. ACM, 2014.

[22] D. Delmas and J. Souyris. Astrée : From Research to Industry. In H. R. Nielson and G. Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.

[23] A. Djoudi and S. Bardin. *BINSEC : Binary Code Analysis with Low-Level Regions* , pages 212–217. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[24] A. Djoudi, S. Bardin, and É. Goubault. *Recovering High-Level Conditions from Binary Programs* , pages 235–253. Springer International Publishing, Cham, 2016.

[25] DWARF Debugging Information Format Committee. *DWARF Debugging Information Format 5*, 2017.

[26] A. Fehnker, R. Huuck, F. Rauch, and S. Seefried. Some Assembly Required - Program Analysis of Embedded System Code. In *2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 15–24, Sept 2008.

[27] R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[28] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE : whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[29] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Commun. ACM*, 12(10):576–580, 1969.

[30] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, September 2016.

[31] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. Testing intermediate representations for binary analysis. In G. Rosu, M. D. Penta, and T. N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 353–364. IEEE Computer Society, 2017.

[32] J. Kinder and D. Kravchenko. Alternating Control Flow Reconstruction. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation - 13th International Conference, VMCAI 2012, Philadelphia, PA, USA, January 22-24, 2012. Proceedings*, volume 7148 of *Lecture Notes in Computer Science*, pages 267–282. Springer, 2012.

[33] J. C. King. Symbolic Execution and Program Testing. *Commun. ACM*, 19(7):385–394, July 1976.

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

[35] J. Lee, T. Avgerinos, and D. Brumley. TIE : Principled Reverse Engineering of Types in Binary Programs. In *NDSS*, 2011.

[36] S. Maus. *Verification of hypervisor subroutines written in Assembler (Verifikation von Hypervisorunterrutinen, geschrieben in Assembler)*. PhD thesis, University of Freiburg, Germany, 2011.

[37] S. Maus, M. Moskal, and W. Schulte. *Vx86 : x86 Assembler Simulated in C Powered by Automated Theorem Proving*, pages 284–298. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[38] X. Meng and B. P. Miller. Binary code is not easy. In A. Zeller and A. Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 24–35. ACM, 2016.

[39] M. O. Myreen, M. J. C. Gordon, and K. Slind. Machine-Code Verification for Multiple Architectures - An Application of Decompilation into Logic. In *2008 Formal Methods in Computer-Aided Design*, pages 1–8, Nov 2008.

[40] G. C. Necula. Translation validation for an optimizing compiler. In M. S. Lam, editor, *Proceedings*

of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000, pages 83–94. ACM, 2000.

[41] P. W. O'Hearn. From Categorical Logic to Facebook Engineering. In *30th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2015, Kyoto, Japan, July 6-10, 2015*, pages 17–20. IEEE Computer Society, 2015.

[42] A. Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In R. Piskac and M. Talupur, editors, *2016 Formal Methods in Computer-Aided Design, FMCAD 2016, Mountain View, CA, USA, October 3-6, 2016*, pages 161–168. IEEE, 2016.

[43] T. Reinbacher and J. Brauer. Precise control flow reconstruction using boolean logic. In S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, editors, *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*, pages 117–126. ACM, 2011.

[44] M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck. An Analysis of x86-64 Inline Assembly in C Programs. In *Proceedings of the 14th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2018, Williamsburg, VA, USA, March 25-25, 2018*, pages 84–99. ACM, 2018.

[45] X. Rival. Certification of compiled assembly code by invariant translation. *STTT*, 6(1):15–37, 2004.

[46] E. Robbins, A. King, and T. Schrijvers. From MinX to MinC : Semantics-driven Decompilation of Recursive Datatypes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 191–203, New York, NY, USA, 2016. ACM.

[47] S. Schmaltz and A. Shadrin. *Integrated Semantics of Intermediate-Language C and Macro-Assembler for Pervasive Formal Verification of Operating Systems and Hypervisors from VerisoftXT*, pages 18–33. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[48] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Logino. Evolving Exact Decompilation. In *BAR 2018, Workshop on Binary Analysis Research, San Diego, California, USA, February 18, 2018*, 2018.

[49] A. Sepp, B. Mihaila, and A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In M. Pinzger, D. Poshyvanyk, and J. Buckley, editors, *18th Working Conference on Reverse Engineering, WCRE 2011, Limerick, Ireland, October 17-20, 2011*, pages 357–366. IEEE Computer Society, 2011.

[50] T. A. L. Sewell, M. O. Myreen, and G. Klein. Translation validation for a verified OS kernel. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 471–482. ACM, 2013.

[51] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK : (State of) The Art of War : Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

Articles courts

Learn-OCaml : un Assistant à l'Enseignement d'OCaml

Çagdas Bozman[2], Benjamin Canou[1], Roberto Di Cosmo[3,5,6,7], Pierrick Couderc[2], Louis Gesbert[2], Grégoire Henry[1], Fabrice Le Fessant[2], Michel Mauny[3,4], Carine Morel[4,5], Loïc Peyrot[4,5], et Yann Régis-Gianas[3,4,5,6]

[1] Nomadic Development
[2] OCamlPro
[3] Inria
[4] Fondation OCaml
[5] Université Paris Diderot
[6] IRIF
[7] Software Heritage
[8] ENSTA

La plateforme Learn-OCaml est un assistant à l'enseignement du langage de programmation OCaml, développée dans le cadre du projet éponyme porté par la Fondation OCaml, qui vise à soutenir et développer l'usage d'OCaml dans l'enseignement. La plateforme permet d'écrire des exercices munis de correcteurs automatiques qui peuvent tester non seulement la correction fonctionnelle des programmes soumis par les étudiants mais aussi la façon dont ces programmes sont écrits ou calculent.

Dans cet article, nous présentons la plateforme dans son ensemble, le projet dans lequel s'inscrit son développement, et ses fonctionnalités principales, à travers notamment l'écriture d'un exercice et de son correcteur ainsi qu'un premier retour sur deux expériences d'usage menées à l'Université McGill et à l'Université Paris Diderot.

1 Introduction

La Fondation OCaml, créée en juin 2018, a pour mission de promouvoir l'usage d'OCaml, de soutenir son développement et celui de son écosystème. La Fondation prend la suite du Consortium Caml, créé à l'Inria en 2001, et qui a réuni depuis cette date une quinzaine d'utilisateurs industriels du langage. Abritée par la Fondation Partenariale Inria, la Fondation OCaml présente un certain nombre d'avantages par rapport au Consortium : elle dispose en effet des facilités de gestion fournies par sa fondation abritante – une organisation de droit privé – ainsi que la fiscalité avantageuse qui est offerte à ses mécènes. De plus, en tant que fondation, la Fondation OCaml a pu se doter de facultés «redistributives» qui lui permettent d'apporter son soutien à des actions conformes à ses missions qui peuvent être opérées par d'autres (organisations ou personnes physiques) que par la Fondation elle-même.

Le projet Learn-OCaml est l'une des premières actions de la Fondation OCaml : il vise à promouvoir l'usage d'OCaml dans l'enseignements et à faciliter aux étudiants, formateurs et ingénieurs, l'accès à des ressources pédagogiques de qualité et librement accessibles. Le développement de la plateforme logicielle Learn-OCaml est l'un des points importants de ce projet.

Cet article présente la plateforme logicielle Learn-OCaml développée par la fondation OCaml et OCamlPro. Elle s'appuie sur une suite logicielle développée par OCamlPro depuis plusieurs années et sur une plateforme d'exercices de programmation en ligne développée avec le soutien de l'IRILL et de SAPIENS, destinée à être au cœur du MOOC "Introduction to functional programming in OCaml". Ses spécificités ont déjà été présentées dans un article publié à ICFP'17[4]. Cet article reprend bien sûr une partie des éléments présentés à ICFP'17, mais

illustre aussi certaines des fonctionnalités introduites par la nouvelle version de la plateforme, typiquement la possibilité de la déployer en dehors de l'infrastructure d'un MOOC et son adaptation au contexte d'un enseignement plus classique.

La publication de cet article aux JFLA 2019 est aussi un moyen d'inviter la communauté enseignante francophone à enrichir cette plateforme par l'ajout de nouveaux traits ou par la création de contenu pédagogique, et à l'utiliser dans des enseignement nouveaux ou existants.

2 Vue d'ensemble de la plateforme Learn-OCaml

Fonctionnalités principales Learn-OCaml assiste l'enseignement du langage de programmation OCaml en (i) donnant accès à des exercices de programmation corrigés automatiquement ; (ii) assurant le suivi de la progression des élèves ; (iii) permettant à l'enseignant de personnaliser le chemin d'apprentissage des élèves.

La plateforme Learn-OCaml suit une architecture client-serveur classique : son usage typique consiste donc à déployer une instance du serveur sur une machine accessible sur le réseau *via* le protocole HTTP/S et à interagir avec ce serveur à l'aide d'un client. Les clients actuels, au nombre de deux, consistent en une application web et une interface en ligne de commande. L'enseignant et l'élève utilisent le même client mais l'enseignant a accès à plus de fonctionnalités.

Serveur Le serveur prend en charge le stockage d'un ensemble d'exercices munis de correcteurs automatiques. Il stocke aussi l'historique des réponses et des résultats des élèves. Enfin, il maintient un ensemble de méta-données sur les élèves (typiquement un ensemble d'étiquettes qui permettent de structurer une cohorte en groupes de travaux dirigés, de niveaux, ...) ainsi qu'un ensemble de méta-données sur les exercices (les compétences travaillées et requises, les exercices de remédiation, des statistiques de réussites, ...).

Une des méta-données essentielles du système est une liste d'exercices associée à chaque élève. Un nouvel exercice apparaît dans cette liste s'il est ouvert à l'élève. Cette ouverture peut être illimitée dans le temps ou prendre la forme d'un "devoir" caractérisé par une date d'ouverture et une date de rendu. À tout moment, l'enseignant peut mettre à jour cette liste pour un élève en particulier, pour un groupe d'élèves ou encore pour l'ensemble des élèves.

Comme dans le cas de l'environnement d'exercices déployé pour le MOOC OCaml[4], la correction automatique des réponses de l'élève est faite par le client [1], directement sur la machine de l'élève, qui transmet ensuite le rapport de correction au serveur pour stockage.

Cette architecture réduit considérablement la charge sur le serveur, qui ne doit gérer que l'envoi et la collecte des données. Elle raccourcit aussi le temps qui prend un cycle édition-correction, qui ne dépend que de l'algorithme de correction automatique et des ressources de la machine de l'étudiant ; le nombre d'étudiants travaillant simultanément sur la plateforme n'intervient pas. L'élève travaille ainsi plus efficacement, sans être tributaire d'une éventuelle surcharge du serveur ou de problèmes de connectivité.

Dans la plateforme Learn-OCaml, une fonctionnalité a été ajoutée pour permettre à l'enseignant de déclencher une correction des réponses des élèves côté serveur. Cela permet à l'enseignant de revalider les résultats des élèves sur le serveur et d'améliorer la fiabilité des évaluations, en rendant vaine toute tentative de modifier le client pour qu'il envoie des rapports de correction forgés pour associer la note maximale à des réponses invalides.

1. Historiquement, cette fonctionnalité est héritée du projet TryOCaml, développé par OCamlPro.

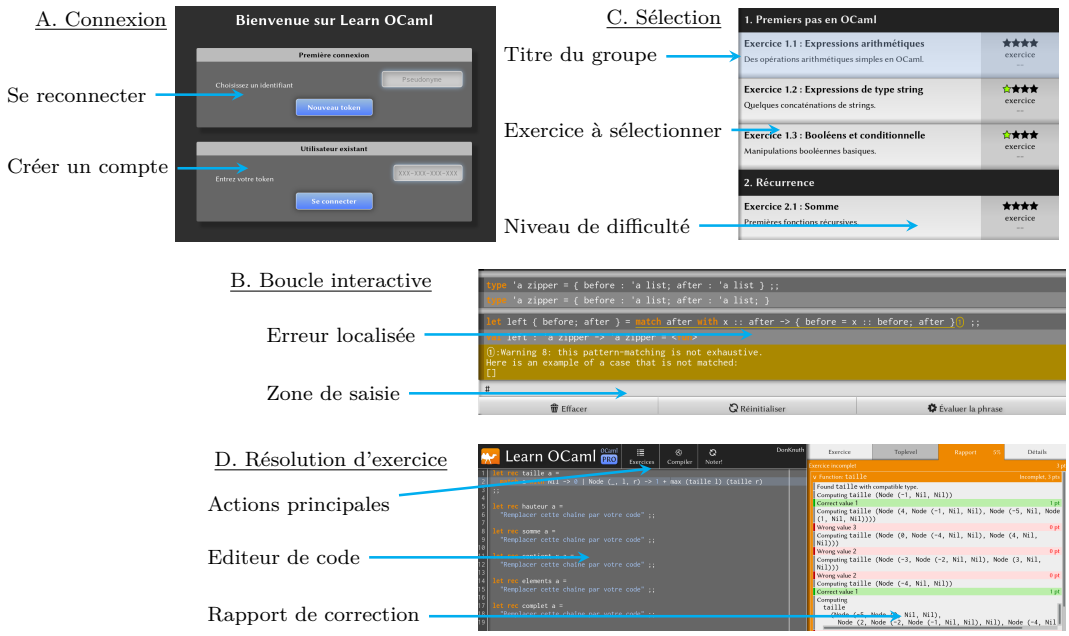


Figure 1 – Captures d'écran du client web de Learn-OCaml.

Pour finir, le serveur peut être déployé localement sur la machine de l'enseignant qui peut ainsi tester ses correcteurs sans avoir à les mettre en production. Ce déploiement s'appuie sur Docker[5] et peut donc s'effectuer sous GNU/Linux, MacOS X et MS/Windows. Sous les systèmes UNIX, pour déployer une instance locale de Learn-OCaml, il suffit ainsi d'exécuter : `docker run --rm -v \`pwd\`:/repository:ro -p 80:8080 ocamlsf/learn-ocaml:dev` depuis un répertoire contenant un dépôt d'exercices [2]. Un serveur est alors disponible à l'adresse `http://localhost:80`.

Client web Si on se connecte avec un navigateur web à l'URL racine d'un serveur Learn-OCaml, on accède au client web de la plateforme. Nous allons maintenant commenter rapidement les captures d'écran des figures 1 et 2. Le client permet ainsi de créer un compte sur la plateforme (écran A). À ce compte est associé un identifiant et un jeton unique. Ce jeton sert à s'authentifier lors des sessions futures. Une fois connecté à la plateforme, deux boutons intitulés "Exercice" et "Toplevel" permettent de choisir une activité [3]. L'écran B correspond à l'activité "Toplevel" : il s'agit d'un boucle interactive OCaml qui tourne dans le navigateur et permet à l'élève d'effectuer des tests. L'avantage de cette boucle interactive nichée dans le navigateur par rapport à la boucle d'interaction classique est qu'elle permet de mieux localiser et mettre en page les messages d'erreur : on profite en effet de l'affichage HTML pour structurer l'affichage et le rendre plus clair qu'en mode textuel. L'écran C correspond à l'activité "Exercice" mentionnée plus haut. Les exercices sont en général regroupés thématiquement et chaque exercice est décrit par un titre, une description courte et un niveau de difficulté quantifié par un nombre

2. Nous allons voir comment créer un tel dépôt dans la section 3.
3. Une activité de consultation de tutoriels est également disponible mais nous ne la documentons pas ici.

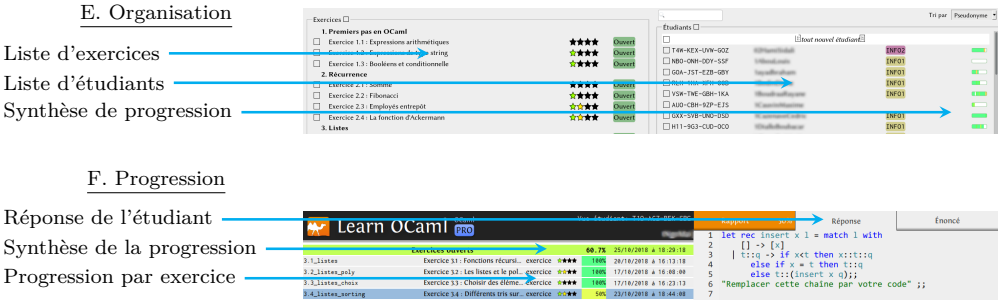


Figure 2 – Captures d'écran des activités de l'enseignant.

d'étoiles (plus un exercice a d'étoiles et plus il est jugé difficile). Enfin, l'écran D offert à l'élève correspond à l'activité de résolution d'exercice à proprement parler.

À gauche de l'écran, on distingue une zone d'édition de code source surmontée de plusieurs boutons. La zone d'édition est une version modifiée par OCamlPro de l'éditeur de texte ACE[1]. Elle prend en charge la coloration syntaxique mais aussi une forme d'indentation automatique (implémentée grâce à ocp-indent). Le bouton "Compiler" permet de s'assurer que le programme est bien formé. Le bouton "Noter" exécute la correction automatique. Cette dernière produit un rapport de correction (que l'on distingue à droite de l'écran). Chaque test rapporte un point s'il réussit. D'autres onglets sont accessibles à l'élève : l'onglet "Exercice" affiche l'énoncé de l'exercice, l'onglet "Toplevel" permet d'exécuter une boucle interactive initialisée par le contexte de travail de l'exercice, l'onglet "Détails" permet de visualiser les méta-données de l'exercice en cours de résolution. Ces méta-données incluent par exemple les compétences travaillées ou requises par l'exercice, son identifiant ou encore le nom de ses auteurs.

La figure 2 donne une idée des activités supplémentaires accessibles aux enseignants. L'écran E correspond à une activité de visualisation de l'ensemble des exercices et des élèves. On peut ainsi constater l'avancement de chaque élève ou d'un groupe d'élèves grâce à des barres de progression. C'est grâce à cette interface que l'enseignant modifie le statut d'un exercice : un exercice peut être fermé, ouvert à tous ou alors affecté à un groupe d'élèves (avec éventuellement des contraintes de temps pour sa réalisation). Cette interface permet aussi d'affecter des étiquettes aux élèves pour créer des groupes (de classes, de niveaux, *etc.*). L'écran F donne une vue plus précise de la progression d'un élève : ce tableau de bord affiche le score de l'étudiant pour chaque exercice et pour chaque compétence travaillée. Ces données sont téléchargeables au format CSV pour que l'enseignant puisse leur appliquer un traitement spécifique à ses besoins.

Interface en ligne de commande Le client web est utile pour des élèves qui débutent en programmation : il n'y a aucun coût d'installation et l'interface s'est révélée suffisamment intuitive d'utilisation pour que les élèves se mettent directement à travailler sur les exercices de programmation. Cependant, cette interface s'avère inadaptée pour des élèves ayant déjà une expérience significative en développement logiciel. En effet, dans ce cas, il est fort probable que l'élève ait déjà l'habitude d'un usage sophistiqué de son éditeur de texte favori et considère le passage à l'éditeur en ligne comme une régression [4].

4. Typiquement, le mécanisme d'indentation automatique est utile aux élèves ne sachant pas mettre en forme leur code source... mais agace ceux qui possèdent déjà cette compétence.

descr.md

```
# Arbres binaires de recherche
Soient `thing` un type et `compare : thing -> thing -> int` l'implémentation d'un préordre total
sur les valeurs de type `thing`. Deux valeurs `a` et `b` de type `thing` sont dites équivalentes
ssi `compare a b = 0`, `a` est plus petit que `b` ssi `compare a b < 0`, `a` est plus grand que `b`
ssi `compare a b > 0`.

Un *arbre binaire de recherche* est un arbre binaire étiqueté par des valeurs de type `thing` tel que:
- `Leaf x` est un arbre binaire de recherche;
- `Node (l, x, r)` est un arbre binaire de recherche ssi
  - `l` et `r` sont des arbres binaires de recherche et
  - pour toute étiquette `y` de `l`, `y` est plus petit que `x` et
  - pour toute étiquette `y` de `r`, `y` est plus grand que `x`.

**Question** Écrire une fonction récursive `mem : thing -> thing bst -> bool` telle que `mem x t`
renvoie `true` ssi une valeur de type `thing` équivalente à `x` apparaît dans `t`.
Essayez de minimiser le nombre de comparaisons utilisées par `mem`.
```

meta.json

```
{ "learnocaml_version" : "2", "kind" : "exercise", "stars" : 2,
   "title" : "Binary Search Trees", "short_description"  : "How to search in a binary search tree.",
   "authors" : [ [ "Don Knuth", "<don.knuth@does-not-read-emails>" ] ],
   "focus":[ "pattern-matching", "complexity" ],
   "requirements" : [ "recursion", "parametric type", "algebraic datatype" ] }
```

template.ml

```
let mem x t = "À compléter"
```

solution.ml

```
let rec mem x = function
  | Leaf y ->
     compare x y = 0
  | Node (l, y, r) ->
     let cmp = compare y x in
     if cmp < 0 then mem x l
     else if cmp > 0 then mem x r
     else true
```

prelude.ml

```
type 'a bst =
  | Leaf of 'a
  | Node of 'a bst * 'a * 'a bst
```

prepare.ml

```
let comparison_counter = ref 0
let reset_comparison_counter () = comparison_counter := 0
let consumed_comparisons () = !comparison_counter
module Thing : sig
  type t
  val compare : t -> t -> int
  val sample : unit -> t
  val shake : t -> t
end = struct
  type t = Thing of int * int
  let sample () = Thing (Random.bits (), Random.bits ())
  let compare (Thing (x, _)) (Thing (y, _)) =
    incr comparison_counter;
    compare x y
  let shake (Thing (x, _)) = Thing (x, Random.bits ())
end
type thing = Thing.t
let sample_thing = Thing.sample
```

Figure 3 – Exercice : La recherche d'un élément dans un arbre binaire de recherche.

Pour cette raison, la plateforme fournit aussi un outil en ligne de commande nommé `learn-ocaml-client` qui permet d'utiliser le sous-système de correction automatique depuis son environnement de développement favori. Il suffit ainsi d'éditer sa réponse dans un fichier source et d'invoquer `learn-ocaml-client` en lui passant l'identifiant de l'exercice et le chemin vers le fichier source utilisé pour que Learn-OCaml déclenche sa correction automatique (locale) et transmette la réponse et le rapport au serveur. Ce client s'appuie sur une API HTTP qui est exposée par le serveur. L'existence de cette API rend envisageable le développement d'autres clients dans le futur.

3 Les fonctionnalités de Learn-OCaml par l'exemple

Pour donner une idée des fonctionnalités de correction automatique de Learn-OCaml, nous allons commenter un exemple d'exercice autocorrigé. Les fichiers nécessaires à la définition de

test.ml

```
open Test_lib
open Report

let failure msg = [Message ([ Text msg ], Failure)]
let success msg = [Message ([ Text msg ], Success 1)]

let rec grade () =
  let expected_type = [%ty : thing -> thing bst -> bool ] in (
  check_no_while @@ fun () ->
  let before_user _ _ = reset_comparison_counter ()
  and sampler () =
    let t = sample_bst () in
    if Random.int 2 = 0 then (Thing.shake (pick t), t) else (sample_thing (), t)
  and after _ tree _ _ =
    if consumed_comparisons () > height tree then
      failure "Le nombre de comparaisons doit être inférieur à la hauteur de l'arbre!"
    else
      success "Bravo"
  and manual_tests =
    let x = sample_thing () in [ (x, Leaf x); (sample_thing (), Leaf x) ]
  in
  test_function_2_against_solution ~gen:10 ~before_user ~sampler ~after expected_type "mem" manual_tests
) |> set_result
and sample_bst () =
  if Random.int 2 = 0 then Leaf (sample_thing ()) else Node (sample_bst (), sample_thing (), sample_bst ())
and pick = function
  | Leaf x -> x
  | Node (l, y, r) -> match Random.int 3 with | 0 -> y | 1 -> pick l | 2 -> pick r | _ -> assert false
and height = function Leaf _ -> 1 | Node (l, x, r) -> 1 + max (height l) (height r)
and check_no_while cb =
  find_binding code_ast "mem" @@ fun expr ->
  let flag = ref false in
  let on_expression = Parsetree.(function { pexp_desc = Pexp_while _ } -> flag := true; [] | _ -> []) in
  ast_check_expr ~on_expression expr |> ignore;
  if !flag then failure "Utilisez la récursion! Pas la boucle while" else cb ()

let () = grade ()
```

Figure 4 – Exercice : La recherche d'un élément dans un arbre binaire de recherche (suite).

cet exercice se trouvent dans les figures 3 et 4. On commente ces fichiers tour à tour.

descr.md L'énoncé de l'exercice est spécifié par le fichier `descr.md`. Ce fichier au format Markdown peut contenir du texte mis en page, des formules mathématiques écrites en LaTeX ou encore des images. Learn-OCaml accepte aussi des énoncés écrits au format HTML. Notons qu'il suffit d'ajouter une extension `.fr`, `.en`, etc, à la fin du nom du fichier pour préciser sa langue. Plusieurs traductions d'un même énoncé peuvent ainsi coexister.

meta.json Les métadonnées d'un exercice sont fournies sous la forme d'un fichier JSON nommé `meta.json`.

prelude.ml Dans l'exercice qui nous intéresse ici, on demande à l'élève d'implémenter une fonction de recherche dans un arbre binaire de recherche. Pour écrire cette fonction, l'étudiant doit s'appuyer sur la définition du type `bst` donnée dans le fichier `prelude.ml`. Le contenu de ce fichier fait partie de l'énoncé de l'exercice et est évalué juste avant la réponse de l'étudiant.

template.ml Il est souvent utile de proposer une base de départ à l'étudiant pour qu'il se concentre sur les aspects importants de l'exercice ou pour qu'il construise sa réponse en suivant une architecture préétablie. Ce code source à compléter est spécifié par le contenu du fichier `template.ml`.

solution.ml Pour faciliter la correction et aussi pour s'assurer de la faisabilité d'un exercice, Learn-OCaml exige que l'enseignant fournisse une solution à l'exercice dans le fichier `solution.ml`. Cette solution est systématiquement corrigée avant le déploiement pour s'assurer de sa validité.

prepare.ml Pour s'assurer que l'étudiant n'utilise que les propriétés du préordre total (i.e. la fonction `compare` et pas la fonction d'égalité générique d'OCaml), il faut se doter d'une définition rusée du type `thing` : le type `thing` doit être abstrait pour que `compare` soit l'unique opération licite sur ses habitants et il faut aussi s'assurer que les opérations de comparaison génériques d'OCaml ne soient pas compatibles avec la relation d'équivalence induite par le préordre. Ces propriétés sont garanties par l'implémentation du module `Thing` définie dans le fichier `prepare.ml`. En effet, d'une part, la signature de ce module cache la définition concrète du type `thing`. D'autre part, les valeurs de type `thing` sont des paires d'entiers dont seule la première composante est observée par le préordre et dont la seconde composante ne sert qu'à dissocier la fonction d'équivalence de l'égalité générique d'OCaml.

Pour s'assurer que l'algorithme utilisé par l'étudiant a la complexité attendue, il faut pouvoir compter les opérations de comparaison qu'il consomme. C'est le rôle des incréments de la variable globale `comparison_counter` introduite par `prepare.ml`.

Cette machinerie nécessaire à la définition du type `thing` et à l'instrumentation de la fonction de comparaison ne doit pas être montrée à l'élève car elle est relativement complexe et surtout totalement décorrélée de l'exercice de programmation dont il est question ici. Pour cette raison, le contenu du fichier `prepare.ml` est évalué avant la réponse de l'élève, et après `prelude.ml` mais contrairement à ce dernier, il n'est pas divulgué à l'élève.

test.ml Pour comprendre le rôle de la fonction `sample` définie dans `prepare.ml`, il faut maintenant se pencher sur le fichier `test.ml` qui contient le programme de correction automatique à proprement parler. Concentrons-nous donc sur la figure 4.

Dans Learn-OCaml, la correction d'un exercice se résume à la production d'un rapport de correction. En deux mots, un rapport est un document structuré qui contient des annotations d'évaluation sous la forme de points de succès (`Success of int`) ou d'échec (`Failure`). Les constructeurs de données du type `report` sont mis à disposition par le module `Report`. Les lignes 4 et 5 les utilisent pour introduire deux opérateurs `failure` et `success` qui servent à informer l'élève de la nature de son erreur éventuelle ou bien à le gratifier d'un point en cas de réussite.

Vient ensuite la fonction `grade ()` et ses fonctions auxiliaires. En première approximation, cette fonction peut se lire comme suit :

```
1  let grade () = ...
2    check_no_while (fun () -> ( ...
3    test_function_2_against_solution ~gen:10 ~sampler ~before_user ~after expected_type "mem" manual_tests
4  ) |> set_result
```

On distingue de cette façon trois parties importantes : (i) l'appel à la fonction `check_no_while` sert à vérifier que l'étudiant n'utilise pas de boucle `while` dans sa solution ; (ii) l'appel à la fonction `test_function_2_against_solution` sert à comparer la réponse de l'étudiant à la solution de référence de l'enseignant ; (iii) l'appel à la fonction `set_result` sert à transmettre le rapport à l'élève.

La première fonction `check_no_while` est implémentée par observation de l'arbre de syntaxe abstraite de la fonction `mem` de l'élève. La fonction `ast_check_expr` du module `Test_lib` réalise en effet un parcours générique de l'arbre de syntaxe que l'on peut personnaliser en lui passant une fonction `~on_expression`. Dans notre cas d'étude, on lève un drapeau lorsque la construction `while` est utilisée dans la définition de `mem`.

Les deux dernières fonctions sont elles aussi offertes par le module `Test_lib`. Il reste à expliquer la signification des arguments de la fonction `test_function_2_against_solution`. Avant toute évaluation, Learn-OCaml commence par vérifier que la fonction de l'élève a bien le type attendu spécifié par l'argument `expected_type`. Si ce n'est pas le cas, un message d'erreur de type est transmis à l'étudiant dans le rapport et aucun point ne lui est affecté. L'argument `gen` précise le nombre d'entrées utilisées pour comparer la réponse de l'élève à la solution de l'enseignant.

Ces entrées sont obtenues par le biais de deux sources distinctes. Tout d'abord, la fonction consomme tous les couples fournis manuellement par l'enseignant dans l'argument `manual_tests`. Si nécessaire, elle complète ensuite ces tests pour atteindre le nombre `gen` en utilisant le `~sampler`. Il s'agit d'une fonction sans argument qui produit un couple à chaque fois qu'on l'appelle. Dans notre exemple, la fonction `sampler` est implémentée en générant un arbre aléatoire `t` puis en décidant aléatoirement de chercher un élément existant dans l'arbre (à équivalence près, grâce à l'appel de `Thing.shake`) ou bien de chercher dans cet arbre un élément de type `thing` tiré aléatoirement.

Enfin, on peut enregistrer des actions à accomplir avant et après chaque test unitaire. Dans notre exemple, l'argument `before_user` remet à zéro le compteur de comparaisons. Quant à l'argument `after`, il sert à apporter des compléments au rapport après l'évaluation de la fonction de l'élève et de l'enseignant. On peut donc observer la valeur renvoyée par chacune d'elles mais aussi comparer les sorties standard et d'erreurs. On en profite ici pour vérifier que le nombre de comparaisons consommées par la fonction de l'élève est raisonnable.

4 Conclusion

Premiers retours d'expérience La plateforme Learn-OCaml est déjà utilisée de façon expérimentale par Brigitte Pientka à l'Université McGill au Canada et par Michele Pagani à l'Université Paris Diderot. Comme ces expériences sont en cours, il est difficile de faire des retours d'expérience significatifs. Cependant, nous pouvons déjà rapporter que (i) les équipes enseignantes ont pu traduire leurs énoncés de travaux dirigés dans le système sans difficultés particulières, la documentation de Learn-OCaml semblant prendre en charge la plupart des cas d'usage ; (ii) durant la période de préparation des cours (juillet et août 2018) et le début du semestre de cours (septembre et octobre 2018), 59 rapports de bug ont été soumis et parmi eux, seulement 2 étaient critiques ; (iii) les étudiants réalisent plus vite les feuilles d'exercices quand ils utilisent la plateforme que lors d'une séance de travaux pratiques classiques ; (iv) les étudiants utilisent la plateforme en dehors des séances de travaux pratiques.

Pour la suite Non seulement le code source de Learn-OCaml est disponible sous une licence libre[2] mais plusieurs autres projets pédagogiques collaboratifs vont maintenant s'appuyer sur cette première pierre.

Le premier d'entre eux est la création d'un grand corpus d'exercices corrigés automatiquement, sous licence libre et accessible ainsi à tous les enseignants. On espère ainsi fédérer les efforts de la communauté enseignante dans l'objectif de créer du matériel pédagogique de qualité, réutilisable en cours.

Pour les enseignants d'informatique qui ne connaîtraient pas les arcanes d'UNIX, le projet Learn-OCaml va mettre en place un serveur d'instances qui permettra à tous de déployer la plateforme pour sa classe en quelques clics. Dans le même esprit, une extension de l'application web pour y inclure un éditeur intégré d'exercices est en cours de développement à l'Université de Toulouse III par Erik Martin Dorel et ses étudiants. Enfin, un outil nommé `autogen`[3] s'appuyant sur des extensions PPX permet de simplifier la création d'exercices en produisant

les cinq fichiers décrits dans la section 3 à partir d'un unique fichier source.

Références

[1] ACE, The High Performance Editor for the Web. https://ace.c9.io/.

[2] Learn-OCaml. https://github.com/ocaml-sf/learn-ocaml.

[3] Learn-OCaml Autogen. https://github.com/ocaml-sf/learn-ocaml-autogen.

[4] Benjamin Canou, Roberto Di Cosmo, and Grégoire Henry. Scaling Up Functional Programming Education : Under the Hood of the OCaml MOOC. *PACML*, 1(ICFP), August 2017.

[5] Dirk Merkel. Docker : Lightweight Linux Containers for Consistent Development and Deployment. *Linux J.*, 2014(239), March 2014.

Formally Verified Decomposition of Non-binary Constraints into Equivalent Binary Constraints

Catherine Dubois

Samovar, ENSIIE, CNRS, Évry, France
{dubois}@ensiie.fr

Abstract

Finite domain constraint solvers, as well as SAT and SMT solvers, are used in program verification platforms and are thus included in the trusted base. It is therefore important to increase the level of confidence in these solvers, especially when the answer is UNSAT. Carlier, Dubois and Gotlieb developed, in 2012, a solver for finite domains formally verified with the help of the Coq proof assistant. However this solver enables only binary constraints, which limits its use. In this article we propose to add a pre-processor that transforms a constraint satisfaction problem (csp) containing non-binary constraints (more precisely here ternary constraints) into an equivalent binary csp. This pre-processor is developed in Coq and proven correct and complete. In particular we show that any solution of the binary csp makes it possible to build a solution of the original problem and vice versa. Transformation is based on the approach widely exploited in constraint programming known as Hidden Variable Encoding. The success key of the connection between the pre-processor and the Coq binary solver is the genericity of the latter.

1 Introduction

Finite domain constraint solvers, as well as SAT and SMT solvers, are used in program verification platforms and are thus included in the trusted base. It is therefore important to increase the level of confidence in these solvers, especially when the answer is UNSAT. Modern solvers are complex software including optimisations, heuristics, efficient data structures, etc. And it is widely known that solvers contain bugs and sometimes return incorrect results. Because of their complexity, formal verification of their code using contract-based deductive verification is beyond the reach of deductive verification tools. Two main approaches are proposed to obtain trustworthy results. One of them consists in making the solver not only produce a result but also a certificate that is then checked by a checker. It is easier to formally verify a checker than the solver itself. This approach is used for SAT and SMT solvers, e.g. in [7, 1]. Regarding constraint programming (CP), the only work we know is [19] which proposes a proof-producing solver. A second approach consists in developing correct-by-construction solvers using proof assistants like PVS, Isabelle/HOL or Coq. Solvers are then specified, implemented and proved correct with respect to their specifications. The benefit of this approach w.r.t the former is that correctness proofs are done once and for all, and nothing has to be dynamically checked. It is usually difficult to design verified provers that include state-of-the-art optimisations and performance. But in critical applications, correctness is often more important than full performance. Such solvers can also serve as verified reference implementations helping for example to test modern tools when optimisations and heuristics are introduced.

This paper tackles the formal verification of solvers for finite domains using this last approach. In 2012, Carlier, Dubois and Gotlieb developed, a solver for finite domains - CP(FD) - formally verified with the help of the Coq proof assistant [5]. For convenience, from now on, let us call it CoqbinFD. However this solver is basic and can only deal with binary constraints. To improve its applicability, we propose, in this paper, to complement this solver with

a pre-processing step implementing decomposition of non-binary constraints into equivalent binary constraints. Our proposition focuses on ternary constraints and implements the Hidden Variable Encoding (HVE) [15] well-known in the CP community. This restriction is justified by the fact that any non-binary constraint can be translated into a set of ternary and binary constraints as long as only binary and unary operators occur in the constraint. The work could be extended to n-ary constraints, this generalization is in progress. The ternary step is helpful to obtain the general version. Hence our contribution is a fully verified and executable CP(FD) constraint solver that integrates CoqbinFD as a black-box. Fully verified means here that the solver is proved sound (in particular, if the solver answers UNSAT then there is no solution) and complete (in particular, if there is a solution, then the solver finds a solution). Like CoqbinFD, our solver is generic w.r.t the types of variables, values and constraints.

The paper is organized as follows. Section 2 briefly presents the notion of constraint satisfaction problem (csp) and the Hidden Variable Encoding. Then Section 3 introduces the main characteristics of CoqbinFD. Section 4 describes the Coq formalisation of the Hidden Variable Encoding and highlights the proven properties. We conclude and discuss the generalization to n-ary csps in the last section.

2 Decomposition of Non-binary Constraints into Binary Constraints

A *Constraint Satisfaction Problem* (csp for short) or network of constraints [12] is a triple (X, D, C) where X is a set of variables, C is a set of constraints over X and D is a partial function that associates a finite domain $D(x)$ to each variable x in X. Constraints are relationships between variables, each taking a value in their respective domain: constraints restrict possible values that variables can take. As commonly, we assume that constraints are normalized, meaning that two distinct constraints cannot hold over exactly the same variables. The arity of a constraint is the number of its variables (assumed as distinct). A n-ary csp contains k-ary constraints with $k \leq n$. A solution is defined as a total assignment of the csp variables which satisfies all the constraints simultaneously.

Let us consider the following ternary csp (X, D, C) where $X = \{x_1, x_2, x_3, x_4, x_5, x_6\}$, $D(v) = \{0, 1\}$ for all v in X and $C = \{c_1 : x_1 + x_2 + x_6 = 1, c_2 : x_1 - x_3 + x_4 = 1, c_3 : x_4 + x_5 - x_6 \geq 1, c_4 : x_2 + x_5 - x_6 = 0, c_5 : x_1 \geq x_6\}$ inspired from [18]. One of its solution is defined as $\{x_1 \mapsto 1, x_2 \mapsto 0, x_3 \mapsto 1, x_4 \mapsto 1, x_5 \mapsto 0, x_6 \mapsto 0\}$.

Decomposition of non-binary constraints into equivalent binary constraints is a subject that has been widely discussed in the CP community and for quite a long time. A well-known transformation for csps with finite domains is the Hidden Variable Encoding (HVE) [15]. In HVE, every non-binary constraint is associated with a variable whose domain is the set of all possible tuples of the original constraint, i.e. the set of tuples (of values of involved variables in the constraint) that satisfy the constraint. Such a variable is called *dual variable* and written v_c if c denotes the constraint. Thus the variables of the equivalent binary csp are the variables of the original csp called *original variables* and the dual variables. The domains of the original variables remain identical to their domain in the original csp. Non-binary constraints do not appear in the binary encoding: they are replaced by *hidden constraints* between a dual variable and each of the original variables in the constraint represented by the dual variable. A hidden constraint enforces the condition that a value of the original variable must be the same as the value assigned to it by the tuple that is the value of the dual variable [2]. In the following we denote them informally as projections: $proj_1$, $proj_2$, ... A mathematical definition of this

transformation (called the *hidden transformation*) can be found in [2] (see Definition 7).

As an illustration, the binary csp resulting from the HVE transformation applied on the example presented previously has 10 variables: the 6 original ones and 4 dual variables v_{c_1}, v_{c_2}, v_{c_3} and v_{c_4}. Domains of original variables remain identical whereas domains of the dual variables are such that $D(v_{c_1}) = \{(0,0,1),(0,1,0),(1,0,0)\}$, $D(v_{c_2}) = \{(0,0,1),(1,0,0),(1,1,1)\}$, $D(v_{c_3}) = \{(0,1,0),(1,0,0),(1,1,0),(1,1,1)\}$ and $D(v_{c_4}) = \{(0,0,0),(0,1,1),(1,0,1)\}$. There are 13 binary constraints: c_5 and 12 hidden constraints, e.g. $proj_1(v_{c_2}, x_1)$, $proj_3(v_{c_4}, x_6)$.

3 Presentation of the Formally Verified Solver CoqbinFD

In this section we briefly describe the binary solver CoqbinFD that we want to reuse. For more details please consult [5]. An important point in this development and crucial for the present work is its genericity. In the following we mainly emphasize the requirements upon the generic parameters. The solver is indeed parameterized by the type of variables (`variable`) and values (`value`) and also by the constraint language (`constraint`). In Coq, these types are abstract, assumed to accept a decidable equality. It is also assumed that the semantics of the constraints is given by an interpretation function as a Boolean function of the values of its two variables and a function that retrieves, for any constraint, its two variables. So a constraint is abstracted as a relation over two distinct variables, represented by an interpretation predicate. These types and functions must be defined either in Coq or directly in OCaml in order to use the extracted solver in a particular context. Here they are given Coq concrete values according to HVE.

A csp is defined as a record of type `network_csp` consisting of a finite list of variables (`CVars` field), a finite list of constraints (`Csts`) and a map (`Doms`) associating each domain with its variable, here a finite list of values (in the following we use the usual dot notation to get the content of a record field). A predicate (`network_inv`) specifies the well-formedness of a csp: the entries of the domain map are exactly the variables of the csp, variables appearing in the constraints are exactly those declared, constraints are normalized and finally the two variables of any constraint are distinct. These requirements are generally implicit in the CP literature, they must be explicit when it comes to formal proof. Numerous theorems in the development take the csp well-formedness as an hypothesis. A solution is implemented as a map from variables to values.

4 Coq Formalisation

Hidden Variable Encoding

As in CoqbinFD, types of variables (`variable3`) and values (`value3`) are abstract. Constraints (see below the definition of the type `constraint3`), either binary or ternary, are also abstract but the arity of a constraint is made explicit. It means that the type of basic constraints (`basic_constraint`) is abstract, equipped with a function to get the variables and an abstract interpretation function as in CoqbinFD. A ternary constraint is defined by its 3 variables and a value of the abstract type `OP`. Again an interpretation Boolean function is expected for each `OP` value. Constraint c_1 of the example given in Section 2 is represented as `Ter3 p1 x1 x2 x6` where `p1` is associated to the interpretation function $f(a,b,c) := a + b + c - 1 = 0$.

```
Inductive constraint3 : Set :=
| Basic3 : basic_constraint → constraint3
| Ter3 : OP → variable3 → variable3 → variable3 → constraint3.
```

The type definition (`network_csp3`) for the input csp - that we call here *non-binary csp* even if it contains no ternary constraint - is a copy of the one for the binary case. The well-formedness predicate (`network_inv3`) for non-binary csps is close to its counterpart in CoqbinFD informally presented in the previous section.

To solve a non-binary csp, we propose to transform it into a binary csp according to HVE, then call CoqbinFD to obtain an answer (UNSAT or a solution) and translates back the solution, if any, into a solution of the original csp. Since CoqbinFD is generic, we have to instantiate its types of variables, values and constraints according to HVE. The type of variables, `variable`, is defined inductively and reflects that variables are either original variables (introduced by the constructor `OVar`) or hidden variables (constructor `HVar`). The latter variables are defined w.r.t an original constraint. Thanks to the genericity of CoqbinFD, we can make explicit this association. For example, the hidden variable v_{c_1} of the example given in Section 2 is encoded in Coq as `HVar p1 x1 x2 x6`.

```
Inductive variable :=
| OVar : variable3 → variable
| HVar : OP → variable3 → variable3 → variable3 → variable.
```

The type of values, `value`, is also defined inductively, it distinguishes raw values, which are the original variables values from tuples which are the hidden variables values.

```
Inductive value :=
| Raw_value : value3 → value
| Triple : value3 → value3 → value3 → value.
```

A decidable equality and a strict order are defined for both types, following from the required equalities and orders on `value3` and `variable3`.

We can now define the type `constraint` whose values are the original binary constraints and the hidden constraints. In our example, the hidden constraint between v_{c_1} and the second original variable is represented in Coq by `Triplesnd p1 x1 x2 x6 x2`. We prove the properties on the constraint language required by CoqbinFD, e.g. any constraint has distinct variables.

```
Inductive constraint : Set :=
| Basic : basic_constraint → constraint
| Triplefst : OP → variable3 → variable3 → variable3 → variable3 → constraint
| Triplesnd : OP → variable3 → variable3 → variable3 → variable3 → constraint
| Triplethd : OP → variable3 → variable3 → variable3 → variable3 → constraint.
```

The Coq function, `translate_csp3`, that translates a non-binary csp into a binary csp, closely follows the presentation in Section 2. It uses several intermediate functions, in particular the function `expand` that computes the domain of a hidden variable, as a list of triples, from the interpretation function and the domains of the ordinary variables of the constraint corresponding to the hidden variable. The computed domain contains only the triples that satisfy the interpretation. It also uses the function `csts3Tocsts2` which computes, for a list of constraints, the list of original binary and hidden constraints and the list of hidden variables coupled with their list of triples computed with the help of `expand`. The ordinary binary constraints of the original csp and the corresponding domains are just copied modulo some mappings. The map containing the domains of the hidden variables is built with the help of the function `new_domain`.

```
Definition translate_csp3 csp3 := match csts3Tocsts2 csp3.Csts3 csp3.Doms3 with
| None ⇒ None
| Some (cs, lvdv) ⇒ Some (Make_csp
```

```
      (List.map (fun x ⇒ OVar x) (csp3.CVars3) ++ List.map first lvdv)
      (new_domain (map3_to_raw (csp3.Doms3) (csp3.CVars3)) lvdv)cs)
end.
```

Note that `translate_csp3` may fail when `csts3Tocsts2` tries to access the domain of unknown variables. We prove that if the non-binary csp is well-formed then the translation does not fail:

```
Lemma network_inv3_translate_None_False : ∀ csp3,
network_inv3 csp3 → ¬ (translate_csp3 csp3 = None).
```

We also prove that the binary csp obtained by HVE is well-formed if the original csp is well-formed:

```
Lemma translate_csp3_network_inv : ∀ csp3 csp,
network3_inv csp3 → translate_csp3 csp3 = Some csp → network_inv csp.
```

Definition of the Solver and Soundness

We can now define the solving function, `solve_csp3`, for a non-binary csp, we simultaneously prove its soundness. Like its counterpart `solve_csp` in CoqbinFD, this function takes as input not only a non-binary csp but also a proof of its well-formedness and returns either `None` and a proof that there is no solution or `Some a` and a proof that `a` is a solution of the original non-binary csp. In the former case, `None` is returned because the solving function in CoqbinFD, `solve_csp`, detects that the binary csp obtained by HVE has no solution. In the latter case, `a` is obtained by transforming the solution obtained for the binary encoding thanks to the function `translate_sol` which throws out of the latter solution the hidden variables part. The code of function `solve_csp3` is written using the `Program` facility. Its code is very close to the OCaml code except the `_` part which is a proof of the well-formedness of the binary encoding.

```
Definition result3_ok o_d csp3 :=
(o_d = None → ∀ a3, ¬ solution3 a3 csp3) ∧ (∀ d', o_d = Some d' → solution3 d' csp3).

Program Definition solve_csp3 (csp3 : network3) (Hinv3: network3_inv csp3) :
{ret3 : option (D3.t value3)|result3_ok ret3 csp3} :=
match (translate_csp3 csp3) with
| None ⇒ None
| Some csp ⇒ match (solve_csp csp _) with
             | None ⇒ None
             | Some a ⇒ Some (translate_sol a (CVars3 csp3))
             end
end.
```

Four proof obligations (po) are generated (Coq statements are omitted here):

- the first po corresponds to the case where the HVE translation fails. The proof follows, by contradiction, from the previous lemma `network_inv3_translate_None_False`;
- the second po states that the computed binary csp is well-formed, it is done by applying the previous lemma `translate_csp3_network_inv`;
- the third po requires to prove that if CoqbinFD returns `None` then the original csp has no solution, it follows from the soundness of CoqbinFD and a lemma stating that if the HVE computed binary csp admits no solution then the original csp has also no solution;
- the last po requires to prove that if CoqbinFD returns a solution then its translation is a solution of the original csp, it follows from the soundness of CoqbinFD and lemma

`translate_sound` (see below) expressing that the translation of a solution of the binary computed csp is a solution of the original csp.

```
Lemma translate_sound : ∀ a csp3 csp,
network3_inv csp3 → translate_csp3 csp3 = Some csp → solution a csp →
  solution3 (translate_sol a (CVars3 csp3)) csp3.
```

Completeness

Completeness of `solve_csp3` follows from completeness of CoqbinFD and from two more properties about `translate_csp3` regarding solutions. For example, lemma `translate_complete` explains that if the original non-binary csp admits a solution, then its mapping to the dual and original variables (computed by `translate_sol3`) is a solution of the binary encoding.

```
Lemma translate_complete: ∀ a3 csp3 csp,
network3_inv csp3 → translate_csp3 csp3 = Some csp →
  solution3 a3 csp3 → solution (translate_sol3 a3 csp3) csp.
```

Proofs related to `translate_csp` are often repetitive, they usually use an induction on binary constraints and subproofs for `Triplefst`, `Triplesnd` and `Triplethd`, are very similar with just a change in the involved original variable.

Completeness of `solve_csp3` is stated with the two following theorems. The first one proves that if the non-binary csp has a solution, then `solve_csp3` returns a solution; the second one states that if the csp has no solution then `solve_csp3` returns `None`, meaning UNSAT here.

```
Theorem solve3_complete_sol : ∀ csp3 (Hinv3 : network3_inv csp3) a3,
(solution3 a3 csp3) → ∃ a3, solve_csp3 csp3 Hinv3 = Some a3 ∧ solution3 a3 csp3.

Theorem solve3_complete_unsat : ∀ csp3 (Hinv3 : network3_inv csp3),
(∀ a3, ¬ (solution3 a3 csp3)) → solve_csp3 csp3 Hinv3 = None.
```

The Coq development (excluding CoqbinFD code) is around 3000 lines of code (loc), distributed in 70 loc for generic parameters, 300 loc for the instanciation of the CoqbinFD generic parameters and 2800 loc for the definition of the HVE translation and the solver *solve_csp3* and proofs. These lines contain 60 definitions of functions among which 33 are only used in the proofs, 12 inductive definitions and around 110 lemmas and theorems. The code is available at www.ensiie.fr/~dubois/HVE.

4.1 Experimentations with the Extracted Solver

After extraction, we ran the new solver (with the AC3 instance of CoqbinFD) to solve some problems. First we have used it with binary csps, for non-regression testing. The time overhead is not significant. Then we used it for problems with reasonable numbers of variables and constraints. We solved a classical mathematical problem, Golomb rulers instances. A Golomb ruler is a set of p marks at integer positions along an imaginary ruler of length n such that no two pairs of marks are the same distance apart. The constraint `golomb n p` is satisfied if there exists a ruler of length `n` with `p` marks such as defined below. For solving `golomb 25 7`, it takes around 23 seconds on a laptop (1,7 GHz Intel Core i7 8 Go 1600 MHz DDR3) and for certifying that `golomb 24 7` has no solution, it takes around 100 seconds. The Golomb instances were already solved by CoqbinFD but here the decomposition is done by the formally verified solver whereas it was done manually and in an untrusted manner in [5]. We also solved some problems from XCSP2.1 library. For example, for the problem known as `normalized-graceful-K2-P3` with 15 variables and 60 constraints, 9 of them being ternary), we obtain a solution in 0.5 sec.

5 Related Work

We focus here on work using theorem proving for constraint solvers and constraint models.

As far as we know, CoqbinFD is the first formally verified constraint solver for finite domains. The work presented here extends it in order to take into account ternary constraints. However, more broadly, some formalisations of verification tools with proof assistants exist. In [11], a reflexive DPLL algorithm is formalized in Coq, resulting in a new Coq tactic for deciding SAT formulas. In [16], Shankar and Vaucher formalize in PVS a SAT solver including clause learning and back-jumping. In Isabelle/HOL, in [13], Maric also takes into account the two-literal watching technique. Fleury et al. go beyond using Isabelle/HOL refinement and propose a CDCL-based SAT solver using more efficient imperative data structures [9]. In [17], the authors propose an Isabelle/HOL formalisation and total correctness proof for the incremental version of the Simplex algorithm used in many SMT solvers. In [8] the authors propose a verified LTL model checker developed in Isabelle/HOL as a reference implementation. In [14], Neumann extends this model checker by giving support for Promela models.

The use of theorem proving for constraint models is considered in [3, 4]. In [3], the authors show that general properties, such as model equivalence, although undecidable in general, can be automatically verified when a restricted language is considered. In [4], theorem proving is used to derive symmetry breaking constraints. In [10], Coq is used to derive and prove properties on a constraint model about a communication protocol. In [6], programs are turned into constraints in order to derive test cases. The equivalence between the solutions of the constraint system and the evaluation of the corresponding program is proved in Coq.

6 Conclusion

In this paper we complement the solver written and proved correct by Carlier *et al.* with a preprocessing step allowing for decomposition of ternary csps into binary csps. Thus the whole solver has a larger applicability.

Two independent extensions can be considered. The first one consists in formalizing and proving the encoding of a n-ary constraint into a set of ternary constraints. The second one, in progress, is to generalize HVE to n-ary csps. Thus a n-ary constraint is encoded using n binary constraints. In the Coq development some functions have to be re-defined and some proofs re-visited. However the proof roadmap remains the same. A solution consists in using lists instead of triples but performance of the extracted solver will be poor. Another solution is to use abstract tuples refined, at extraction time, to arrays.

Acknowledgements We thank the anonymous referees for their constructive and helpful comments.

References

[1] M. Armand, G. Faure, B. Grégoire, Chantal Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In J. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011.

[2] F. Bacchus, X. Chen, P. van Beek, and T. Walsh. Binary vs. non-binary constraints. *Artificial Intelligence*, 140(1):1 – 37, 2002.

[3] C. Bessiere, E. Hebrard, G. Katsirelos, Z. Kiziltan, N. Narodytska, and T. Walsh. Reasoning about Constraint Models. In D. N. Pham and S. Park, editors, *PRICAI 2014: Trends in Artificial Intelligence - 13th Pacific Rim Int. Conference on Artificial Intelligence, Gold Coast, QLD, Australia*, volume 8862 of *LNCS*, pages 795–808. Springer, 2014.

[4] M. Cadoli and T. Mancini. Using a Theorem Prover for Reasoning on Constraint Problems. *Applied Artificial Intelligence*, 21(4&5):383–404, 2007.

[5] M. Carlier, C. Dubois, and A. Gotlieb. A Certified Constraint Solver over Finite Domains. In *Formal Methods (FM)*, volume 7436 of *LNCS*, pages 116–131, Paris, 2012.

[6] M. Carlier, C. Dubois, and A. Gotlieb. A First Step in the Design of a Formally Verified Constraint-Based Testing Tool: Focaltest. In A. D. Brucker and J. Julliand, editors, *Tests and Proofs - TAP 2012, Prague, Czech Republic*, volume 7305 of *LNCS*, pages 35–50. Springer, 2012.

[7] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt, M. Kaufmann, and P. Schneider-Kamp. Efficient Certified RAT Verification. In L. de Moura, editor, *Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden*, volume 10395 of *LNCS*, pages 220–236. Springer, 2017.

[8] J. Esparza, P. Lammich, R. Neumann, T. Nipkow, A. Schimpf, and J. Smaus. A Fully Verified Executable LTL Model Checker. In N. Sharygina and H. Veith, editors, *Computer Aided Verification 2013, Saint Petersburg, Russia*, volume 8044 of *LNCS*, pages 463–478. Springer, 2013.

[9] M. Fleury, J. C. Blanchette, and P. Lammich. A verified SAT solver with watched literals using imperative HOL. In J. Andronick and A. P. Felty, editors, *7th ACM SIGPLAN Int. Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA*, pages 158–171. ACM, 2018.

[10] O. Grinchtein, M. Carlsson, C. Dubois, and J. Pearson. Exploring Properties of a Telecommunication Protocol with Message Delay using an Interactive Theorem Prover. In E. B. Johnsen and I. Schaefer, editors, *SEFM 2018, Toulouse, France*, volume 10886 of *LNCS*, pages 239–253. Springer.

[11] S. Lescuyer and S. Conchon. Improving Coq Propositional Reasoning Using a Lazy CNF Conversion Scheme. In *FroCos*, volume 5749 of *LNCS*, pages 287–303. Springer, 2009.

[12] A. Mackworth. Consistency in Networks of Relations. *Art. Intel.*, 8(1):99–118, 1977.

[13] F. Maric. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411(50):4333–4356, 2010.

[14] R. Neumann. Using Promela in a Fully Verified executable LTL Model Checker. In D. Giannakopoulou and D. Kroening, editors, *Verified Software: Theories, Tools and Experiments, VSTTE 2014, Vienna, Austria*, volume 8471 of *LNCS*, pages 105–114. Springer, 2014.

[15] F. Rossi, C. J. Petrie, and V. Dhar. On the Equivalence of Constraint Satisfaction Problems. In *ECAI*, pages 550–556, 1990.

[16] N. Shankar and M. Vaucher. The Mechanical Verification of a DPLL-based Satisfiability Solver. *Electr. Notes Theor. Comput. Sci.*, 269:3–17, 2011.

[17] M. Spasic and F. Maric. Formalization of Incremental Simplex Algorithm by Stepwise Refinement. In D. Giannakopoulou and D. Méry, editors, *FM 2012: Formal Methods - 18th Int. Symposium, Paris, France, 2012*, volume 7436 of *LNCS*, pages 434–449. Springer, 2012.

[18] K. Stergiou and T. Walsh. Encodings of Non-Binary Constraint Satisfaction Problems. In J. Hendler and D. Subramanian, editors, *Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, 1999, Orlando, Florida, USA.*, pages 163–168. AAAI Press / The MIT Press, 1999.

[19] M. Veksler and O. Strichman. A Proof-producing CSP Solver. In M. Fox and D. Poole, editors, *Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 2010.* AAAI Press, 2010.

En Finir avec les Faux Positifs grâce à l'Exécution Symbolique Robuste

Benjamin Farinier[1], Sébastien Bardin[2], Richard Bonichon[2], and Marie-Laure Potet[3]

[1] LRI, Université Paris-Saclay, France
prenom.nom@lri.fr
[2] CEA LIST, Université Paris-Saclay, France
prenom.nom@cea.fr
[3] Univ. Grenoble Alpes, Verimag, France
prenom.nom@univ-grenoble-alpes.fr

Résumé

L'exécution symbolique est une technique de vérification formelle par sous-approximation ayant fait ses preuves en recherche de bogues, notamment grâce à son absence de faux positifs : un bogue trouvé est un bogue réel. Cependant, si cette propriété est vraie dans le cas où l'utilisateur contrôle toutes les entrées du programme, les choses se compliquent quand certaines entrées sont hors de son contrôle, typiquement l'environnement. L'exécution symbolique devient alors *fragile* dans le sens où elle peut produire des faux positifs. Ce cas se rencontre particulièrement en recherche de vulnérabilités, où les failles cherchées doivent être reproductibles. Dans cet article nous montrons comment l'utilisation de quantificateurs permet de passer du problème de l'accessibilité à celui de l'*accessibilité robuste* et proposons une modélisation cohérente en tant que sous-approximation. Ces quantificateurs sont ensuite éliminés et les formules obtenues simplifiées afin de limiter au maximum l'impact sur le temps de résolution. Il en résulte ainsi une analyse par exécution symbolique efficace et *robuste* vis-à-vis de son environnement, réellement exempte de faux positifs.

1 Introduction

Contexte. La vérification de programmes est un succès indéniable pour les logiciels critiques. De nombreux efforts poussent depuis plusieurs années à l'adapter à des logiciels non critiques. Notamment, *l'exécution symbolique* [4] vise à explorer une partie des chemins d'un programme afin de trouver des bogues. Chaque nouveau chemin est accompagné d'une entrée obtenue par raisonnement symbolique et Satisfiabilité Modulo Théorie (SMT) [2] permettant de l'activer. Cette méthode rencontre un énorme succès depuis quelques années [10] pour sa capacité à passer sur des codes complexes et son *absence de faux positifs* : un bogue rapporté est un bogue certain. L'exécution symbolique fait ainsi partie des techniques de vérification par *sous-approximation.*

Problème. Cependant, en pratique *les faux positifs existent* [3, 5]. Ils proviennent d'abstractions de certains morceaux de code absents ou trop compliqués à gérer, par exemple lors d'une analyse niveau C d'un code contenant de l'assembleur, d'une modélisation de l'état initial ou de l'environnement imparfaite, par exemple en analyse de code binaire. Ce problème est connu dans la communauté de l'exécution symbolique, mais pas véritablement étudié ni quantifié. Par ailleurs, il est plutôt vu comme une fatalité, pas forcément gênante si la méthode trouve suffisamment de bogues par ailleurs. C'est cependant un problème sérieux pour l'utilisateur, qui ne peut plus faire confiance aux taux de couverture calculés (test) ou doit passer du temps à analyser un bogue retourné non confirmé par l'entrée associée (sécurité). Le problème est

particulièrement saillant au niveau de l'analyse de code binaire, car les entrées initiales et les interactions avec l'environnement ne sont pas aisées à établir a priori.

En pratique, des solutions ad hoc peuvent diminuer le nombre de faux positifs, mais elles n'en garantissent pas l'absence et parfois en introduisent de nouveaux ou surchargent complètement le solveur automatique.

Proposition. Nous visons à développer une *exécution symbolique robuste*, c'est-à-dire réellement sans aucun faux positif. Pour cela nous proposons un nouveau cadre, *l'accessibilité robuste*, dans lequel les entrées sont partitionnées en *contrôlées* (par l'utilisateur) et *non contrôlées* (environnement). Une solution est dite robuste si elle atteint le but voulu indépendamment des valeurs non contrôlées. Ce cadre permet d'encoder les cas gênants évoqués plus haut. Cet article présente nos premiers résultats sur l'exécution symbolique robuste.

— Nous définissons formellement le cadre d'accessibilité robuste et montrons intuitivement son intérêt pratique ;

— Nous proposons une révision de l'exécution symbolique, dite exécution symbolique robuste, destinée à résoudre le problème de l'accessibilité robuste et à éliminer les faux positifs ;

— Finalement, nous décrivons une première implémentation dans l'outil d'analyse de code binaire Binsec [6, 7] et rapportons de premiers résultats expérimentaux encourageants.

2 Avant-propos sur l'exécution symbolique

L'exécution symbolique [4] est une approche formelle au test automatisé de logiciel, fondée sur la sémantique du code du programme. Cette approche repose sur l'idée suivante : étant donné un chemin dans un programme, il est possible de calculer une formule logique sur les *entrées du programme* [1], appelée *prédicat de chemin*, dont une solution est une entrée de test exerçant ledit chemin. Bien que cela soit souhaitable, il n'est pas nécessaire que toutes les entrées exerçant le chemin soient solutions du prédicat puisque la technique est par définition une sous-approximation — seule une partie des chemins est explorée.

```
int main () {
  int x = input();
  int y = input();
  int z = 2 * y;
  if (z == x) {
    if (x > y + 10)
      printf("Success!\n");
  }
  printf("Failure...\n");
}
```

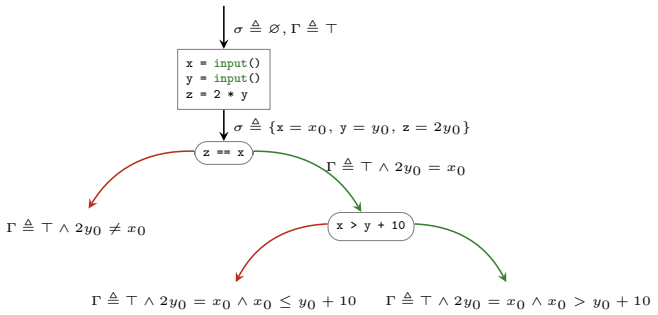


Figure 1 – Exécution symbolique sur un petit programme

Sur l'exemple en Fig. 1, l'exécution symbolique va énumérer les trois chemins possibles de ce petit programme. Chacun d'entre eux correspond à un prédicat de chemin Γ différent. Celui-ci est construit par la conjonction des contraintes générées aux points de branchement du

1. Au sens large : ligne de commande, arguments, fichiers, environnement, etc.

programme, en fonction des valeurs logiques des variables dans l'environnement σ. La faisabilité du chemin est ici équivalente à la satisfiabilité de la formule qui le représente.

Nous utilisons ensuite un solveur automatique, typiquement un solveur SMT. Lorsque ces solveurs répondent positivement à la question de la satisfiabilité, les modèles de la formule générée font voir des jeux d'entrées exerçant le chemin exécuté symboliquement. Ainsi, le modèle $\{x_0 = 0,\ y_0 = 1\}$ est une solution (parmi d'autres) au chemin dont la contrainte est $\top \wedge 2y_0 \neq x_0$. Notons que les formules générées sont naturellement *non quantifiées* et tombent dans le *fragment décidables* des théories employées — ici vecteurs de bits (variables) et tableaux (mémoire) — et que c'est la *génération de modèles* plutôt que la simple question de la satisfiabilité qui importe.

Une exécution symbolique est dite *complète* si elle couvre toutes les exécutions concrètes possibles. Une exécution symbolique est dite *correcte* si chaque solution générée pour chaque chemin parcouru est *correcte*, c'est-à-dire qu'elle suivra le chemin prévu lors de son exécution concrète. Cette définition esquisse en pointillés la notion de *faux positifs*, des solutions venant d'un prédicat de chemin incorrect et ne suivant pas le chemin prévu lors de l'exécution concrète.

3 Motivation

```c
#define SIZE

void get_secret (char secr[]) {
// Retrieve the secret
}

void read_input (char src[], char dst[]) {
  int i = 0;
  while (src[i]) {
    dst[i] = src[i];
    i++;
  }
}

int validate (char secr[], char inpt[]) {
  int b = 1;
  for (int i = 0; i < SIZE; i++) {
    b &= secr[i] == inpt[i];
  }
  return b;
}

int main (int argc, char *argv[]) {
  char secr[SIZE];
  char inpt[SIZE];

  if (argc != 2) return 0;

  get_secret(secr);
  read_input(argv[1],inpt);

  if (validate(secr, inpt)) {
    printf("Success!\n");
  }
  else {
    printf("Failure...\n");
  }
}
```

FIGURE 2 – Programme pour lequel nous cherchons une entrée permettant d'atteindre « Success ! »

Prenons en guise d'exemple le programme donné en Fig. 2 comportant, en plus de la fonction `main` les fonctions `get_secret`, `read_input` et `validate`. Schématiquement, la fonction `get_secret`, dont le code n'est pas donné, récupère un secret à deviner, inconnu de l'utilisateur et imprévisible (fonction cryptographique, etc.), et l'écrit dans le tampon prévu à cet effet. La fonction `read_input` copie l'entrée utilisateur dans le tampon qui lui est consacré tandis que la fonction `validate` vérifie que l'entrée utilisateur est égale au secret à deviner. La fonction `main`, après avoir contrôlé que le bon nombre d'arguments a été donné, utilise ces fonctions pour vérifier si l'entrée correspond au secret, et affiche selon le cas « Success ! » ou « Failure... ».

Objectif. Notre but est de trouver une entrée permettant d'atteindre le branche « Success ! ». Nous montrons tout d'abord comment une utilisation directe de l'*exécution symbolique classique*

au niveau du code C conduit à une première solution non robuste, c'est-à-dire un *faux positif*. Nous montrons aussi comment notre méthode d'*exécution symbolique robuste* évite cet écueil (pas de faux positif), sans toutefois trouver de « vraie » solution au problème. Nous considérons ensuite la sémantique au niveau binaire du code associé. L'exécution symbolique classique conduit toujours à un faux positif, mais cette fois l'*exécution symbolique robuste* trouve bien une « vraie » solution, en passant par un dépassement de tampon.

Premier essai : sémantique haut niveau. Essayons donc par exécution symbolique classique de trouver une entrée qui nous permette d'atteindre la branche « Success ! ». Puisque son contenu provient d'une entrée utilisateur, le tableau de caractères `impt` va être modélisé par une première variable symbolique i. De même le tableau de caractères `secr` va être modélisé par une seconde variable symbolique s, car son contenu provient de l'exécution d'un code ne faisant pas partie de l'analyse (appel à une bibliothèque cryptographique externe, etc). Finalement, la contrainte à satisfaire pour valider le problème va être $i_{[0,\,\texttt{SIZE}-1]} = s_{[0,\,\texttt{SIZE}-1]}$, où $i_{[0,\,\texttt{SIZE}-1]}$ (resp. $s_{[0,\,\texttt{SIZE}-1]}$) dénote les valeurs de i (resp. de s) aux indices allant de 0 à $\texttt{SIZE}-1$. Mais il s'avère que cette contrainte a pour solution triviale $\left\{i_{[0,\,\texttt{SIZE}-1]} = 0,\ s_{[0,\,\texttt{SIZE}-1]} = 0\right\}$, qui ne sera que très improbablement une solution réelle à notre problème.

Ce *faux-positif* vient du fait que les outils d'exécution symbolique considèrent toutes les entrées du programme de la même manière, c'est-à-dire *contrôlées par l'utilisateur*, ce qui donne ici le prédicat de chemin $\exists i.\exists s.i_{[0,\,\texttt{SIZE}-1]} = s_{[0,\,\texttt{SIZE}-1]}$ — nous ajoutons les quantifications existentielles sur i et s pour bien faire ressortir les entrées ; en pratique ce sont juste des variables libres puisque nous effectuons une requête de satisfiabilité. Or ici, puisque s est hors du contrôle de l'utilisateur, une modélisation appropriée du problème est $\exists i.\forall s.i_{[0,\,\texttt{SIZE}-1]} = s_{[0,\,\texttt{SIZE}-1]}$, qui nécessite des quantificateurs universels. Cette contrainte n'est cette fois-ci pas satisfiable, mais, à défaut de solutions, il n'y a plus de faux positifs.

Second essai : sémantique bas niveau. Si l'on souhaite trouver une solution à notre problème, il nous faut aller plus en détail dans le fonctionnement de l'exécution symbolique et dans la modélisation de notre programme. On remarque premièrement que la fonction `read_input` lit l'entrée utilisateur jusqu'à rencontrer le caractère nul de fin de chaîne. Du côté de l'exécution symbolique, cela se matérialise par l'introduction d'une constante N implicitement définie par le nombre d'itérations de la boucle de lecture et représentant la longueur de l'entrée utilisateur. Deuxièmement, remarquons que les variables locales `inpt` et `secr` sont des tableaux de caractères de taille fixe qui sont alloués directement dans la pile, consécutivement. Du côté de la modélisation, cela se matérialise par l'introduction de deux variables symboliques supplémentaires, m_0 représentant l'état mémoire du programme à un instant donné, et p_0 un pointeur vers la pile du programme. Les variables s, m_0 et p_0 sont universellement quantifiées car l'utilisateur ne peut directement fixer leur valeur de sorte à résoudre le problème. En notant $\triangleq$ la définition de variables intermédiaires, la contrainte que l'on cherche à satisfaire devient :

$$
\begin{aligned}
&\exists i.\,\forall s.\,\forall m_0.\,\forall p_0.\\
&\quad p_1 \triangleq p_0 - \texttt{SIZE}\\
&\quad p_2 \triangleq p_1 - \texttt{SIZE}\\
&\quad m_1 \triangleq m_0\,[p_1,\, p_1 + \texttt{SIZE} - 1] \leftarrow s\\
&\quad m_2 \triangleq m_1\,[p_2,\, p_2 + N - 1] \leftarrow i\\
&m_2\,[p_1,\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2,\, p_2 + \texttt{SIZE} - 1]
\end{aligned}
$$

Encore une fois, cette contrainte comporte des quantificateurs, en particulier universels, qui augmentent significativement la difficulté à trouver une solution pour les solveurs. On

cherche donc à les éliminer en utilisant une approche par teinte [8]. Sur notre exemple, on peut ainsi remplacer les quantifications universelles par des existentielles sur s et m_0 si on ajoute la contrainte $[p_1,\, p_1 + \texttt{SIZE} - 1] \subseteq [p_2,\, p_2 + N - 1]$ équivalente à $N \geq 2 \cdot \texttt{SIZE}$, soit donc :

$$
\begin{aligned}
&\exists i.\, \exists s.\, \exists m_0.\, \forall p_0.\\
&\quad p_1 \triangleq p_0 - \texttt{SIZE}\\
&\quad p_2 \triangleq p_1 - \texttt{SIZE}\\
&\quad m_1 \triangleq m_0\,[p_1,\, p_1 + \texttt{SIZE} - 1] \leftarrow s\\
&\quad m_2 \triangleq m_1\,[p_2,\, p_2 + N - 1] \leftarrow i\\
&m_2\,[p_1,\, p_1 + \texttt{SIZE} - 1] = m_2\,[p_2,\, p_2 + \texttt{SIZE} - 1]\\
&\wedge\ N \geq 2 \cdot \texttt{SIZE}
\end{aligned}
$$

On simplifie ensuite cette contrainte en réduisant les *read-over-write* (lectures suivant une écriture) [9]. On obtient la contrainte $\exists i.\ i_{[0,\,\texttt{SIZE}-1]} = i_{[\texttt{SIZE},\,2\cdot\texttt{SIZE}-1]} \wedge N \geq 2 \cdot \texttt{SIZE}$ ne faisant plus référence qu'à la variable symbolique i existentiellement quantifiée. Ce dernier quantificateur pouvant simplement être enlevé (requête de satisfiabilité), nous devons donc trouver un modèle pour la formule désormais sans quantificateur $i_{[0,\,\texttt{SIZE}-1]} = i_{[\texttt{SIZE},\,2\cdot\texttt{SIZE}-1]} \wedge N \geq 2 \cdot \texttt{SIZE}$.

Pour par exemple $\texttt{SIZE} = 8$, une solution possible à cette formule est `abcdefghabcdefgh`, qui permet effectivement d'attendre la branche « Success ! » par dépassement de tampon dans la fonction `read_input`.

4 Exécution symbolique robuste

Accessibilité robuste. Nous rappelons tout d'abord la notion classique d'accessibilité. Pour un programme P sur des entrées (a, x), nous disons qu'un objectif (l, φ) — où l est une localisation dans le code et φ un prédicat local sur l'état du programme à la localisation l, est *accessible* s'il existe un couple d'entrées (a_0, x_0) tel que l'exécution de P sur (a_0, x_0) atteint effectivement la localisation l dans un état qui satisfait φ. Nous noterons $P(a_0, x_0) \rightsquigarrow (l, \varphi)$.

Nous définissons maintenant la notion d'*accessibilité robuste*. Pour un programme P sur des entrées (a, x) où a est contrôlée et x ne l'est pas, nous disons qu'un objectif (l, φ) est *accessible de manière robuste* s'il existe une entrée a_0 telle que pour toute entrée x_0 on ait $P(a_0, x_0) \rightsquigarrow (l, \varphi)$. Une telle donnée d'entrée utilisateur a_0 permet donc d'atteindre l'objectif quelle que soit la donnée de l'environnement.

Exécution symbolique robuste. Nous adaptons maintenant le principe de l'exécution symbolique au contexte de l'accessibilité robuste.

Un *prédicat de chemin robuste* doit désormais assurer l'accessibilité robuste, et non plus seulement l'accessibilité, du chemin visé. Il est en définitive aisé de calculer un tel prédicat robuste : il suffit d'utiliser le calcul symbolique habituel, puis de quantifier universellement les variables non contrôlées. On obtient ainsi un prédicat quantifié de la forme $\exists a.\forall x.\Gamma(a, x)$. Cette étape demande que l'utilisateur ait spécifié au préalable les entrées contrôlées et non contrôlées. Nous discutons ce point dans le prochain paragraphe.

Le problème est ensuite de pouvoir résoudre de tels prédicats quantifiés, car l'ajout de quantificateurs rend de nombreuses théories indécidables — c'est le cas par exemple de la théorie des tableaux. Heureusement, nous pouvons réutiliser au choix : certaines extensions récentes des solveurs SMT (par exemple, quantificateurs sur les vecteurs de bits), ou notre méthode d'élimination des quantificateurs de manière à maintenir une génération de modèles correcte [8].

Spécification des entrées utilisateurs. Par rapport à l'exécution symbolique classique, notre méthode demande donc à l'utilisateur de spécifier quelles entrées sont contrôlées ou non. Cet aspect est crucial : si l'utilisateur déclare comme contrôlées des entrées qui ne le sont pas dans le système réel, alors notre méthode retournera des faux positifs — pas par rapport au problème sous-jacent d'accessibilité robuste, mais par rapport au système réel.

Nous pouvons par contre remarquer que si l'utilisateur a déclaré « trop » d'entrées non contrôlées, alors notre technique ne peut pas produire de faux positifs — elle perdra juste en généricité. Une manière *robuste* de faire est donc de considérer que par défaut une entrée n'est contrôlée que si l'utilisateur l'a spécifié explicitement, là où l'exécution symbolique classique correspond exactement au choix inverse.

Ainsi dans notre implantation, en mode robuste, les entrées utilisateurs doivent être explicitement marquées comme contrôlées pour être modélisées par des variables existentielles. À titre d'exemple, le fichier de spécification pour l'analyse du programme en Fig. 2 est donné en Fig. 3.

```
controlled argc<32>;
@[esp<32> + 4<32>, 4] := argc<32>;

argv0_ptr := esp<32> + 0x00000100;          controlled argv0<160>;
argv1_ptr := esp<32> + 0x00000200;          controlled argv1<160>;

@[esp<32> +  8<32>, 4] := argv0_ptr;        @[argv0_ptr, 20] := argv0<160>;
@[esp<32> + 12<32>, 4] := argv1_ptr;        @[argv1_ptr, 20] := argv1<160>;
```

FIGURE 3 – Spécification des entrées contrôlées par l'utilisateur

Implémentation. Les différentes étapes nécessaires à l'exécution symbolique robuste sont implémentées pour la théorie des vecteurs de bits et tableaux dans TFML, le module de traitement des formules logiques de l'outil d'exécution symbolique BINSEC [6]. Côté élimination, nous utilisons la procédure générique d'élimination des quantificateurs par inférence de conditions d'indépendance présentée dans [8], en particulier son traitement de la théorie des tableaux : pour chaque tableau dans la formule un tableau fantôme est introduit permettant de suivre quelles cellules du tableau initial dépendent de variables universellement quantifiées. Cela évite l'introduction d'un grand nombre de if-then-else et permet de faire bénéficier aux contraintes introduites de nos simplifications sur les tableaux. Côté simplification donc, les règles de réécritures présentées dans [9] sont systématiquement appliquées à la construction des termes. De plus le traitement des *read-over-write* (théorie des tableaux) est appliqué une fois avant l'élimination des quantificateurs afin d'en faciliter les raisonnements sur les pointeurs, et une fois après pour réduire les contraintes introduites.

5 Évaluation expérimentale

On évalue l'impact de notre approche selon deux critères : 1) le nombre de vrais positifs, qui doit être dégradé le moins possible ; et 2) le nombre de faux positifs, qui doit être réduit à zéro. Pour ce faire, on considère un ensemble de challenges de type `crackme` dont le but est de trouver une entrée amenant le programme à dévoiler son secret. L'intérêt de ces programmes est qu'ils permettent de valider simplement une solution en rejouant le challenge avec. On compare notre exécution symbolique robuste avec élimination des quantificateurs et simplifications à

l'exécution symbolique classique avec simplification, et à une exécution symbolique robuste avec simplification mais sans élimination des quantificateurs.

Les expérimentations sont réalisées sur un processeur Intel Core i7-4712HQ @ 2.30GHz avec 16Go de RAM, et les résultats sont présentés dans la Table 1. L'approche classique trouve jusqu'à 12 solutions correctes pour ces challenges en fonction du solveur sous-jacent. Cependant, elle produit aussi de 9 à 12 faux positifs ne permettant pas de résoudre les challenges correspondant. L'approche robuste sans élimination des quantificateurs est, elle, bien exempte de faux positifs. Néanmoins le nombre de vrais positifs est fortement impacté, tombant à 7 pour le meilleur solveur. Enfin notre approche robuste avec élimination des quantificateurs permet de retrouver tous les vrais positifs de l'approche classique tout en restant exempt de faux positifs.

Table 1 – Comparaison du nombre de vrais et de faux positifs pour l'exécution symbolique classique, robuste sans élimination et robuste avec élimination des quantificateurs

	approche classique			approche robuste			approche robuste + élim.		
	SAT correct	SAT incorrect	UNSAT ou UNKNOWN	SAT correct	SAT incorrect	UNSAT ou UNKNOWN	SAT correct	SAT incorrect	UNSAT ou UNKNOWN
Boolector	**12**	11	1	N/A	**0**	24	**12**	**0**	12
CVC4	**7**	9	8	5	**0**	19	**7**	**0**	17
Yices	**7**	11	6	N/A	**0**	24	**7**	**0**	17
Z3	**12**	12	0	7	**0**	17	**12**	**0**	12

Application à l'analyse de vulnérabilités. Précisons tout d'abord que notre exécution symbolique robuste permet de trouver une solution correcte à l'exemple donné en Fig. 2. Nous l'avons aussi appliquée avec succès à des cas d'études réels issus de l'analyse de vulnérabilité, comme par exemple une version adaptée d'une faille de sécurité nommée « Back to 28 » [1]. Dans sa version originale, elle permettait en appuyant successivement 28 fois sur la touche retour arrière de contourner la procédure d'authentification du chargeur d'amorçage GRUB2. Dans la version de notre cas d'étude, le code à l'origine de cette faille est repris à l'identique à ceci près que la corruption d'adresse de retour est remplacée par une corruption des données en mémoire similaire à notre exemple. Ici encore, l'exécution symbolique classique retourne une solution fragile, tandis que les solveurs sous-jacents échouent à résoudre les formules produites par l'exécution symbolique robuste sans élimination des quantificateurs. Notre exécution symbolique robuste trouve une solution en 80 secondes avec élimination des quantificateurs, et même en 30 secondes en rajoutant les simplifications de tableau.

6 Autres approches, conclusion et travaux futurs

Autres approches. Comme dit auparavant, il n'existe pas d'approche systématique et correcte au problème de l'accessibilité robuste. Les outils d'exécution symbolique actuels se contentent de résoudre le problème de l'accessibilité classique (non robuste), et en pratique les utilisateurs essaient de pallier le problème des faux positifs de manière ad hoc.

Par exemple, pour le problème de l'état initial mal défini, particulièrement criant pour de l'analyse niveau binaire, on rencontre les mitigations suivantes :

— Initialisation de la mémoire à une valeur arbitraire, par exemple 0 : cela enlève certains faux positifs mais peut en ajouter d'autres, dans le cas où 0 établit la solution du prédicat de chemin sans pour autant correspondre à une réalité de l'OS / architecture sous-jacente.

— Initialisation de la mémoire à celle observée à l'exécution (concrétisation) : là encore on enlève certains faux positifs (une partie de ces valeurs est effectivement constante d'une exécution à l'autre), mais pas tous (certaines valeurs ne sont pas constantes d'une exécution à l'autre) ; par ailleurs la contrainte induite peut devenir trop imposante pour être gérée par les solveurs.

Conclusion et travaux futurs. Le cadre de l'exécution symbolique robuste permet de remédier à ce problème de manière définitive, à condition que : 1) les solveurs arrivent à gérer les formules quantifiées générées ; et 2) l'utilisateur ait bien spécifié les entrées contrôlées et non contrôlées. Nos travaux récents [8, 9] montrent comment une approche par pré-traitement permet de réutiliser les solveurs (*quantifier-free*) pour résoudre le point (1), et nous avons implémenté une preuve de concept dans l'outil Binsec.

Le point (2) est évidemment un problème puisqu'il repose sur l'utilisateur. Comme expliqué précédemment, notre technique ne peut pas produire de faux positifs quand « trop » d'entrées sont déclarées non contrôlées, et c'est pour cela que par défaut, nous considérons toutes les entrées non contrôlées. La tâche fastidieuse de spécifier explicitement quelles entrées sont contrôlées revient alors à l'utilisateur. Nos prochains travaux s'orientent justement vers l'aide à l'identification des entrées contrôlées.

Références

[1] http://hmarco.org/bugs/CVE-2015-8370-Grub2-authentication-bypass.html.

[2] C. Barrett and C. Tinelli. Satisfiability modulo theories. In E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, editors, *Handbook of Model Checking.*, pages 305–343. Springer, 2018.

[3] C. Cadar, D. Dunbar, and D. R. Engler. KLEE : Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224, 2008.

[4] C. Cadar and K. Sen. Symbolic execution for software testing : three decades later. *Commun. ACM*, 56(2) :82–90, 2013.

[5] V. Chipounov, V. Kuznetsov, and G. Candea. S2E : a platform for in-vivo multi-path analysis of software systems. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011*, pages 265–278, 2011.

[6] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. BINSEC/SE : A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. In *SANER, Suita, Osaka, Japan, March 14-18, 2016*, pages 653–656, 2016.

[7] A. Djoudi and S. Bardin. BINSEC : Binary Code Analysis with Low-Level Regions. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 212–217, 2015.

[8] B. Farinier, S. Bardin, R. Bonichon, and M.-L. Potet. Model generation for quantified formulas : A taint-based approach. In *30th International Conference on Computer Aided Verification (CAV 2018)*, 2018.

[9] B. Farinier, R. David, S. Bardin, and M. Lemerre. Arrays made simpler : An efficient, scalable and thorough preprocessing. In *22nd International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR 2018)*, 2018.

[10] P. Godefroid, N. Klarlund, and K. Sen. DART : directed automated random testing. In *PLDI, Chicago, IL, USA, June 12-15, 2005*, pages 213–223, 2005.

Un Mécanisme de Preuve par Réflexion pour Why3 et son Application aux Algorithmes de GMP

Raphaël Rieu-Helft[1,2]

[1] TrustInSoft, Paris F-75014
[2] Inria, Université Paris-Saclay, Palaiseau F-91120

Résumé

Nous présentons un mécanisme de preuves par réflexion pour la plateforme de vérification Why3. L'utilisateur peut facilement écrire des procédures de décision dédiées, formellement vérifiées et qui utilisent pleinement les fonctionnalités impératives du langage WhyML. L'impact sur la base de confiance est minimal.

Nous avons évalué cette approche sur des preuves d'algorithmes de GMP. Elle permet de supprimer un très grand nombre d'indications de coupure qui devaient auparavant être fournies manuellement à Why3.

1 Introduction

La plateforme de vérification Why3 [1] fournit un langage programmation proche de ML, WhyML, qui permet d'écrire des programmes et de spécifier leur comportement fonctionnel à l'aide de pré- et postconditions et d'invariants de boucle [7]. L'outil transforme les programmes et spécifications en théorèmes logiques qui peuvent être envoyés à des prouveurs externes, tant automatiques (solveurs SMT ou TPTP) qu'interactifs (Coq, Isabelle/HOL, PVS). Une fois ces théorèmes prouvés, en supposant la correction de Why3 et des prouveurs externes, on sait que les programmes satisfont leur spécification.

Dans de précédents travaux, nous avons utilisé Why3 pour implémenter des algorithmes issus de la bibliothèque d'arithmétique en précision arbitraire GMP [2], prouver leur correction, et générer une bibliothèque performante écrite en C [13]. Les preuves ont été faites en utilisant exclusivement des prouveurs automatiques. Cependant, non seulement certains algorithmes sont très complexes (par exemple la division [12]), mais nous avons aussi dû fournir nettement plus d'indications de coupure que prévu, y compris pour prouver des théorèmes apparemment triviaux. En effet, la taille des contextes de preuve et la présence d'arithmétique non-linéaire est problématique pour les solveurs. Pour certains théorèmes, nous avons dû écrire plusieurs assertions de plus de 100 lignes, ce qui limite grandement l'intérêt d'utiliser des outils automatiques plutôt qu'un assistant de preuve interactif.

Une façon d'ajouter à un prouveur de nouvelles fonctionnalités, comme par exemple une règle d'inférence adaptée à un certain problème, est d'"incorporer un principe de réflexion, afin que l'utilisateur puisse vérifier à l'intérieur du système de preuve que le code qui implémente une nouvelle règle est correct, et d'ajouter ce code au système" [9]. Nous avons ajouté à Why3 un mécanisme qui permet à l'utilisateur d'écrire des procédures de décision comme des programmes Why3 classiques et de les utiliser pour prouver des théorèmes par le calcul. Cet article montre comment nous avons utilisé notre approche pour implémenter et vérifier une procédure de décision adaptée à la vérification d'algorithmes de GMP.

Une transformation Why3 présentée en section 2.1 génère automatiquement les termes à passer en argument à la procédure de décision à partir de sa spécification et du but courant.

1. http://why3.lri.fr
2. http://gmplib.org

Traditionnellement, les preuves par réflexion se font en évaluant des termes purs présents dans des propositions logiques. Cependant, le langage de programmation fourni par Why3 est beaucoup plus riche : variables mutables, tableaux, exceptions, boucles... Notre mécanisme permet à l'auteur de procédures de décision de bénéficier de toute l'expressivité de WhyML. Nous avons dû ajouter à Why3 un interpréteur de code WhyML (section 2.2). Ceci étend la base de confiance de Why3, mais de manière minimale. Nous discutons la correction de notre approche en section 2.3.

Étant capables d'écrire des procédures de décision en WhyML, de les vérifier avec Why3 et de les exécuter sur des propositions logiques réifiées, nous avons tous les outils pour vérifier des algorithmes de GMP à l'aide d'une procédure de décision dédiée, que nous présentons en section 3. Il s'agit d'une procédure de résolution de systèmes d'équations linéaires, mais les coefficients qu'elle manipule sont des produits de rationnels par des exposants symboliques, comme par exemple $-5/3 \cdot \beta^{i+j-2}$. En effet, les algorithmes que nous vérifions manipulent des sommes de monômes de forme $\sum a_i \beta^i$.

Ce travail fait partie de Why3 et les exemples présentés sont disponibles à l'adresse `http://toccata.lri.fr/gallery/multiprecision.en.html`.

Cet article reprend du travail déjà publié à la conférence IJCAR 2018 [11].

2 Preuves par réflexion

Le principe général de la preuve par réflexion est le suivant. Notons P la proposition que l'on cherche à prouver. D'abord, on trouve un plongement de P dans le langage logique du système formel. Notons $\ulcorner P \urcorner$ le terme résultant, par exemple l'arbre de syntaxe abstraite de P. Ensuite, on prouve que si $\ulcorner P \urcorner$ satisfait une certaine propriété φ, alors P est vraie. Ainsi, lorsque l'on veut prouver une proposition P, il suffit de prouver $\varphi(\ulcorner P \urcorner)$. Si φ est telle que $\varphi(\ulcorner P \urcorner)$ peut être validée par le calcul, il s'agit d'une procédure de preuve par réflexion. Cette approche a été utilisée dans de nombreux contextes [9, 8, 1, 3, 4, 6].

2.1 Réification

Une difficulté est que la fonction $\ulcorner \urcorner$ n'est pas exprimable dans le langage logique, et que le terme $\ulcorner P \urcorner$ peut être d'une telle taille qu'on ne peut pas s'attendre à ce qu'un utilisateur le fournisse manuellement. Pour qu'un mécanisme de preuve par réflexion soit utile, il est important de fournir un moyen d'automatiser l'obtention de $\ulcorner P \urcorner$ à partir de P. On appelle ce processus *réification*. Une approche classique est d'exprimer $\ulcorner \urcorner$ dans le méta-langage du système formel utilisé (comme Ltac pour Coq), mais ceci demanderait à l'utilisateur d'apprendre le fonctionnement interne du système.

Notons cependant que si $\ulcorner \urcorner$ n'est pas exprimable dans le langage logique, son inverse l'est. De plus, l'utilisateur doit nécessairement définir quelque chose s'en approchant afin de donner une spécification utile à φ. Dans l'exemple en figure 1, il s'agit de la fonction `interp`. Nous utilisons cet inverse pour générer $\ulcorner P \urcorner$ automatiquement. La réification utilise une approche similaire à la tactique Coq `quote`, avec quelques améliorations : la fonction d'interprétation peut être plus complexe, elle supporte les quantificateurs, le contexte logique peut être réifié, et l'utilisateur n'a pas besoin de spécifier quels termes peuvent être des constantes.

Étant donné un but logique P et une procédure de décision φ, l'utilisateur peut essayer de prouver P par réflexion en tapant la commande "`reflection_f` φ" dans l'IDE. Why3 utilise alors la spécification de φ pour inverser syntaxiquement `interp` et fournir un terme $\ulcorner P \urcorner$ approprié, puis évalue $\varphi(\ulcorner P \urcorner)$. Le contrat de φ est ensuite utilisé comme indication de coupure pour

```
type t = Var int | And t t | ...
type vars = int → bool

function interp (x:t) (y:vars) : bool =
match x with
  | Var n → y n
  | And x1 x2 → interp x1 y && interp x2 y
  ...
end

let φ (x:t) : bool
  ensures { result → forall y. interp x y }
```

FIGURE 1 – Exemple de spécification pour φ

peut-être prouver P. Dans l'exemple de la figure 1, si $\varphi(\ulcorner P \urcorner) = \texttt{True}$, on prouve facilement P car la formule $\ulcorner P \urcorner$ a été choisie de sorte que la postcondition de $\varphi(\ulcorner P \urcorner)$ implique P. Si $\varphi(\ulcorner P \urcorner) = \texttt{False}$, la tentative de preuve par réflexion échoue.

2.2 Interprétation

Traditionnellement, les procédures de décision sont des fonctions écrites dans la logique du système de preuve de manière à pouvoir être interprétées par des prouveurs automatiques ou par le moteur de réécriture du système. Ceci limite leur pouvoir d'expressivité. Par exemple, les fonctions logiques de Why3 ne peuvent pas avoir d'effet de bord et leur terminaison doit être garantie par la décroissance structurelle d'un paramètre.

À l'inverse, notre approche permet d'écrire des procédures de décision comme de simples programmes WhyML. Elles peuvent donc utiliser toutes les fonctionnalités impératives du langage, telles que les références, les tableaux, les boucles ou les exceptions. Leur correction peut être prouvée à l'aide de Why3 et de prouveurs automatiques, et leur contrat est instancié par réification et utilisé comme indication de coupure. Cependant, elles n'ont pas d'implémentation dans la logique de Why3 et ne peuvent donc pas être interprétées par le moteur de réécriture de Why3 ou par des prouveurs automatiques.

Nous avons donc ajouté à Why3 un interpréteur capable d'évaluer n'importe quelle procédure de décision. Il opère sur un langage intermédiaire proche de ML, qui correspond aux programmes WhyML dont les assertions logiques et le code *ghost* ont été effacés, en supposant que le programme a été prouvé au préalable et que ses préconditions sont satisfaites. Ce code intermédiaire est produit par le mécanisme d'extraction existant, qui produit du code OCaml et C à partir de programmes WhyML prouvés.

Peu de travaux antérieurs sur la preuve par réflexion utilisent des procédures de décision avec effets de bord. On peut citer Claret *et al* [4]. Ils utilisent un encodage monadique des calculs avec effet dans Coq (par exemple, la non-terminaison). Les procédures de décision monadiques sont transformées en des programmes impurs exécutés en-dehors de Coq. Le résultat de ces calculs externe est utilisé comme une "prophétie" pour simuler l'exécution de la procédure de décision à l'intérieur de Coq. Puisque nous travaillons avec Why3, qui supporte nativement les calculs impurs, nous n'avons pas besoin d'un tel mécanisme lourd de simulation.

2.3 Correction

L'implémentation de notre mécanisme de preuves par réflexion consiste en deux additions à Why3 : une transformation de réification et un interpréteur de programmes WhyML.

Le code de la réification est complexe et relativement long, car il comporte beaucoup d'heuristiques qui permettent de donner à l'utilisateur un maximum de liberté dans la syntaxe des fonctions d'interprétation. Heureusement, la procédure de réification ne fait pas partie de la base de confiance de Why3. En effet, la réification se contente de deviner des termes appropriés à interpréter et demande à Why3 d'instancier le contrat de la procédure de décision avec. En supposant que l'utilisateur a prouvé la correction de la procédure de décision, la proposition instanciée est vraie, que l'algorithme de réification soit correct ou non. Un échec de la réification empêcherait une instantiation bien typée de la postcondition, ou alors l'indication de coupure résultante serait insuffisante pour prouver le but, mais ne permettrait en aucun cas de prouver quelque chose de faux.

L'interpréteur, en revanche, fait bien partie de la base de confiance. Heureusement, il est très simple, car il ne manipule que des valeurs concrètes. Il n'y a pas besoin d'évaluation partielle, d'exécution symbolique ou d'égalité polymorphe, ce qui le rend bien plus simple que le moteur de réécriture de termes logiques existant. Une autre raison pour cette simplicité est que le langage intermédiaire interprété a relativement peu de constructions. En effet, les transformations de programme réalisées par le mécanisme d'extraction éliminent certains comportements complexes du langage de surface, comme par exemple l'assignation parallèle.

3 Application : GMP

Dans cette section, nous présentons brièvement notre bibliothèque vérifiée d'arithmétique en précision arbitraire [13], et nous montrons comment nous avons éliminé un grand nombre d'assertions à l'aide d'une procédure de décision dédiée.

3.1 Une fonction de GMP

Dans GMP, les entiers naturels sont représentant par des tableaux little-endian d'entiers machine non signés, appelés *limbs*. On pose une base β, typiquement 2^{64}. Tout entier naturel N a une décomposition unique en base β : $N = \sum_{k=0}^{n-1} a_k \beta^k$, représentée par le tableau $a_0 a_1 \dots a_{n-1}$.

Dans les fonctions de bas niveau, il y a très peu de gestion de la mémoire ; les opérandes sont représentées par un pointeur sur leur limb le moins significatif et une taille de type `int32`. Étant donné un tel pointeur `a` et une taille `n`, en supposant que le pointeur est valide sur une longueur `n`, on note

$$\texttt{value a n} = \overline{a_0 \dots a_{n-1}} = \sum_{k=0}^{n-1} a_k \beta^k.$$

La figure 2 montre la fonction qui additionne deux entiers naturels de même longueur. Pour des raisons de lisibilité, la plupart des invariants et une partie de la spécification ont été omis. Il s'agit de l'algorithme d'addition enseigné à l'école : on additionne les opérandes limb par limb, en commençant par le moins significatif, et en maintenant une retenue.

Malheureusement, même cet algorithme simple pose problème pour les prouveurs automatiques. Pour prouver l'invariant de boucle, nous avons eu besoin de l'assertion à la ligne 18. Sa preuve consiste en une suite d'une dizaine d'étapes relativement simples (réécriture d'une

```
let wmpn_add_n (r x y: ptr limb) (sz: int32) : limb
  ensures { 0 ≤ result ≤ 1 }
  ensures { value r sz + (power radix sz) * result =
            value x sz + value y sz }
=
  let i = ref 0 in
  let lx = ref 0 in
  let ly = ref 0 in
  let c = ref 0 in
  while !i < sz do
    invariant { value r !i + (power radix !i) * !c = value x !i + value y !i }
    lx := get_ofs x !i;
    ly := get_ofs y !i;
    let res, carry = add_with_carry !lx !ly !c in
    set_ofs r !i res;
    c := carry;
    assert { value r (!i+1) + (power radix (!i+1)) * !c
             = value x (!i+1) + value y (!i+1)
             by value r (!i+1) + (power radix (!i+1)) * !c
               = value r !i + (power radix !i) * res
                 + (power radix !i) * c
               = ... (* 10+ sous-buts *) };
    i := !i + 1;
  done;
  !c
```

FIGURE 2 – Addition de deux entiers

égalité dans le contexte, application de la distributivité...), mais la taille de l'espace de recherche empêche les prouveurs d'aboutir en un temps raisonnable. Nous avons donc dû fournir de nombreuses indications de coupure à la main avec la construction `by` [5].

Cependant, avec un choix judicieux de coefficients, ce but, ainsi que de nombreux autres buts difficiles dans les preuves de notre bibliothèque, peut être vu comme une combinaison linéaire d'égalités présentes dans le contexte logique (c'est l'exemple en section 3.3). On peut donc espérer le prouver avec une procédure de décision relativement simple.

3.2 Procédure de décision

Nous avons implémenté une procédure de décision pour les systèmes d'équations linéaires dans un corps arbitraire (extraits du code en figure 3). Étant donné des égalités linéaires supposées valides dans le contexte logique, cette procédure tente de prouver une égalité linéaire en montrant que c'est une combinaison linéaire du contexte.

Pour ce faire, on représente le contexte et le but dans une matrice et on effectue un pivot de Gauss (fonction `gauss_jordan`). En cas de succès, on obtient un vecteur de coefficients et on vérifie que la combinaison linéaire correspondante du contexte est égale au but (fonction `check_combination`). Si ce n'est pas le cas, la fonction renvoie `False` et on ne peut rien prouver d'intéressant, puisque la postcondition a `result = True` comme prémisse.

Comme pour les tactiques Coq `lia` et `lra` [1], il s'agit d'une preuve par certificat, le certificat étant le vecteur de coefficients renvoyé par `gauss_jordan`. Il n'y a pas besoin de prouver l'algorithme du pivot de Gauss, ni de définir une sémantique pour la matrice qu'on lui passe comme argument. En fait, on ne prouve rien sur le contenu d'aucune matrice dans le programme. La preuve de la correction de la procédure de décision est donc très simple par rapport à sa

```
clone LinearEquationsCoeffs as C with type t = coeff

type expr = Term coeff int | Add expr expr | Cst coeff
type equality = (expr, expr)

let linear_decision (l: list equality) (g: equality) : bool
  requires { valid_ctx l }
  requires { valid_eq g }
  ensures { forall y z. result = True → interp_ctx l g y z }
  raises  { C.Unknown → true }
= ...
  fill_ctx l 0;
  let (ex, d) = norm_eq g in
  fill_goal ex;
  let ab = m_append a b in
  let cd = v_append v d in
  match gauss_jordan (m_append (transpose ab) cd) with
    | Some r → check_combination l g (to_list r 0 ll)
    | None → False
  end
```

FIGURE 3 – Procédure de décision pour les systèmes d'équations linéaires

longueur et à sa complexité conceptuelle.

Considérons maintenant le contrat de la procédure de décision. La postcondition dit que si la procédure renvoie `True`, le but est vrai pour toutes les valuations `y` et `z` des variables telles que les égalités du contexte sont vraies. Les préconditions `valid_ctx` et `valid_eq` demandent que les entiers utilisés comme indices de variables dans le contexte et le but soient positifs, ce qui est nécessaire pour prouver la sûreté des accès dans les tableaux. La nature de la procédure de réification assure que ces préconditions seront toujours vérifiées en pratiques, mais comme on ne fait pas confiance à la réification, l'utilisateur doit prouver les préconditions explicitement. En pratique, ça ne pose aucun problème aux solveurs SMT. Enfin, la clause `raises` exprime que la procédure peut lever une exception non rattrapée (typiquement une erreur arithmétique, puisqu'on autorise les opérations du corps à être partielles). Dans ce cas, rien n'est prouvé.

On peut remarquer que la procédure de décision est indépendante de Why3, au sens où elle ne contient pas de méta-instructions pour la réification ou quoi que ce soit de lié au fonctionnement interne de Why3. On pourrait imaginer trouver du code similaire dans un prouveur automatique.

Enfin, le type `coeff` est laissé abstrait dans ce module. L'utilisateur peut l'instancier avec n'importe quel type qui vérifie certaines propriétés (données sous forme d'axiomes omis de la figure). Essentiellement, `coeff` doit implémenter les opérations élémentaires d'un corps, mais certaines de ces opérations peuvent être partielles et lever l'exception `Unknown`. Considérons maintenant comment choisir les coefficients dans le cas de GMP.

3.3 Coefficients

Voici une version simplifiée du contexte et du but donnés par Why3 au niveau de la longue assertion dans la boucle de `wmpn_add_n` (figure 2 ligne 18). Les variables `r1` et `c1` dénotent les valeurs de `r` et `c` au début du corps de la boucle (avant les modifications des lignes 13-14).

On remarque que la combinaison linéaire $\texttt{H5} - \texttt{H4} - \texttt{H3} + \texttt{H2} + \beta^{\texttt{i}} \cdot \texttt{H1} + \texttt{H}$ se simplifie en une égalité équivalente à `g`. Pour prouver cela, la procédure de décision a besoin d'inclure dans les coefficients des puissances de β (`radix` dans le code WhyML), et de permettre les exposants

```
axiom H:  value r1 i + (power radix i) * c1 = value x i + value y i
axiom H1: res + radix * c = lx + ly + c1
axiom H2: value r i = value r1 i
axiom H3: value x (i+1) = value x i + (power radix i) * lx
axiom H4: value y (i+1) = value y i + (power radix i) * ly
axiom H5: value r (i+1) = value r i + (power radix i) * res
goal g:   value r (i+1) + power radix (i+1) * c = value x (i+1) + value y (i+1)
```

symboliques (car `i` est une variable).

Plus précisément, les coefficients sont le produit d'un rationnel et d'une puissance (symbolique) de β. On peut définir sans problème une multiplication et un inverse multiplicatif sur ces coefficients. L'addition est partielle, car on ne peut additionner que des coefficients dont les exposants sont égaux. Si ce n'est pas le cas, l'addition lève une exception, ce qui est permis par la procédure de décision. Notons que les exposants n'ont pas besoin d'être structurellement égaux mais seulement égaux pour toutes valuations des variables présentes dans les exposants, ce qui peut être prouvé automatiquement par une procédure de décision secondaire très simple appelée par la procédure primaire.

4 Conclusion

Cet article présente deux contributions. Premièrement, nous avons développé un mécanisme pour les preuves par réflexion qui utilise des programmes WhyML impératifs comme procédures de décision. Deuxièmement, nous avons implémenté et vérifié une procédure de décision pour les systèmes d'équations linéaires avec coefficients symboliques. Nous avons utilisé cette procédure pour prouver de nombreux buts dans notre formalisation d'algorithmes de GMP. Instanciée avec un autre ensemble de coefficients, elle pourrait être réutilisée dans d'autres contextes.

Nous avons revisité toutes nos preuves existantes d'algorithmes d'addition, de soustraction et de multiplication, qui demandaient précédemment de nombreuses indications de coupure manuelles. La procédure de décision a été capable de décharger toutes les longues assertions (dans la même veine que celle de la figure 2). Cette section de notre bibliothèque faisait auparavant 1660 lignes. Les 660 lignes de code de programmes n'ont évidemment pas changé, mais les 1000 lignes de spécification et preuve ont été diminuées de moitié. Qui plus est, une grande partie des 500 lignes restantes est constituée de contrats de fonctions et d'invariants de boucle, essentiellement incompressibles.

Le but le plus difficile auquel nous avons pu appliquer notre procédure de décision (une assertion dans la preuve du cas général de la division longue) impliquait un pivot de Gauss sur une matrice de taille 150×90 environ, et le temps d'exécution était de 3 secondes environ, ce qui est acceptable du point de vue de l'expérience utilisateur. Si le temps d'exécution devient problématique pour de plus grandes matrices, une option pour améliorer les performances serait d'extraire la procédure de décision vers OCaml et d'exécuter le binaire résultant plutôt que notre interpréteur.

Bien que notre procédure de décision ne traite que des systèmes d'équations linéaires, nous l'avons utilisé avec succès dans les preuves d'algorithmes de multiplication, division et décalages logiques pour prouver des buts qui semblent complètement non-linéaires de prime abord. Dans ces cas, nous avons dû fournir une ou deux indications de coupure pour gérer les parties non-linéaires du raisonnement, mais c'est très acceptable puisque la plupart de ces buts demandaient plus de cinquante assertions auparavant. On peut penser que ce nouvel outil nous permettra de vérifier de nouveaux algorithmes de GMP bien plus efficacement.

L'approche présentée dans cet article n'est pas limitée à Why3 en principe. Pour développer un mécanisme similaire, il suffit de pouvoir spécifier et prouver des procédures de décision, et de pouvoir exécuter des programmes vérifiés. Il ne serait donc sans doute pas trop difficile d'adapter notre mécanisme pour des plateformes de vérification telles que Leon [2] ou Dafny [10].

Références

[1] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In Thorsten Altenkirch and Conor McBride, editors, *International Workshop on Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 48–62, Nottingham, UK, 2007.

[2] Régis William Blanc, Etienne Kneuss, Viktor Kuncak, and Philippe Suter. An overview of the Leon verification system: Verification by translation to recursive functions. In *4th Annual Scala Workshop*, 2013.

[3] Amine Chaieb and Tobias Nipkow. Proof synthesis and reflection for linear arithmetic. *Journal of Automated Reasoning*, 41(1):33–59, 2008.

[4] Guillaume Claret, Lourdes del Carmen González Huesca, Yann Régis-Gianas, and Beta Ziliani. Lightweight proof by reflection using a posteriori simulation of effectful computation. In *4th International Conference on Interactive Theorem Proving*, volume 7998 of *Lecture Notes in Computer Science*, pages 67–83. Springer, July 2013.

[5] Martin Clochard. Preuves taillées en biseau. In *Vingt-huitièmes Journées Francophones des Langages Applicatifs (JFLA)*, Gourette, France, January 2017.

[6] Martin Clochard, Léon Gondelman, and Mário Pereira. The Matrix reproved. *Journal of Automated Reasoning*, 60(3):365–383, 2017.

[7] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[8] Benjamin Grégoire and Assia Mahboubi. Proving equalities in a commutative ring done right in Coq. In Joe Hurd and Tom Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, pages 98–113, Oxford, UK, August 2005.

[9] John Harrison. Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI International Cambridge Computer Science Research Centre, 1995.

[10] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR-16*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[11] Guillaume Melquiond and Raphaël Rieu-Helft. A Why3 Framework for Reflection Proofs and its Application to GMP's Algorithms. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *9th International Joint Conference on Automated Reasoning*, number 10900 in Lecture Notes in Computer Science, pages 178–193, Oxford, United Kingdom, July 2018.

[12] Niels Moller and Torbjörn Granlund. Improved division by invariant integers. *IEEE Transactions on Computers*, 60(2):165–175, 2011.

[13] Raphaël Rieu-Helft, Claude Marché, and Guillaume Melquiond. How to get an efficient yet verified arbitrary-precision integer library. In *9th Working Conference on Verified Software: Theories, Tools, and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 84–101, Heidelberg, Germany, July 2017.

Merci à

Université
de Strasbourg

list
ceatech

TRUST IN SOFT

OCaml PRO

GPL
Génie de la Programmation et du Logiciel
GdR
Groupement de Recherche