Des briques de calcul formel plus solides avec Capla

Josué Moreau

Des briques de calcul formel plus solides avec Capla

Josué Moreau[1]

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

Cet article présente Capla, un langage de programmation dédié à l'écriture des algorithmes de bas niveau utilisées en calcul formel. Ces algorithmes (GMP, BLAS/LAPACK, etc) sont habituellement écrits en C, Fortran et assembleur et utilisent les pointeurs de manière intensive. Le langage que nous proposons possède une syntaxe proche de celles de C ou Rust et un système de types similaire à celui de SPARK. Ce langage n'a pas de comportement indéfini et est conçu pour faciliter la vérification déductive de programmes, tout en étant suffisamment bas niveau pour être adapté aux applications nécessitant beaucoup de calculs. Cet article décrit le langage, ainsi que son compilateur, basé sur CompCert. La sûreté du langage, ainsi que la correction du compilateur, ont été formellement prouvées à l'aide de l'assistant de preuves Coq. Cet article est un résumé d'un travail précédemment publié.

1 Introduction

Bien que les systèmes de calcul formel soient majoritairement utilisés pour effectuer des calculs symboliques, ils reposent en interne sur des bibliothèques de bas niveau dédiées à la manipulation de certains objets mathématiques. Parmi ces bibliothèques, on peut compter GMP pour la manipulation de grands entiers, BLAS/LAPACK pour l'algèbre linéaire, ou encore FFTW pour la transformée rapide de Fourier. Ces bibliothèques sont écrites en C pour GMP et FFTW et en Fortran pour BLAS/LAPACK, sans compter une importante quantité d'assembleur écrit à la main et spécifique à chaque architecture ou modèle de processeur.

L'un des avantages des langages C et Fortran est qu'ils ne gênent pas le programmeur, dans le sens où ils permettent l'écriture de code efficace dont les performances sont prédictibles par le programmeur. Ils bénéficient également de compilateurs hautement optimisants. Cependant, ces avantages viennent souvent au détriment de la correction ou de la sûreté du code généré.

En effet, ces langages ne sont pas sûrs (dans le sens où aucun comportement indéfini ne se produit dans un programme bien typé) et leurs sémantiques sont souvent compliquées et inadaptées à la vérification de programmes dont la preuve est déjà mathématiquement très difficile. Les compilateurs, très optimisants, sont également très compliqués, ce qui implique une confiance limitée dans la correction du code généré. Par exemple, dans GMP 5.1.1, la fonction d'exponentiation modulaire `mpz_powm_ui` calculait un résultat incorrect si le nombre passé à l'exponentielle dépassait 15000 chiffres. De plus, GMP a, par le passé, régulièrement eu des problèmes de compilation sous MacOS, par exemple avec XCode 11 à 11.2 où le code généré était incorrect [1].

1. Les bugs que nous donnons ici sont listés sur cette page : `https://gmplib.org/#STATUS`.

La solution que nous proposons à ce problème est Capla, un langage sûr adapté à l'écriture de tels algorithmes, ainsi que son compilateur, vers de l'assembleur, formellement vérifié. Le travail décrit dans cet article a fait l'objet d'une publication à ICFP'24 [MM24].

Le langage Capla, décrit en section 2, est centré sur l'utilisation des tableaux, car ils sont la structure de base de tous les algorithmes de bas niveau des bibliothèques de calcul formel. La taille des tableaux fait explicitement partie de la signature des fonction. Notre langage garantit également une politique de non-alias similaire à celle de Rust, afin de permettre une compilation efficace, et fournit une notion d'emprunt. Contrairement à Rust, cependant, le système de types de Capla n'inclut pas de durée de vie, car celles-ci compliquent le système de types et ne fournissent pas d'expressivité supplémentaire pour les applications qui nous intéressent.

Un aperçu de la sémantique de Capla est décrite en section 3. La sûreté de l'exécution des programmes bien typés a été formellement vérifiée. Concernant les accès dans les bornes des tableaux, ils ne sont pas garantis par le typage. Accéder en dehors des bornes n'est, cependant, par un comportement indéfini car une erreur sera levée à l'exécution dans ce cas.

Notre langage dispose également d'un compilateur, décrit en section 4. Il se base sur le compilateur formellement vérifié CompCert [Ler09b, Ler09a]. La correction de notre compilateur a été prouvée à l'aide de l'assistant de preuves Coq. La section 4 décrit également des tests de performances de code généré.

Travaux liés

Un certain nombre de travaux tentent de palier aux problèmes que nous énonçons ici. Le travail le plus proche de ce que nous proposons est Checked C [ERHT18, RLS+19, LLP+22], qui construit un langage sûr au dessus de C qui permet de spécifier les bornes inférieures et supérieures des adresses accessibles par les pointeurs. La gestion des tableaux est très proche de la nôtre, bien que la leur soit légèrement plus expressive. Une sémantique pour une version simplifiée de Checked C a été formalisée en Coq, de même que la preuve de sûreté. En particulier, il a été prouvé que dans tout programme qui mélange C et Checked C, un comportement indéfini provient nécessairement de la partie écrite en C. Cependant, la formalisation n'inclut pas la vérification du compilateur et la partie sûre n'inclut pas la désallocation de mémoire.

Pour ce qui est de la gestion des alias, le système de type de Rust a été une inspiration. Notre système de type est, cependant, plus simple et ne parle pas des durées de vie. Notre approche est donc davantage comparable à celle du fragment SPARK d'Ada [JDM+20]. En revanche, il n'existe aucune preuve formelle de ce dernier.

Enfin, il existe de nombreux travaux sur les langages manipulant des tableaux. Il y a, en particulier, bon nombre de systèmes de types pour encoder les tailles et les formes des tableaux [TG09, XP98, HE21, CPP23]. La principale différence entre ces approches et la nôtre provient du choix entre vérification statique ou dynamique des accès aux tableaux. La plupart de nos vérifications sont dynamiques, comme c'est le cas pour Rust et Ada, alors que les implémentations basées sur des types dépendants utilisent souvent des vérifications statiques.

2 Description du langage

Tout d'abord, la plupart des bibliothèques de calcul formel reposent sur des tableaux et, plus généralement, des tableaux multidimensionnels. Ceux-ci sont différents des tableaux de tableaux car ces derniers sont des tableaux de pointeurs, tandis que les tableaux multidimensionnels sont représentés de manière linéaire en mémoire. La représentation en Capla est *row-major* : le code `t[x, y]` est interprété comme `t[x * m + y]`, avec $0 \leq \mathtt{y} \leq \mathtt{m}$.

Dans les bibliothèques de haut niveau, les tableaux multidimensionnels sont habituellement représentés par une structure qui contient à la fois son contenu et sa forme. Dans le cas

```
fun zdotu(n: i32, zx: [f64; 1 + (n - 1) * incx, 2], incx: i32,
                  zy: [f64; 1 + (n - 1) * incy, 2], incy: i32,
          res: mut [f64; 2]) {
  res[0] = 0.; res[1] = 0.;
  if n <= 0 return;

  if incx == 1 && incy == 1 {
    for i: i32 = 0 .. n {
      res[0] = res[0] + (zx[i,0] * zy[i,0] - zx[i,1] * zy[i,1]);
      res[1] = res[1] + (zx[i,1] * zy[i,0] + zx[i,0] * zy[i,1]);
    }
  } else {
    let ix: i32 = 0;
    let iy: i32 = 0;
    if incx < 0 { ix = (-n + 1) * incx; }
    if incy < 0 { iy = (-n + 1) * incy; }

    for i: i32 = 0 .. n {
      res[0] = res[0] + (zx[ix,0] * zy[iy,0] - zx[ix,1] * zy[iy,1]);
      res[1] = res[1] + (zx[ix,1] * zy[iy,0] + zx[ix,0] * zy[iy,1]);
      ix = ix + incx;
      iy = iy + incy;
    }
  }
}
```

Figure 1. Implémentation de la fonction `zdotu` de BLAS

des bibliothèques bas niveau, telles que BLAS ou la couche *mpn* de GMP, les tailles de tableaux sont passées explicitement dans les paramètres des fonctions. Cela permet plus de flexibilité, étant donné qu'une taille plus petite que celle allouée peut être passée. Notre langage garantit qu'à tout moment l'espace alloué à chaque tableau est au moins égal à sa taille. Les tailles explicites permettent également d'établir les liens entre les tailles des différents tableaux passés en arguments et donc de coller le plus possible à la définition de l'opération mathématique sous-jacente. Par exemple, la multiplication de matrices peut avoir la signature suivante dans notre langage :

```
fun matrix_mul(a: [i32; m, n], b: [i32; n, p], dest: mut[i32; m, p],
               m n p:u64)
```

Un autre avantage de ces tailles explicites est illustré ci-dessous par l'addition de vecteurs. Les trois tableaux `a`, `b` et `c` ayant la même taille, la validité des accès aux tableaux n'a pas besoin d'être vérifiée séparément.

```
fun add_vectors(a b: [i64; n], dest: mut [i64; n], n: u64) {
    for i: u64 = 0 .. n { dest[i] = a[i] + b[i]; }
}
```

Enfin, les tableaux passés en argument peuvent être annotés avec `mut` pour indiquer qu'ils sont mutables et que les modifications sont visibles par l'appelant. La politique de non-alias de notre langage prend en compte cette notion de mutabilité et garantit qu'à chaque instant tout tableau mutable ne peut être accédé que d'une seule manière. Contrairement à Rust, notre langage ne possède pas de durées de vie, ce qui implique que les permissions s'étendent en profondeur sur les tableaux. Il n'est donc pas possible d'avoir, par exemple, un tableau mutable de tableaux immuables.

Pour clore cette présentation de notre langage, la figure 1 présente une implémentation en Capla de la fonction `zdotu` de BLAS qui calcule le produit scalaire de deux vecteurs `zx` et `zy`

de `n` éléments chacun, et dont les distances entre deux éléments successifs sont respectivement `incx` et `incy`. Ainsi, la documentation BLAS énonce que les tailles respectives de `zx` et `zy` sont $1 + (\texttt{n} - 1) \times |\texttt{incx}|$ et $1 + (\texttt{n} - 1) \times |\texttt{incy}|$, mais cela n'apparaît pas dans la signature de la fonction Fortran :

```
complex*16 function zdotu(n, zx, zy, incx, incy)
  integer incx, incy, n
  complex*16 zx(*), zy(*)
```

Notre code suit précisément celui de l'implémentation de référence de BLAS [2]. Il conserve la même signature et représentation des données, à l'exception du dernier argument qui permet de renvoyer la paire en résultat, car Capla ne supporte pas encore les tableaux renvoyés par une fonction. Notre code peut ainsi être compilé et appelé à la place de l'implémentation traditionnelle de `zdotu`. Il n'est pas encore possible d'appeler de fonctions dans les expressions de tailles, d'où l'absence de la valeur absolue dans la signature de la fonction Capla.

3 Sémantique et sûreté

Sémantique par copy-restore

La politique de non-alias décrite précédemment permet de munir Capla d'une sémantique très simple. Plus précisément, contrairement à la sémantique de CompCert [LB08] qui utilise une mémoire, des pointeurs et dont les valeurs qui correspondent aux tableaux contiennent un emplacement mémoire, nos valeurs de tableau contiennent directement leur contenu. Les autres valeurs possibles sont des valeurs primitives : booléens, entiers, flottants. Les environnements locaux, notés E, sont des fonctions partielles $x \rightharpoonup v$ des identifiants vers les valeurs.

$$v ::= \texttt{Vbool}\ b \mid \texttt{Vint}\ n \mid \texttt{Vfloat}_{32}\ f \mid ... \mid \texttt{Varr}\ [v_1, v_2, ...]$$

Il n'existe donc aucune mémoire globale (utilisant une notion de pointeurs), mais seulement des environnements locaux. Lors des appels de fonctions, les tableaux passés en argument sont copiés en profondeur et les modifications effectuées par la fonction appelée sont ensuite restaurées, à nouveau par une copie en profondeur, vers l'appelant. Il s'agit d'une sémantique par *copy-restore*.

Ce mécanisme possède un certain nombre d'avantages. L'absence de mémoire globale et de pointeurs permet d'éviter l'utilisation ultérieure d'une logique de séparation pour la preuve de programmes. De plus, avec la distinction entre tableaux mutables et immuables, tout tableau non annoté par `mut` est trivialement non modifié. La preuve de programmes s'en retrouve ainsi simplifiée, en rendant certains invariants inutiles.

Nous présentons maintenant, en figure 2, un sous-ensemble simplifié des règles de la sémantique de Capla. Hormis la notion de mémoire, cette dernière s'inspire grandement des sémantiques de CompCert [Ler09a]. Il s'agit d'une sémantique à petits pas, utilisant des continuations pour décrire la suite de l'exécution du programme.

Afin d'accéder à un emplacement donné dans nos valeurs structurées, notre sémantique utilise des chemins dans les environnements locaux. La règle WRITE écrit à l'emplacement $x[\vec{e}]$, qui est évalué en un chemin sémantique $x \cdot \vec{n}$, dans lequel est écrit la valeur évaluée de e'. Plus précisément, $\vec{e}$ est une séquence d'expressions représentant des indices dans des tableaux [3] tandis que $\vec{n}$ est la séquence d'indices (après évaluation) permettant d'accéder au même

2. `https://www.netlib.org/lapack/explore-html/d1/dcc/group__dot_ga6b0b69474b384d45fc4c7b1f7ec5959f.html`

3. La séquence doit être suffisamment longue, en particulier dans le cas des tableaux de tableaux, pour atteindre une case contenant une valeur primitive.

$$\textsc{Write}\ \frac{\texttt{perm}(x) \geq \texttt{Mutable} \qquad E \vdash \vec{e} \Rightarrow \vec{n} \qquad E \vdash e' \Rightarrow v \qquad \texttt{primitive}(v) \qquad \dots}{(E, \langle x[\vec{e}] = e' \rangle, k) \to (E[x \cdot \vec{n} \leftarrow v], \langle\rangle, k)}$$

$$\textsc{Alloc}\ \frac{\texttt{perm}(x) = \texttt{Owned} \qquad \dots}{(E, \langle \texttt{alloc}\ x \rangle, k) \to (E[x \leftarrow \texttt{Varr}\ \ [0, 0, ...]], \langle\rangle, k)}$$

$$\textsc{Free}\ \frac{\texttt{perm}(x) = \texttt{Owned} \qquad \dots}{(E, \langle \texttt{free}\ x \rangle, k) \to (E \setminus \{x\}, \langle\rangle, k)}$$

$$\textsc{Call}\ \frac{\begin{array}{c} E \vdash \vec{a} \Rightarrow \vec{p} \qquad x = \texttt{params}(f) \\ \forall i\ j, i \neq j \wedge \texttt{perm}(p_i) \geq \texttt{Mutable} \Rightarrow p_i \not\preceq p_j \wedge p_j \not\preceq p_i \\ m = \{\ (p_i, x_i) \mid \texttt{perm}(x_i(f)) = \texttt{Mutable}\ \} \\ \dots \end{array}}{(E, \langle y = f(\vec{a}) \rangle, k) \to (\{x_i \mapsto E[p_i]\}, \texttt{body}(f), \texttt{Kreturnto}(y, E, m, k))}$$

$$\textsc{Return}\ \frac{}{(E_f, \langle \texttt{return}\ v \rangle, \texttt{Kreturnto}\,(y, E, \{(p_i, x_i) \mid i\}, k)) \to (E\,[p_i \leftarrow E_f[x_i]]\,[y \leftarrow v]\,, \langle\rangle, k)}$$

Figure 2. Sélection simplifiée de règles de la sémantique de Capla

emplacement que $\vec{e}$ mais de manière unidimensionnelle (les tableaux multidimensionnels sont représentés linéairement dans l'environnement).

La règle Write ne fait qu'écrire dans l'environnement local de la fonction en cours d'exécution, même si le tableau modifié provient d'une fonction parente. De même, les règles Alloc et Free ne manipulent que l'environnement local, respectivement en y ajoutant un tableau initialisé ou en le supprimant.

La règle Call évalue les arguments vers des chemins sémantiques et vérifie qu'aucun de ceux qui sont passés à des paramètres mutables ne sont en situation d'alias, c'est-à-dire qu'ils ne doivent pas être préfixes d'un autre chemin en argument (nous notons $p \preceq q$ pour « p est un préfixe de q ») et aucun de leurs préfixes ne doivent être présents dans les autres arguments. Le non-respect de cette condition est un comportement indéfini dans notre sémantique (et devra donc être détecté par typage). L'appel est ensuite effectué, l'environnement local de la fonction appelée est construit et l'ensemble m des chemins passés en argument mutable est stockée dans la continuation.

Enfin, la règle Return restaure les modifications effectuées par la fonction appelée vers la fonction appelante, leurs environnements locaux étant respectivement E_f et E. Dans cette règle, l'ensemble des (p_i, x_i), construite par la règle Call, correspond à chaque chemin p_i dans E d'un tableau passé à un paramètre mutable x_i dans E_f. L'environnement E est donc bien mis à jour avec les nouvelles valeurs de tableaux provenant de E_f.

Sûreté du typage

La sémantique que nous présentons ici est sûre dans le cas d'un programme bien typé. Certaines hypothèses des règles de sémantique ne sont, cependant, pas garanties par le typage. C'est le cas, par exemple, des vérifications de bornes lors d'un accès à un tableau, ou encore de la division par un entier non nul. Il existe ainsi un certain nombre de règles de sémantique, non présentes en figure 2, qui lèvent une erreur dynamique dans les cas où ces hypothèses ne sont pas vérifiées.

Le typage de notre langage se concentre sur la vérification des deux principaux comportements indéfinis restants dans notre sémantique. Le premier, énoncé précédemment, est le cas où des arguments mutables sont en situation d'alias lors de l'appel de fonction. Le second correspond à l'accès à une variable non initialisée. Le typage effectue donc, en tout point du programme, une sur-approximation des variables non initialisées et rejette le programme si

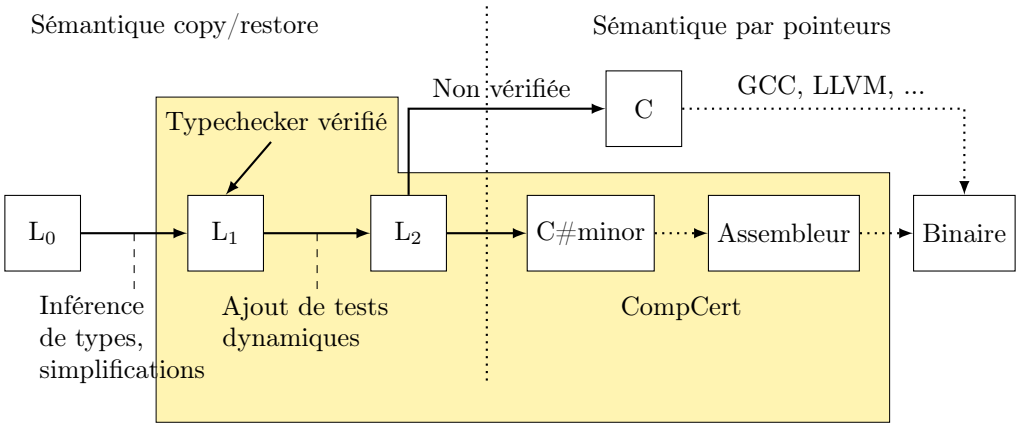

Figure 3. Architecture du compilateur (le travail présenté ici porte sur les flèches pleines)

l'une d'entre-elles est accédée.

Le théorème de sûreté du typage est le suivant :

Théorème 1 (Sûreté du typage)**.** *Soit un programme Capla bien typé, si tous les invariants sont vérifiés pour un état non final s, alors il existe un état t tel que $s \to t$ et, pour tout t tel que $s \to t$, tous les invariants sont vérifiés par t.*

Dans ce théorème, les invariants couvrent principalement deux choses. Premièrement, le contenu de l'environnement a bien le type donné par l'environnement de typage. Il s'agit de l'invariant classique des langage typés. Deuxièmement, les tailles des tableaux sont valides, c'est-à-dire que l'espace alloué pour chaque tableau est au moins aussi grand que sa taille.

Le théorème de sûreté du typage a été prouvé en Coq et il suppose que le typechecker, également écrit en Coq, accepte le programme. La principale difficulté pour la preuve de ce théorème réside dans le traitement des variables non allouées et, de manière secondaire, dans la gestion des tableaux, en particulier dans les appels et retours de fonctions.

4 Compilation

Compilation formellement vérifiée

La compilation de Capla vers le langage assembleur s'effectue en plusieurs passes qui sont résumées en figure 3. La première d'entre-elles part du langage Capla, que nous nommerons ici L_0, et effectue certaines simplifications ainsi que de l'inférence de types pour aboutir à un langage L_1. Ce dernier est le langage d'entrée de la partie formellement vérifiée de notre compilateur. Le typage et son théorème de sûreté, décrits brièvement en section 3, sont respectivement définis et prouvés sur ce langage et sa sémantique. L_1 est donc un langage sans comportements indéfinis pour les programmes bien typés.

Par la suite, L_1 est traduit vers un autre langage L_2 dont la sémantique est similaire à celle de L_1, à l'exception que toutes les règles levant des erreurs dynamiques ont été retirées. Pour que cette traduction préserve la sémantique, et ainsi éviter que ces comportements indéfinis ne se produisent, des tests dynamiques, sous forme d'assertions, sont ajoutés au programme. Les accès aux tableaux, les divisions et les conversions de nombres flottants vers les entiers lèvent ainsi des erreurs dynamique si les assertions ne sont pas satisfaites. Du point de vue de la correction de cette passe de compilation, la principale difficulté a été de prouver la correction et la complétude des tests ajoutés, au regard des cas d'erreurs prévus par la sémantique de L_1.

Le langage L_2 est ensuite traduit vers le langage intermédiaire C#minor [BDL06] de CompCert [4]. Cette traduction est très simple car il s'agit quasiment d'une traduction 1-pour-1 vers C#minor. La politique de non-alias permet de remplacer les tableaux par des pointeurs et les copies – coûteuses – lors des appels et retours de fonctions sont transformées en passages de pointeurs. Cela garantit un code généré efficace, tout en gardant, du point de vue de l'utilisateur, une sémantique simple.

Cette traduction, bien que simple à écrire, est difficile à prouver car elle demande de maintenir un certain nombre d'invariants, dont des propriétés de séparation dans la mémoire de C#minor. Plus précisément, il s'agit de maintenir l'invariant qu'à chaque instant tous les tableaux mutables dans L_2 sont liés à des emplacements mémoires séparés dans la mémoire de C#minor.

Notre compilateur utilise enfin CompCert pour traduire le code C#minor vers de l'assembleur. Toutes les architectures compatibles avec CompCert sont ainsi également compatibles avec notre langage [5]. En combinant les résultats de correction de CompCert et ceux de nos passes, nous obtenons le théorème suivant. La notation $\Leftrightarrow$ est une relation entre les environnements de L_1 et la mémoire de ASM.

Théorème 2 (Préservation de la sémantique par simulation avant)**.** *Soient s et t deux états de la sémantique de L_1. Si $s \xrightarrow{L_1} t$, alors pour tout état s' de l'exécution en assembleur tel que $s \Leftrightarrow s'$, il existe un état t' tel que $s' \xrightarrow{ASM}{}^{*} t'$ et $t \Leftrightarrow t'$.*

Ce théorème énonce une simulation avant entre L_1 et l'assembleur. Or, la sémantique de l'assembleur est déterministe, ce qui implique le théorème de correction du compilateur (*i.e.*, préservation de la sémantique par simulation arrière). Celui-ci utilise une fonction de mesure $|\cdot|$ qui permet de traiter les cas où la sémantique progresse dans l'assembleur mais pas dans L_1, ce qui est le cas lorsqu'une instruction de L_1 a été compilée en une séquence de plusieurs instructions assembleur.

Corollaire 1 (Correction du compilateur)**.** *Soient s et t deux états de la sémantique de l'exécution en assembleur. Si $s \xrightarrow{ASM} t$, alors soit $|t| < |s|$, soit pour tout état s' de L_1 tel que $s \Leftrightarrow s'$, il existe un état t' tel que $s' \xrightarrow{L_1} t'$ et $t \Leftrightarrow t'$.*

Sortie non vérifiée et tests de performance

La compilation précédemment décrite n'est pas adaptée aux cas où la performance du code généré est plus importante que les garanties formelles de bonne compilation sur ce même code. En effet, CompCert est un compilateur peu optimisant, en comparaison avec GCC et LLVM. Une sortie non vérifiée du langage L_2 vers C a donc été ajoutée au compilateur. Cette traduction produit du code très verbeux, effectuant de nombreux casts, car nos opérations arithmétiques signées sont modulaires, contrairement à celles de C. Le code généré est également annoté à l'aide de mots clés comme `restrict`. Ce dernier permet aux compilateurs d'effectuer un certain nombre d'optimisations (par exemple, réordonner des instructions d'accès à la mémoire, afin de vectoriser le code) car il garantit des accès sans alias.

Des tests de performance sur un sous-ensemble de fonctions de BLAS (niveaux 1 et 2) et GMP sont résumés en table 1. Les versions de références sont BLAS/LAPACK [6] 3.12.0 et GMP [7] 6.3.0. GMP a été compilé pour une architecture x86-64 générique, ce qui veut dire

4. Il n'est pas facile d'utiliser un langage de plus haut niveau, tel que Clight, car il ne nous donne pas suffisamment de contrôle sur la traduction des opérations arithmétiques et est pollué par les types du C.

5. Notre compilateur effectue, pour le moment, la supposition d'une architecture 64 bits pour simplifier les passes de compilations et les preuves. Cependant, ces dernières ne reposent pas intrinsèquement sur cette supposition et il serait simple d'ajouter les architectures 32 bits.

6. `https://www.netlib.org/lapack/`

7. `https://gmplib.org/`

Table 1. Temps d'exécution pour une sélection de fonction de BLAS et GMP, en fonction du backend utilisé pour compiler le code Capla, relativement aux bibliothèques de référence (une valeur inférieure à 1 signifie que le code Capla est plus performant)

	CompCert	GCC	LLVM	GCC avec assertions	LLVM avec assertions
`zdotu`	2.25	1.01	0.93	1.06	0.82
`saxpy`	5.37	4.25	0.78	3.68	0.78
`sgemv`	2.86	1.32	0.96	1.32	0.87
`dgemv`	1.73	1.03	0.59	0.68	0.53
`dtrsv` (N)	2.87	1.32	1.78	0.92	1.60
`dtrsv` (T)	2.30	1.14	1.35	0.91	0.90
`mpn_addmul_1`	2.74	1.20	1.06	–	–

que le code exécuté est de l'assembleur écrit à la main, mais non optimisé pour un modèle spécifique de processeur.

Tous les tests ont été effectués avec les version 13.2.0 et 17.0.6 de GCC et LLVM, respectivement, et le niveau d'optimisation `-O2 -ftree-vectorize`. Pour la partie formellement vérifiée, notre code se base sur la branche `master` post 3.14 de CompCert.

Les fonctions BLAS testées calculent, respectivement, le produit scalaire (`zdotu`), l'addition de vecteurs $\vec{y} \leftarrow \alpha\vec{x} + \vec{y}$ (`saxpy`), la multiplication matrice/vecteur (`sgemv` et `dgemv`) et la résolution d'un système triangulaire (`dtrsv`), potentiellement en transposant la matrice. Du côté de GMP, la fonction `mpn_addmul_1` correspond à l'opération $x \leftarrow \alpha x + y$, où x et y sont des entiers arbitrairement grands et α est un mot machine.

La chaîne de compilation complètement vérifiée produit du code deux à cinq fois plus lent que le code de référence. En effet, le code produit n'est pas vectorisé, ce qui est rédhibitoire pour ce genre de bibliothèque. Un code deux fois plus lent reste cependant acceptable pour un compilateur formellement vérifié.

GCC et LLVM sont quant à eux capables de supprimer la majeure partie des tests dynamiques et de vectoriser le code. Il en résulte de bonnes performances, avec un code généré environ 30% plus lent que le code de référence dans le pire des cas (`dtrsv`) et 70% plus rapide dans le meilleur des cas (`dgemv`). Pour les tests dynamiques restants, certains peuvent être supprimés en ajoutant des assertions manuellement dans le programme source. Par exemple, les assertions suivantes peuvent être ajoutées au début de la première branche de la condition `incx == 1 && incy == 1` dans le code de `zdotu` en figure 1.

```
assert (1 + (n - 1) * incx == n);
assert (1 + (n - 1) * incy == n);
```

Ces deux assertions permettent à LLVM de supprimer la totalité des tests dynamiques dans la première boucle et, ainsi, de vectoriser le code. Le gain de performances après ajout de ces assertions peut être observé dans les deux dernières colonnes de la table 1, avec du code jusqu'à 90% plus rapide par rapport à l'implémentation de référence (`dgemv`). Pour `mpn_addmul_1`, il n'est pas nécessaire d'ajouter manuellement des assertions car les deux compilateurs sont capables d'éliminer tous les tests dynamiques.

5 Conclusion

Dans cet article, nous avons présenté Capla, un langage sûr adapté à l'implémentation des algorithmes de bas niveau des bibliothèques de calcul formel. Étant donné que la plupart de ces algorithmes sont écrits dans des langages de bas niveau, C et Fortran, notre langage tente de rester le plus proche possible de ces langages, tout en ajoutant une politique de non-alias qui permet des optimisations et simplifie la preuve de programmes.

La majeure partie du travail présenté dans cet article consiste en la conception d'un compilateur pour notre langage. Sa sémantique a été formalisée à l'aide de l'assistant de preuves Coq, de même que la preuve de sûreté du typage et la correction du compilateur. Le développement Coq représente environ 2400 lignes de code, 7000 lignes de spécification et 11000 lignes de preuve.

Un certain nombre de constructions manquent toutefois à Capla. La principale est la notion de vue, essentielle pour pouvoir implémenter une part plus importante des algorithmes de GMP qui effectuent de l'arithmétique de pointeurs. Elle peut entre autres être implémentée à l'aide d'une opération `split`, qui coupe virtuellement un tableau mutable en deux sous-tableaux. Cette opération s'est révélée suffisante pour exprimer une part importante des algorithmes de GMP dans WhyML [RH20, MRH23]. L'ajout de cette opération est en cours ; la majeure partie de la preuve de correction de sa compilation a déjà été effectuée.

Une sortie non vérifiée est disponible quand les performances sont jugées plus importantes que les garanties sur le code généré. Dans ce cas, la correction de la compilation pourrait être renforcée en effectuant une preuve de préservation de sémantique jusqu'à LLVM IR, qui a été formalisée en Coq dans le cadre de Vellvm [ZNMZ12]. Pour ce faire, nous pourrions utiliser CompCert SSA [BDP14], puis prouver la traduction restante jusqu'à Vellvm.

Par ailleurs, il serait important de donner plus de contrôle à l'utilisateur concernant la génération des tests dynamiques. En particulier, il devrait pouvoir spécifier quels tests doivent être absents du code généré parce que vérifiés par le compilateur. Une telle optimisation serait ainsi bénéfique aussi bien aux sorties vérifiées que non vérifiées.

Enfin, la sémantique de Capla a été conçue, dès le départ, dans l'optique de la simplification des raisonnements sur les programmes. À cette fin, la sémantique des appels de fonctions se fait par *copy-restore*. Il est ainsi plus simple de comprendre si un tableau est modifié par un appel de fonction ou non et aucun invariant n'est nécessaire pour garantir que les tableaux immuables restent inchangés. Il n'est donc pas nécessaire d'utiliser une logique de séparation pour raisonner sur les programmes, une simple logique à la Hoare suffit. Un calcul de plus faibles préconditions a ainsi été implémenté en Coq, prouvé correct, et utilisé pour vérifier quelques exemples jouets. Du travail reste néanmoins à réaliser pour améliorer l'ergonomie et ainsi pouvoir réaliser des preuves de correction fonctionnelle, à la manière de ce qui est fait avec VST pour le langage C [App11].

Remerciements

L'auteur tient à remercier les relecteurs pour leur commentaires précieux et leurs suggestions utiles à l'amélioration de cet article. Ce projet a reçu un financement du Conseil Européen de la Recherche (ERC) dans le cadre du programme de recherche et d'innovation de l'Union Européenne Horizon 2020 (convention de subventionnement n°101001995).

Références

[App11] Andrew W. Appel : Verified Software Toolchain. *In* Gilles Barthe, éditeur : *20th European Symposium on Programming*, volume 6602 de *Lecture Notes in Computer Science*, pages 1–17, 2011.

[BDL06] Sandrine Blazy, Zaynah Dargaye et Xavier Leroy : Formal verification of a C compiler front-end. *In International Symposium on Formal Methods*, pages 460–475. Springer, 2006.

[BDP14] Gilles Barthe, Delphine Demange et David Pichardie : Formal verification of an ssa-based middle-end for compcert. *ACM Transactions on Programming Languages and Systems*, 36(1), mars 2014.

[CPP23] Jean-Louis Colaço, Baptiste Pauget et Marc Pouzet : Polymorphic types with

polynomial sizes. *In 9th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 36–49, 2023.

[ERHT18] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks et David Tarditi : Checked C : Making C safe by extension. *In 2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60, septembre 2018.

[HE21] Troels Henriksen et Martin Elsman : Towards size-dependent types for array programming. *In* Tze Meng Low et Jeremy Gibbons, éditeurs : *7th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*, pages 1–14, juin 2021.

[JDM+20] Georges-Axel Jaloyan, Claire Dross, Maroua Maalej, Yannick Moy et Andrei Paskevich : Verification of programs with pointers in SPARK. *In* Shang-Wei Lin, Zhe Hou et Brendan Mahony, éditeurs : *22nd International Conference on Formal Engineering Methods*, volume 12531 de *Lecture Notes in Computer Science*, pages 55–72, mars 2020.

[LB08] Xavier Leroy et Sandrine Blazy : Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1):1–31, 2008.

[Ler09a] Xavier Leroy : A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[Ler09b] Xavier Leroy : Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[LLP+22] Liyi Li, Yiyun Liu, Deena Postol, Leonidas Lampropoulos, David Van Horn et Michael Hicks : A formal model of Checked C. *In 35th IEEE Symposium on Computer Security Foundations*, pages 49–63, août 2022.

[MM24] Guillaume Melquiond et Josué Moreau : A safe low-level language for computer algebra and its formally verified compiler. *Proc. ACM Program. Lang.*, 8(ICFP), 2024.

[MRH23] Guillaume Melquiond et Raphaël Rieu-Helft : WhyMP, a formally verified arbitrary-precision integer library. *Journal of Symbolic Computation*, 115:74–95, 2023.

[RH20] Raphaël Rieu-Helft : *Development and verification of arbitrary-precision integer arithmetic libraries.* Theses, Université Paris-Saclay, novembre 2020.

[RLS+19] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi et Michael Hicks : Achieving safety incrementally with Checked C. *In* Flemming Nielson et David Sands, éditeurs : *8th International Conference on Principles of Security and Trust*, volume 11426 de *Lecture Notes in Computer Science*, pages 76–98, avril 2019.

[TG09] Kai Trojahner et Clemens Grelck : Dependently typed array programs don't go wrong. *Journal of Logic and Algebraic Programming*, 78(7):643–664, août 2009.

[XP98] Hongwei Xi et Frank Pfenning : Eliminating array bound checking through dependent types. *ACM SIGPLAN Notices*, 33(5):249–257, mai 1998.

[ZNMZ12] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin et Steve Zdancewic : Formalizing the LLVM intermediate representation for verified program transformations. *ACM SIGPLAN Notices*, 47(1):427–440, janvier 2012.