Compilation formellement vérifiée d'un langage pour le calcul formel

Josué Moreau

▶ **To cite this version:**

Josué Moreau. Compilation formellement vérifiée d'un langage pour le calcul formel. 2022. hal-03824148

HAL Id: hal-03824148
https://inria.hal.science/hal-03824148v1

Preprint submitted on 21 Oct 2022

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.

Compilation formellement vérifiée d'un langage pour le calcul formel

Josué Moreau*†

Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles
F-91190 Gif-sur-Yvette, France

Résumé

Dans cet article, nous présentons une ébauche de langage de bas niveau ayant pour but de pouvoir écrire les algorithmes de base du calcul formel. Les structures de données manipulées par ces algorithmes étant principalement des tableaux, c'est sur ces derniers que notre langage se concentre. Nous présentons sa sémantique qui a été formalisée en Coq. Elle a été conçue dans l'optique d'être sûre, en éliminant certains comportements indéfinis tels que l'accès en dehors des bornes d'un tableau. Un autre objectif de cette sémantique est de simplifier la preuve de programme, par exemple en distinguant tableaux mutables et immuables. Nous nous intéressons enfin à la compilation formellement vérifiée de ce langage vers un langage assembleur. Pour atteindre cet objectif, nous avons écrit un compilateur de notre langage vers le langage intermédiaire Clight de CompCert et nous avons prouvé formellement la préservation de la sémantique lors de cette traduction.

1 Introduction

Le calcul formel s'intéresse aux algorithmes et structures de données pour effectuer des calculs avec des objets mathématiques tels que les polynômes, les matrices, ou les entiers relatifs, d'où le nom de Computer Algebra Systems pour désigner les logiciels tels que Maple, Sage, Maxima, etc. Nombre d'entre eux utilisent des bibliothèques, telles que GMP et BLAS, spécialisées dans la manipulation de certains objets mathématiques. Ces dernières ont pour principal objectif d'être efficaces car elles doivent effectuer un très grand nombre de calculs. Elles sont donc le plus souvent très optimisées, ce qui est aussi source de problèmes : il est difficile d'avoir confiance en ces bibliothèques.

Ce problème découle de trois facteurs. Tout d'abord, ces algorithmes sont subtils et la simple lecture de leur code ne permet pas de se convaincre de leur correction. Par ailleurs, les améliorations ajoutées pour rendre le code encore plus rapide rendent ces programmes très éloignés de leur implémentation naturelles. La vérification de leur correction et de leur sûreté nécessite alors des invariants, lemmes et assertions plus compliqués et en quantité bien plus importante que dans le cas de l'implémentation naturelle. À cette difficulté de vérification s'ajoute parfois une difficulté de compilation, car les compilateurs ne sont pas exempts de bugs et peuvent produire du code incorrect. C'est le cas, par exemple, du compilateur Xcode 11 à 11.3, sous MacOS, qui produit du code faisant des erreurs de calcul pour GMP 6.2.0 [1]. Dans ces conditions, la sûreté et la correction du code sont impossibles à garantir.

Pour palier à ces difficultés, on souhaiterait avoir un langage performant, adapté à l'écriture d'algorithmes du calcul formel, assurant la sûreté des programmes assembleurs générés sans

*Ce projet a reçu un financement du Conseil Européen de la Recherche (ERC) dans le cadre du programme de recherche et innovation Horizon 2020 de l'Union Européenne (convention de subvention N°101001995).

†Je remercie le Labex DigiCosme pour la bourse qui m'a été attribuée en master et qui m'a permis d'étudier dans les meilleures conditions.

1. https://gmplib.org/#STATUS

avoir à les vérifier, ainsi qu'un compilateur formellement vérifié pour ce langage. Plusieurs environnements de programmation assurent partiellement ces objectifs. C'est le cas du langage StandardML, qui est sûr, avec CakeML [6], un compilateur formellement vérifié. Cependant, ce dernier n'est pas particulièrement adapté au calcul formel. Rust assure aussi la sûreté et présente certaines constructions intéressantes pour les algorithmes qui nous intéressent, mais son compilateur n'est pas formellement vérifié, il n'est donc pas garanti que le programme compilé soit sûr. Dans un autre registre, VST [3] est un compilateur formellement vérifié qui produit du code sûr, mais uniquement à condition que l'utilisateur ait prouvé la sûreté pour chacun de ses programmes. On constate donc qu'aucune combinaison langage/compilateur ne semble couvrir tous ces objectifs.

Dans cet article, nous présentons un langage et son compilateur. Le langage permet de manipuler des tableaux et, par conséquent, d'écrire des algorithmes de base du calcul formel. Les principaux objectifs de ce langage sont de posséder une sémantique garantissant la sûreté du code assembleur généré, ainsi que des constructions simplifiant au maximum la preuve de correction des programmes. Ces deux objectifs ont ainsi influencé certains choix dans la conception du langage. Cependant, étant donné que notre langage n'est actuellement qu'une ébauche et qu'il sera enrichi ultérieurement, nous ne nous intéressons pas ici à prouver formellement sa sûreté mais seulement à le compiler de manière sûre. Le compilateur que nous présentons pour ce langage utilise le compilateur CompCert, en compilant vers son langage intermédiaire Clight [2]. La sémantique de notre langage a été formalisée avec l'assistant de preuve Coq, de même que la preuve de préservation de la sémantique jusqu'à Clight. Nous obtenons donc, en composant notre résultat et celui de CompCert, un compilateur formellement vérifié pour notre langage.

Dans un premier temps, nous présentons notre langage, en section 2, ainsi que les grandes lignes de sa sémantique, en section 3. Nous nous intéressons ensuite, en section 4, à la compilation de ce langage vers Clight. Enfin, la section 5 présente la preuve formelle de préservation de la sémantique pendant la compilation jusqu'à Clight.

2 Langage

Le langage que nous souhaitons concevoir doit être capable de manipuler des tableaux car ils sont à la base des bibliothèques telles que GMP et BLAS. Étant donné que l'un des objectifs finaux de ce langage est d'être sûr, il est nécessaire de vérifier que tous les accès aux tableaux se font bien dans leurs bornes. On a donc pour chaque tableau, en plus du pointeur le représentant, une variable supplémentaire contenant sa taille.

Par ailleurs, afin de simplifier la preuve de correction des programmes, nous faisons le choix d'interdire l'aliasing entre deux tableaux. Certaines constructions du langage, telles que l'affectation ou l'appel de fonction, seront alors restreintes. L'interdiction des alias étant tout de même très forte, et éliminant trop d'algorithmes intéressants en calcul formel, on distingue tableaux mutables et immuables et on autorise que deux tableaux immuables soient aliasés. Cette distinction permet également de supprimer bon nombre d'invariants, ce qui répond aussi à l'objectif de simplification de la preuve de programmes.

Les fonctions et les appels de fonctions sont similaires à ceux d'un langage tel que C, et les fonctions sont mutuellement récursives. Quelques modifications ont cependant été faites. La première est que les tailles de tableau sont passées explicitement en argument. La spécification d'une fonction lie ensuite chaque tableau en argument à l'argument contenant sa taille. On autorise, de plus, qu'une variable de taille passée en argument soit liée à plusieurs tableaux. L'implémentation correspond alors exactement à la signature d'opérations comme l'addition de

vecteurs qui, mathématiquement, prend toujours deux vecteurs de même taille n et renvoie un vecteur de taille n. Il s'agit ici, en quelque sorte, d'un système simpliste de types dépendants.

On peut voir ci-dessous deux implémentations de l'addition de vecteurs, l'une en Rust et l'autre dans notre langage. Dans les deux langages, une vérification est théoriquement effectuée à chaque accès aux tableaux `dest`, `a` et `b`. Cependant, dans ce cas précis, contrairement à Rust où on ne pourrait éliminer que les tests sur `dest`, tous les tests pourraient être supprimés dans notre langage car il garantit que tous les tableaux sont de taille au moins n. Pour ce faire, une exception sera levée au moment de l'appel de la fonction si l'appelant transmet une taille trop grande en paramètre n. Une conséquence de l'utilisation d'une même variable de taille pour plusieurs tableaux est donc de déplacer l'endroit où l'exception peut être levée, ce qui permet d'éviter bon nombre de tests lors des accès aux tableaux.

```rust
fn add_vectors(a: &[i64], b: &[i64], dest: &mut [i64]) {
    for i in 0..dest.len() {
        dest[i] = a[i] + b[i]
    }
}
```

RUST

```
void add_vectors(i64[n] a, i64[n] b, mut i64[n] dest, i64 n) {
    for i <- 0 to n - 1 {
        dest[i] <- a[i] + b[i]
    }
}
```

Notre Langage

3 Sémantique

Nous décrivons maintenant la sémantique de notre langage. Celle-ci a été conçue et formalisée avec Coq et s'inspire fortement de la sémantique à petits pas de Clight [2], telle qu'elle est définie dans le développement Coq de CompCert [2]. De nombreuses prémisses ont, cependant, été ajoutées à des fins de sûreté.

Dans la sémantique de notre langage, nous distinguons erreurs et blocages. En particulier, pour toutes les erreurs impossibles à détecter statiquement, par exemple les accès en dehors des bornes d'un tableau, il est nécessaire de construire des règles pour lever une erreur. Concrètement, nous ajoutons une manière de spécifier que l'évaluation d'une expression a mené à une erreur et nous ajoutons également une instruction `error` qui propage ensuite cette erreur dans la sémantique des instructions. Il s'agit d'une différence importante avec Clight et C, où aucun mécanisme d'erreur n'est présent et où, lors d'un accès en dehors des bornes d'un tableau, la sémantique bloquera, c'est-à-dire qu'il s'agira d'un comportement indéfini.

Mémoire des tableaux et valeurs

Le modèle mémoire de notre langage est constitué de deux parties : un environnement local E et une mémoire des tableaux M. Cette dernière, une version simplifiée de la mémoire de CompCert, associe à chaque identifiant de tableau une liste de valeurs, qui représente son contenu. Quant à l'environnement local, il associe une valeur à chaque identifiant de variable, manipulé par la fonction. Notre environnement local est similaire à celui de Cminor et au

2. The CompCert Coq development, https://compcert.org/doc/index.html

`temp_env` de Clight, et donc différent de l'environnement local E de Clight. Pour ce qui est du lien entre tableau et variable de taille, il se trouve actuellement dans la représentation des fonctions car celles-ci ne créent pas de tableaux temporaires. Cependant, il sera plus tard intégré dans l'environnement local.

Le lien entre les deux éléments de notre modèle mémoire est effectué par les valeurs de tableau `Varr` id $\vec{v}$ associées dans E aux variables contenant des tableaux. Dans cette valeur, id est l'identifiant de tableau associé à la variable et $M[id]$ contient les valeurs du tableau. Ces mêmes valeurs sont également copiées dans $\vec{v}$. Il sera alors nécessaire que la sémantique garde toujours une synchronisation entre ces deux environnements, afin que $M[id]$ et $\vec{v}$ soient toujours égaux lors de l'accès au tableau id. L'utilité de cette copie peut être observée dans le code $t[i] \leftarrow x; s; y \leftarrow t[i]$. Si l'instruction s n'écrit pas dans t, alors on déduit immédiatement que l'égalité $x = y$ est vérifiée. On peut donc savoir, uniquement en regardant le code, que le tableau n'a pas changé, ce qui peut faciliter la preuve d'algorithmes.

Sémantique des instructions

Dans la sémantique des instructions, nous utilisons la même représentation que CompCert pour les états de la sémantique, il y a donc trois types d'états. Premièrement, les états $\mathcal{C}(M, F, \vec{v}, k)$ d'appel de la fonction F avec les valeurs $\vec{v}$ en argument. Deuxièmement, les états $\mathcal{S}(M, E, F, s, k)$ d'exécution de l'instruction s dans le corps d'une fonction F. Troisièmement, les états $\mathcal{R}(M, v, k)$ de retour à une fonction appelante, en renvoyant la valeur v. Dans ces trois types d'états, M et E sont respectivement la mémoire et l'environnement, décrits précédemment. La continuation k représente la suite de l'exécution du programme. Le type des continuations est constitué de quatre constructeurs. La fin de l'exécution du programme est représentée par `stop`, `seq`(s, k) exécute l'instruction s puis la continuation k et `loop`(s, k) boucle sur l'instruction s avant d'exécuter la continuation k. Enfin, la continuation `returnto`$(id^?, \vec{l}, E, F, k)$ retourne à la fonction F, dont l'environnement local était E, puis met à jour les tableaux mutables indiqués dans $\vec{l}$ et écrit la valeur de retour dans la variable id de l'environnement E.

Un échantillon des règles de la sémantique est donné en figure 1, les autres règles correspondent à des cas d'erreurs ou sont similaires à celles de Clight. G est l'environnement global qui contient les définitions de fonction. La notation $M, E, F \vdash e \not\Rightarrow$ énonce que e s'évalue vers une erreur.

La règle WriteArr contient de nombreuses prémisses intéressantes. Les prémisses de la troisième ligne obligent d'écrire à un indice valide du tableau. La deuxième ligne énonce la synchronisation, évoquée précédemment, entre environnement local et mémoire. Quant à la quatrième ligne, elle décrit comment ces mêmes environnement et mémoire sont mis à jour. Un certain nombre de règles, non décrites ici, traitent les cas où une erreur dynamique doit être levée lorsque certaines prémisses de cette règle ne sont pas respectées.

Dans la règle Call, le prédicat `valid_call` vérifie que les valeurs des tailles de tableau fournies en argument sont bien inférieures ou égales à celles des tableaux fournis en argument. La règle Call passe ensuite dans la continuation la liste $\vec{l}$ des identifiants des tableaux passés en argument mutable. Les valeurs associées aux identifiants de $\vec{l}$ dans l'environnement local de la fonction appelée seront ensuite mises à jour, via la règle RState, avec les nouvelles listes de valeurs contenues dans M (après exécution de la fonction appelée) pour resynchroniser l'environnement local et la mémoire, modifiée par la fonction appelée.

$$\textsc{WriteVar}\ \frac{M, E, F \vdash e \Rightarrow v}{G \vdash \mathcal{S}(M, E, F, id \leftarrow e, k) \rightarrow \mathcal{S}(M, E[id \mapsto v], F, \texttt{skip}, k)}$$

$$\textsc{WriteVarErr}\ \frac{M, E, F \vdash e \not\Rightarrow}{G \vdash \mathcal{S}(M, E, F, id \leftarrow e, k) \rightarrow \mathcal{S}(M, E, F, \texttt{error}, k)}$$

$$\textsc{WriteArr}\ \frac{\begin{array}{c} F.\texttt{tenv}(id) = \tau\ \texttt{arr}_{\texttt{mut}} \quad M, E, F \vdash e \Rightarrow v \quad v \in \tau \quad (id, id_{sz}) \in F.\texttt{arrszvar} \\ E[id] = \texttt{Varr}\ \ id_{arr}\ \ \vec{v} \quad M[id_{arr}].\texttt{type} = \tau \quad M[id_{arr}].\texttt{values} = \vec{v} \\ M, E, F \vdash idx \Rightarrow \texttt{Vint}_{64}\ \texttt{Unsigned}\ i \quad M, E, F \vdash id_{sz} \Rightarrow \texttt{Vint}_{64}\ \texttt{Unsigned}\ n \quad i < n \\ M' = M[id_{arr} \mapsto \vec{v}[i \mapsto v]] \quad E' = E[id \mapsto \texttt{Varr}\ \ id_{arr}\ \vec{v}[i \mapsto v]] \end{array}}{G \vdash \mathcal{S}(M, E, F, id[idx] \leftarrow e, k) \rightarrow \mathcal{S}(M', E', F, \texttt{skip}, k)}$$

$$\textsc{Call}\ \frac{\begin{array}{c} G[id_F] = F' \quad \texttt{valid\_call}(M, E, F, F', \vec{a}) \\ G, F \vdash \vec{a} : \vec{\tau} \quad F.\texttt{sig.args} \preceq \vec{\tau} \quad M, E, F \vdash \vec{a} \Rightarrow \vec{v} \\ \forall i, (\exists \sigma, \vec{\tau}[i] = \sigma\ \texttt{arr}_{\top}) \iff \vec{a}[i] \in \vec{l} \end{array}}{G \vdash \mathcal{S}(M, E, F, id^{?} \leftarrow id_F(\vec{a}), k) \rightarrow \mathcal{C}(M, F', \vec{v}, \texttt{returnto}(id, \vec{l}, E, F, k))}$$

$$\textsc{RState}\ \frac{}{G \vdash \mathcal{R}(M, v, \texttt{returnto}(id^{?}, \vec{l}, E, F, k)) \rightarrow \mathcal{S}(M, \texttt{update}(E, M, \vec{l})[id^{?} \mapsto v], F, \texttt{skip}, k)}$$

FIGURE 1 – Sémantique à petit pas des instructions

4 Compilation

La traduction de notre langage à Clight a été réalisée en deux passes. La première a pour but de traduire notre langage vers un langage qui est exactement le même, à l'exception de sa sémantique qui correspond à celle de notre langage sans les règles générant des erreurs. Il ne sera donc plus possible, dans cette nouvelle sémantique, de lever une erreur automatiquement, seule l'instruction `error` peut en lever une. Dans la suite, nous parlerons de sémantique bloquante pour désigner cette nouvelle sémantique et de sémantique non bloquante pour désigner la sémantique présentée en section 3.

Le but de la première passe est de placer des tests dynamiques avant les instructions pour lesquelles on ne peut pas vérifier statiquement toutes les prémisses. Cela permet alors d'éviter les blocages et de lever une erreur lorsque l'assertion liée à la prémisse n'est pas respectée. Une assertion est ajoutée à chaque accès à un tableau, pour vérifier que l'indice auquel on accède est bien valide. Une assertion est également ajoutée à chaque appel de fonction, pour vérifier que les tailles passées en argument sont inférieures ou égales à celles des tableaux associés.

La seconde passe est ensuite la traduction du langage intermédiaire à sémantique bloquante vers Clight. Il s'agit en fait de l'identité car les constructions du langage à sémantique bloquante sont identiques à celles de Clight. Cela permet de se concentrer, pendant la preuve de préservation de sémantique, sur la correspondance entre le modèle mémoire de notre langage et celui de CompCert.

Le code assembleur x86-64 généré par la compilation de la fonction `add_vectors` est donné ci-dessous. Plusieurs choses sont à remarquer. La première correspond au point `.L102`, qui correspond à la traduction de l'erreur. On peut voir qu'il s'agit d'une boucle infinie qui est la sémantique que nous avons, pour l'instant, choisi de donner à l'instruction `error`. Nous utiliserons, plus tard, les fonctions externes de CompCert pour effectuer un appel à une fonction, telle que `abort` ou `longjmp`, mettant un terme à l'exécution du programme. La seconde chose à remarquer est le test avant l'accès aux tableaux `a`, `b` et `c`, qui garantit la sûreté. Il n'y a qu'un seul test d'accès dans les bornes du tableau, au lieu de trois s'il y en avait eu un pour chaque

accès dans l'instruction `a[i]` ← `b[i]` + `c[i]`. Il s'agit ici d'une légère optimisation qui a été ajoutée pour éviter certains doublons dans les assertions générées. On remarque tout de même que ce test est précédé d'un test identique correspondant à la condition de fin de boucle. Il reste donc une optimisation à faire pour détecter que le test d'accès est inutile et le retirer car il est identique à la condition de fin de boucle.

```
add_vectors: ; %rdi = a; %rsi = b, %rdx = dest, %rcx = n
  ...
  xorq  %r9, %r9              ; %r9 = i <- 0
.L100:                        ; boucle
  cmpq  %rcx, %r9
  jae .L101                   ; i >= n => fin de la boucle
  cmpq  %rcx, %r9             ; test d'accès dans les bornes
  jae .L102                   ; i >= n => erreur
  movq  0(%rdi,%r9,8), %r10   ; %r10 <- a[i]
  movq  0(%rsi,%r9,8), %r11   ; %r11 <- b[i]
  leaq  0(%r10,%r11,1), %r10  ; %r10 <- %r10 + %r11 = a[i] + b[i]
  movq  %r10, 0(%rdx,%r9,8)   ; dest[i] <- %r10 = a[i] + b[i]
  leaq  1(%r9), %r9           ; i <- i + 1
  jmp .L100
.L102:          ; traduction de error
  jmp .L102     ; (on traduit error par une boucle infinie)
.L101:
  ...
  ret
```

5 Préservation de la sémantique

Nous nous intéressons maintenant à prouver que la sémantique est préservée lors de la traduction de notre langage vers Clight. On obtient alors un compilateur formellement vérifié pour notre langage, en composant nos résultats et ceux de CompCert. La formalisation de la sémantique de notre langage et la preuve que nous présentons ici ont été réalisées avec Coq, en utilisant un certain nombre d'éléments provenant de la formalisation des langages de CompCert. L'ensemble de la preuve Coq est disponible ici : https://github.com/josuemoreau/stage-M2-JFLA.

Tout comme la traduction a été faite en deux passes, la préservation de la sémantique a été prouvée en deux parties, une pour chaque passe de la compilation. De la même manière que CompCert [7, 8], nos deux preuves ont nécessité d'établir une certaine correspondance entre les états des sémantiques de départ et d'arrivée. Elles ont ensuite été faites par simulation avant, en utilisant le cadre formel fourni par CompCert. Le théorème Coq pour la préservation de la sémantique lors de la première passe est, par exemple, donné ci-dessous. La fonction `transl_program` est la fonction de traduction de notre langage à sémantique non bloquante vers le langage à sémantique bloquante.

```
Theorem transl_program_correct (p: program) (tp: program):
  transl_program p = OK tp →
  forward_simulation (SemanticsNonBlocking.semantics p)
                     (SemanticsBlocking.semantics tp).
```

La correspondance entre les états de la sémantique non bloquante et ceux de la sémantique bloquante est l'identité pour ce qui est du modèle mémoire. La plus grande partie de la preuve de la première passe a donc consisté à montrer que les tests qui sont générés capturent exactement les cas où une erreur était levée dans la sémantique non bloquante.

En ce qui concerne la deuxième passe, la traduction des instructions est quasiment l'identité de notre langage à CompCert. La majeure partie de la preuve consiste donc à faire le lien entre notre modèle mémoire et celui de CompCert et prouver que ce lien est préservé par chaque instruction. La correspondance entre les états de notre sémantique bloquante et ceux de Clight est représentée par le prédicat inductif `match_states`, donné partiellement ci-dessous.

```
Inductive match_states (p: program) : state → Clight.state → Prop :=
| match_states_State:
    ...
| match_states_Callstate:
    ...
| match_states_Returnstate: ∀ tp trarr m r k tm tk
    (VTRARR  : valid_trarr trarr)
    (TK      : transl_cont trarr (genv_of_program p) None k = Some tk)
    (MMEM    : match_mem tp trarr m tm),
    match_states p (Returnstate m r k)
                   (Clight.Returnstate (transl_value trarr r) tk tm).
```

Nous ne montrons ici que le troisième cas de correspondance, qui est celui qui concerne les états de retour de fonction. Dans celui-ci, la fonction *trarr* traduit les identifiants de la mémoire des tableaux en associant à chacun d'entre eux un bloc et une position dans la mémoire de CompCert. Le prédicat `valid_trarr` de l'hypothèse `VTRARR` énonce que deux tableaux ayant des identifiants différents dans la mémoire de notre sémantique sont placés dans deux blocs différents de la mémoire de Clight. L'hypothèse `TK` demande ensuite que la fonction `transl_cont` de traduction des continuations de la sémantique bloquante à celles de Clight s'exécute sans erreur sur la continuation *k*.

Finalement, l'hypothèse `MMEM` énonce une correspondance entre la mémoire de notre sémantique bloquante et celle de la sémantique de Clight. La définition simplifiée du prédicat `match_mem` est donnée en figure 2. Elle montre trois conditions sur la correspondance des mémoires. La première est que les positions correspondant aux indices des tableaux de la mémoire de notre langage sont représentables sans dépassement par les entiers de CompCert. La seconde énonce qu'il est possible d'écrire dans tous les emplacements de la mémoire de CompCert correspondants à des tableaux de notre mémoire. Enfin, la troisième dit que lire une valeur dans la mémoire de CompCert est équivalent à lire la traduction de la valeur correspondante dans notre mémoire.

6 Conclusion et travaux futurs

Nous avons présenté un langage typé, avec tableaux, qui a vocation à être sûr. Grâce à l'utilisation de CompCert, nous avons, pour ce langage, un compilateur formellement vérifié produisant du code assembleur dans différentes architectures (celles prises en charge par CompCert). Ce code peut ensuite être lié à des fichiers objets écrits dans d'autres langages. La formalisation de nos sémantiques et la preuve de préservation de sémantique ont nécessité respectivement 1700 et 3800 lignes de Coq.

```
Definition match_mem (tp: Clight.program) (trarr: ident → (block * ptrofs))
                     (m: mem) (tm: Memory.mem) : Prop :=
  Forall (fun (x: (ident * memory_block)) ⇒
    let (idarr, blk) := x in
    let (b, ofs) := trarr idarr in
    let chunk := typ_to_memory_chunk (blk.(blk_type)) in
     ∀ (idx: nat),
           (idx < length blk.(blk_values))%nat →
           (Z.of_nat idx <= Int64.max_unsigned) →
    (* 1 *) ofs + sizeof ... * idx <= Ptrofs.max_unsigned
    (* 2 *) ∧ Mem.valid_access tm chunk b (ofs + sizeof ... * idx) Writable
    (* 3 *) ∧ Mem.loadv chunk tm (Vptr b (ofs + sizeof ... * idx))
              = option_map (transl_value trarr) (nth_error blk.(blk_values) idx))
                           (PTree.elements m).
```

FIGURE 2 – Correspondance entre la mémoire de notre langage et celle de CompCert

Plusieurs optimisations peuvent être faites dans le compilateur concernant l'élimination des tests dynamiques générés. Il serait, en particulier, intéressant de supprimer le plus possible, voir totalement dans un cas comme `add_vectors`, ces tests dynamiques pour ne pas trop influer sur les performances du code généré. Une telle phase d'optimisation pourrait utiliser des solveurs de programmation linéaire et SMT pour détecter et éliminer le plus possible ces tests.

Par la suite, le langage que nous avons présenté n'est qu'un petit langage et de nombreuses constructions pratiques seraient à ajouter. C'est le cas de l'extraction et du découpage de tableaux. Ces deux constructions vont nécessiter davantage de réflexion sur la sémantique qu'il faudra leur donner, concernant la création et la destruction des sous-tableaux. Il faut, en effet, réfléchir à la représentation du lien entre tableau original et sous-tableaux de sorte que, à la destruction des sous-tableaux, le tableau original ait bien été modifié. Une idée pour représenter cela pourrait être d'utiliser les variables de prophétie [1] à la manière de ce qui a été fait pour la logique de séparation [5]. Un autre point important sur la destruction des tableaux est l'emplacement de la destruction. Nous souhaiterions que l'utilisateur n'ait pas besoin de spécifier l'endroit où les sous-tableaux doivent être détruits, à la manière de ce qui a été fait en Rust [4].

Une fois toutes ces constructions ajoutées, il sera nécessaire de prouver formellement que le code généré par notre compilateur est toujours sûr. Nous nous intéresserons alors à montrer que le typage permet bien de garantir l'absence d'alias ; ou encore que la sémantique garantit que l'environnement et la mémoire sont toujours synchronisés pendant l'exécution d'un programme bien typé. Nous montrerons ensuite, toujours dans le cas de ces programmes bien typés, le progrès de notre sémantique.

Enfin, pour passer de programmes sûrs à programmes sûrs et corrects, nous souhaitons ajouter une logique de programmes, qui devra être la moins intrusive possible, à notre langage. Par ailleurs, les algorithmes des briques élémentaires du calcul formel ont souvent des formes communes : ils utilisent des tableaux qu'ils manipulent souvent avec des boucles imbriquées qui comportent des conditions simples. Cela implique que les algorithmes n'auront pas n'importe quelle forme et il serait alors intéressant d'utiliser ces particularités afin de développer des procédures de décision dédiées pour automatiser au maximum leurs preuves de correction.

Références

[1] M. Abadi and L. Lamport. The existence of refinement mappings. In *3rd Symposium on Logic in Computer Science*, pages 165–175, 1988.

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

[3] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W Appel. VST-Floyd : A separation logic tool to verify correctness of C programs. *Journal of Automated Reasoning*, 61(1):367–422, 2018.

[4] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt : Securing the foundations of the Rust programming language. In *45th ACM Symposium on Principles of Programming Languages*, number POPL in 2, pages 1–34, 2018.

[5] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. The Future is Ours : Prophecy Variables in Separation Logic. In *47th ACM Symposium on Principles of Programming Languages*, volume 4, pages 1–32, 2020.

[6] Ramana Kumar, Magnus O Myreen, Michael Norrish, and Scott Owens. CakeML : a verified implementation of ML. In *41st ACM Symposium on Principles of Programming Languages*, volume 1, pages 179–191, 2014.

[7] Xavier Leroy. Formal certification of a compiler back-end or : programming a compiler with a proof assistant. In *33rd ACM Symposium on Principles of Programming Languages*, pages 42–54, 2006.

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