Un mécanisme d'extraction vers C pour Why3

Raphaël Rieu-Helft

Un mécanisme d'extraction vers C pour Why3

Raphaël Rieu-Helft[1,2]

[1] TrustInSoft
[2] Inria, Université Paris-Saclay, F-91120 Palaiseau

Résumé

Nous présentons un mécanisme d'extraction d'un sous-ensemble des programmes écrits avec l'outil de vérification Why3 vers le langage C. Un modèle mémoire partiel mais simple permet aux utilisateurs d'écrire des programmes Why3 dans un style impératif très proche du C, avec une gestion manuelle et vérifiée formellement de la mémoire. De tels programmes peuvent ensuite être extraits de manière transparente vers du code C idiomatique, ce qui évite d'introduire des pertes de performances à l'extraction.

1 Introduction

L'objectif de ce travail est d'obtenir des programmes C à la fois efficaces et vérifiés formellement. L'idée principale est d'écrire le code dans un langage de haut niveau, à savoir le langage de programmation WhyML [6] fourni par la plateforme de vérification Why3, et de l'extraire vers C. WhyML est un dialecte de ML visant à faciliter la preuve automatique, notamment en faisant en sorte que tous les alias soient connus statiquement par typage [4]. WhyML est dédié à la vérification statique de comportements fonctionnels décrits à l'aide d'annotations fournies par l'utilisateur (assertions intermédiaires [8], code *ghost* [5]...) dans un langage de spécification expressif [2, 3]. Il s'agit ensuite de convertir un tel programme de haut niveau en code exécutable et performant. Le mécanisme d'*extraction* de Why3 traduit le code WhyML vers un langage de programmation classique en oubliant les spécifications et annotations destinées à la vérification. Why3 supporte nativement l'extraction vers OCaml, et nous y avons ajouté une extraction vers C. Pour obtenir du code C qui inclut des accès mémoire bas niveau à l'aide de pointeurs, il est nécessaire de mettre au point un modèle des pointeurs et du tas C en Why3, dans lequel les alias potentiels entre pointeurs sont contrôlés (Section 2). Nous avons conçu ce modèle mémoire pour permettre une compilation directe et transparente de WhyML vers C (Section 3). Nous présentons un exemple en Section 4.

Ce travail a déjà été publié dans le contexte du développement d'une bibliothèque d'arithmétique de grands entiers efficace et vérifiée, extraite de Why3 vers C. [9].

2 Modèle mémoire

Nous avons ajouté à la bibliothèque standard de Why3 un modèle mémoire du C dans lequel les opérations de base sur les pointeurs sont axiomatisées (figure 1). Au moment de l'extraction, ces opérations peuvent être simplement remplacées par leurs équivalents C (en commentaires sur la figure). Notre modèle est plus restrictif que le standard C, afin de simplifier le modèle et les preuves. Par exemple, l'arithmétique de pointeur est réduite à l'addition d'un pointeur et d'un entier, tandis que les soustractions et inégalités entre pointeurs ne sont pas permises. Le modèle ne prévoit pas non plus de cast d'entier vers pointeur, et l'opérateur addresse `&` n'est pas présent.

La raison principale pour ces restrictions est le problème de l'aliasing entre pointeurs. Le tas mémoire C est vu comme un ensemble de blocs mémoire appelés *objets* dans le standard

```
1  type ptr 'a = { mutable data : array 'a ; offset : int }
2
3  function plength (p:ptr 'a) : int = p.data.length
4
5  function pelts (p:ptr 'a) : (int → 'a) = p.data.elts
6
7  val malloc (sz:uint32) : ptr 'a              (*  malloc(sz * sizeof('a))   *)
8    requires { sz > 0 }
9    ensures  { plength result = sz ∨ plength result = 0 }
10   ensures  { result.offset = 0 }
11
12 val free (p:ptr 'a) : unit                              (*         free(p)    *)
13   requires { p.offset = 0 }
14   writes   { p.data }
15   ensures  { plength p = 0 }
16
17 predicate valid (p:ptr 'a) (sz:int) =
18   0 ≤ sz ∧ 0 ≤ p.offset ∧ p.offset + sz ≤ plength p
19
20 val get (p:ptr 'a) : 'a                                 (*              *p    *)
21   requires { 0 ≤ p.offset < plength p }
22   ensures  { result = p.data[p.offset] }
23
24 val set (p:ptr 'a) (v:'a) : unit                         (*          *p = v    *)
25   requires { 0 ≤ p.offset < plength p }
26   writes   { p.data.elts }
27   ensures  { pelts p = Map.set (pelts (old p)) p.offset v }
28
29 val incr (p:ptr 'a) (ofs:int32) : ptr 'a                 (*           p+ofs    *)
30   requires { p.offset + ofs ≤ plength p }
31   alias    { p.data ~ result.data }
32   ensures  { result.offset = p.offset + ofs }
33   ensures  { result.data = p.data }
34
35 val get_ofs (p:ptr 'a) (ofs:int32) : 'a                  (*         *(p+ofs)   *)
36   requires { 0 ≤ p.offset + ofs < plength p }
37   ensures  { result = p.data[p.offset + ofs] }
38
39 val set_ofs (p:ptr 'a) (ofs:int32) (v:'a) : unit         (*     *(p+ofs) = v   *)
40   requires { 0 ≤ p.offset + Int32.to_int ofs < plength p }
41   ensures { pelts p = Map.set (pelts (old p)) (p.offset + Int32.to_int ofs) v }
42   writes { p.data.elts }
```

FIGURE 1 – Un modèle des pointeurs et du tas mémoire du C.

C99 [7]. Deux pointeurs sont dits *alias* l'un de l'autre lorsqu'ils pointent vers le même objet mémoire (y compris avec des décalages différents), c'est-à-dire qu'une écriture via l'un affecte les valeurs lues via l'autre. La connaissance des relations d'aliasing entre pointeurs est cruciale pour la preuve. Notre modèle restreint permet de profiter du système de types de Why3 [4] pour connaître statiquement les alias présents entre pointeurs. Un modèle plus général du tas C ne permettrait pas de s'appuyer sur le système de types. Il faudrait alors exprimer de nombreuses hypothèses d'aliasing ou de non-aliasing entre pointeurs dans les préconditions de fonctions, ce qui compliquerait énormément le travail de preuve même pour des programmes simples.

Le type polymorphe WhyML `ptr 'a` (figure 1 ligne 1) représente les pointeurs vers des blocs contenant des données de type `'a`. Le champ `data` d'un pointeur est un tableau contenant les données du bloc, et le champ `offset` indique vers quelle case du tableau il pointe. Cette

construction supporte les alias entre pointeurs : différents pointeurs peuvent référencer le même tableau (donc pointer vers le même bloc en mémoire). Grâce au système de types de Why3, les alias sont connus statiquement et une assignation à travers un pointeur est propagée à ses alias.

Les pointeurs sont alloués avec la fonction `malloc`. En cas d'échec, celle-ci renvoie un pointeur *invalide*, représenté par un bloc de longueur 0. La fonction `free` invalide son argument en remplaçant la longueur du bloc pointé par 0. Un pointeur est dit *valide pour une taille* t (ligne 17) lorsque son offset plus t n'excède pas la longueur du bloc pointé. La fonction `get` (ligne 20) représente le déréferencement de pointeur en lecture. La fonction `set` représente l'assignation en mémoire ; la clause `writes` spécifie l'effet d'écriture sur le bloc.

La fonction `incr` (ligne 29) renvoie la somme d'un pointeur et d'un entier. Conformément au standard C [7, Section 6.5.6, "Additive Operators"], on peut seulement calculer un pointeur vers l'intérieur d'un bloc valide ou vers l'élément juste après un bloc valide, ce qui est reflété dans la précondition.

Le mot clé `alias` (que nous avons ajouté à Why3 dans le cadre de ce travail) dans la signature de `incr` déclare que le pointeur renvoyé par `incr` est un alias du pointeur passé en argument. Plus précisément, il unifie les régions `p.data` et `result.data`. Ceci est vrai à la fois du point de vue du contenu du bloc pointé, et du point de vue du comportement de `free`. Ceci permet d'écrire une spécification particulièrement courte pour `free` : son effet `writes` sur `p.data` y induit un effet dit de *reset* [4], ce qui signifie que la région précédemment pointée n'est plus accessible par aucun de ses alias, qui sont donc invalidés.

3 Extraction vers du C idiomatique

L'objectif de la procédure d'extraction est de générer du code C efficace. Certaines fonctionnalités du langage WhyML, comme les types algébriques ou les fonctions d'ordre supérieur, sont difficiles à traduire en C sans introduire de constructions complexes (clôtures, allocation de mémoire automatique, ramasse-miettes...) qui nuiraient aux performances du code extrait. Nous avons donc décidé de ne supporter qu'un sous-ensemble de WhyML. Il ne s'agit donc pas d'extraire des programmes WhyML arbitraires vers C, mais d'extraire des programmes WhyML écrits dans un style proche du C. Nous permettons donc à notre procédure d'extraction d'échouer, ce qui arrive quand le programme à extraire contient des fonctionnalités WhyML non supportées. Les fonctionnalités de WhyML supportées sont celles qui peuvent être traduites vers C de manière transparente, comme les boucles et les références. En abandonnant un certain nombre de fonctionnalités du langage, on gagne une grande simplicité dans la procédure d'extraction, et le code extrait ressemble au code WhyML d'origine. Le programmeur WhyML peut donc prédire à quoi ressemblera le programme C extrait, et peut donc obtenir plus facilement du code C efficace. Cette transparence permet aussi de faire davantage confiance au code extrait et à la procédure d'extraction, qui n'est pas vérifiée formellement.

Compilation d'exceptions en instructions `break` ou `return`. Si l'extraction n'a pas vocation à supporter toutes les fonctionnalités de WhyML, il est en revanche souhaitable qu'un maximum de fonctionnalités du C soient exprimables dans le sous-ensemble de WhyML traité. En particulier, on veut pouvoir obtenir des programmes C comportant des instructions `break` ou `return`. Dans WhyML, ces structures de contrôle sont exprimées à l'aide d'exceptions. On cherche donc à détecter les exceptions WhyML qui peuvent être extraites directement vers `break` ou `return` en C. Les autres exceptions sont rejetées.

Dans le cas de `break`, on détecte le motif suivant et extraire toutes les instances de `raise B` dans le corps de la boucle (mais pas dans d'éventuelles boucles imbriquées) par `break`.

```
try while ... do ... raise B ... done with B → () end
```

Similairement, pour `return`, on détecte le motif suivant dans les définitions de fonctions et on extrait toutes les instances de `raise (R e)` par `return e`. À noter que la construction `try with` doit être en position terminale dans le corps de la fonction.

```
let f (args) = ... ; try ... raise (R e) ... with R v → v end
```

Notre procédure d'extraction reconnaît ces motifs indépendamment des noms des exceptions utilisées. Toutes les instances de `try with` ou `raise` qui ne correspondent pas à l'un des motifs sont rejetées par l'extraction.

Valeurs de retour multiples. De nombreuses fonctions WhyML renvoient plusieurs valeurs sous forme d'un n-uplet. Ceci n'a pas d'équivalent natif en C. Nous avons choisi d'extraire chaque fonction renvoyant un n-uplet en une fonction C renvoyant `void` et prenant comme arguments supplémentaires un pointeur par élément du n-uplet. On peut ensuite détecter l'appel

```
let (x1, x2, ...) = f(args) in ...
```

et l'extraire comme

```
f(&x1, &x2, ..., args); ...
```

La fonction `add_with_carry` (figure 3) est un exemple d'appel d'une telle fonction.

4 Un exemple : addition de grands entiers

Le type Why3 `uint64` (défini en figure 2) est un type abstrait, équipé d'une projection `to_int` qui associe à un entier machine sa valeur mathématique. À l'extraction, ils sont traduits directement vers le type C `uint64_t`. Les préconditions des opérations arithmétiques élémentaires sur ces entiers (comme `add` sur la figure) maintiennent l'invariant que cette valeur est comprise entre 0 et $2^{64} - 1$. Un entier de taille arbitraire est représenté par un tableau d'`uint64`, et sa valeur mathématique est donnée par la fonction logique `value`.

La figure 3 contient une fonction WhyML tirée de notre bibliothèque de calcul en précision arbitraire [9]. Elle effectue l'addition de deux entiers de taille arbitraire. Par souci de concision, les annotations de preuve ont été omises.

L'algorithme est celui qui est enseigné à l'école : on ajoute les "chiffres" (ici en base 2^{64}) les moins significatifs des deux opérandes, on maintient une retenue, on passe à l'étape suivante.

Il y a trois boucles pour prendre en compte le cas où les deux opérandes n'ont pas la même taille. Lorsque tous les chiffres du nombre le plus court ont été pris en compte (première boucle `while`), on termine de propager la retenue (deuxième boucle), puis on recopie les chiffres restants du nombre le plus long (troisième boucle).

L'implémentation est écrite dans un style impératif proche du C, et utilise des exceptions pour simuler `break`. Une fonction renvoyant deux valeurs, la primitive `add_with_carry`, est utilisée. Elle additionne deux entiers machine et une retenue entrante et renvoie la somme sous forme de deux entiers machine : la somme modulo 2^{64} et la retenue sortante.

Le code extrait (figure 3, à droite) ressemble fortement au code WhyML. Les types natifs du C (entiers machine, pointeurs) sont utilisés directement, et le fait que le programme ait été écrit en WhyML puis extrait vers C n'ajoute pas de complexité significative par rapport à un programme écrit directement en C. Si l'on écrivait ce programme directement en C, la différence principale serait l'élimination des variables intermédiaires `o9`, `o10`, `o11`, `res2`, `res3`, `carry2`, `carry3` (qui sont des artefacts de la syntaxe interne de Why3) ainsi que `i6`, `lx4` et `ly2` (qui

```
type uint64
val function to_int (n:uint64) : int
meta coercion function to_int
predicate in_bounds (n:int) = 0 ≤ n ≤ 0xffff_ffff_ffff_ffff
axiom to_int_in_bounds: forall n:uint64. in_bounds (to_int n)

val add (x y:uint64) : uint64
  requires { in_bounds (to_int x + to_int y)  }
  ensures  { to_int result = to_int x + to_int y }
```

FIGURE 2 – Extrait de la spécification Why3 des entiers machine.

```
let wmpn_add (r x y:ptr uint64) (sx sy:int32) : uint64
    requires { 0 ≤ sy ≤ sx }
    requires { valid x sx }
    requires { valid y sy }
    requires { valid r sx }
    ensures { value r sx + (power radix sx) * result =
              value x sx + value y sy }
    ensures { 0 ≤ result ≤ 1 }
    writes { r.data.elts }
 =
    let zero = UInt64.of_int 0 in
    let lx = ref zero in
    let ly = ref zero in
    let c = ref zero in
    let i = ref (Int32.of_int 0) in
    while Int32.(<) !i sy do
      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;
      i := Int32.(+) !i (Int32.of_int 1);
    done;
    try
    while Int32.(<) !i sx do
      if (UInt64.(=) !c zero) then raise Break;
      lx := get_ofs x !i;
      let res, carry = add_with_carry !lx zero !c in
      set_ofs r !i res;
      c := carry;
      i := Int32.(+) !i (Int32.of_int 1);
    done
    with Break → assert { !c = 0 }
    end;
    while Int32.(<) !i sx do
      lx := get_ofs x !i;
      set_ofs r !i !lx;
      i := Int32.(+) !i (Int32.of_int 1);
    done;
    !c
```

```
uint64_t wmpn_add(uint64_t * r4,
    uint64_t * x5, uint64_t * y3,
    int32_t sx, int32_t sy)
{
  uint64_t lx4, ly2, c2;
  uint64_t res2, carry2, res3, carry3;
  int32_t i6, o9, o10, o11;
  lx4 = (0ULL); ly2 = (0ULL);
  c2 = (0ULL); i6 = (0);
  while (i6 < sy)
    {
      lx4 = *(x5+(i6));
      ly2 = *(y3+(i6));
      (add64_with_carry)(&res2, &carry2,
          lx4, ly2, c2);
      *(r4+(i6)) = res2;
      c2 = carry2;
      o9 = (i6 + 1);
      i6 = o9;
    }
  while (i6 < sx)
    {
      if(c2 == 0ULL)
        break;
      lx4 = *(x5+(i6));
      (add64_with_carry)(&res3, &carry3,
          lx4, 0ULL, c2);
      *(r4+(i6)) = res3;
      c2 = carry3;
      o10 = (i6 + 1);
      i6 = o10;
    }
  while (i6 < sx)
    {
      lx4 = *(x5+(i6));
      *(r4+(i6)) = lx4;
      o11 = (i6 + 1);
      i6 = o11;
    }
  return (c2);
}
```

FIGURE 3 – Fonction d'addition de grands entiers et code extrait correspondant.

sont présentes dans le code WhyML pour faciliter la preuve). Cependant, cette optimisation est largement à la portée du compilateur C, il n'y a donc pas de perte de performance. Nous prenons le parti de ne pas trop optimiser le programme à l'extraction, afin de garder le mécanisme (non formellement vérifié) aussi simple et transparent que possible. L'enjeu principal est d'éviter les structures de contrôle complexes et les indirections, qui pourraient être mal optimisées même par un bon compilateur.

5 Conclusion

L'objectif de cette procédure d'extraction est de développer des programmes C efficaces et vérifiés formellement. Nous avons déjà pu nous en servir pour obtenir une bibliothèque d'arithmétique en précision arbitraire [1] offrant les mêmes primitives et des performances similaires à celles de GMP [2] pour les entiers de taille inférieure à 1000 bits [9].

Notre modèle mémoire et la notion de pointeurs qu'il fournit nous ont permis d'écrire les fonctions WhyML de notre bibliothèque en collant de très près à l'implémentation de GMP, et de développer une procédure d'extraction simple de Why3 vers C. De plus, comme le modèle mémoire s'appuie sur le mécanisme de contrôle d'alias de Why3, les spécifications et les preuves de nos fonctions ne concernent que leurs propriétés arithmétiques, sans s'embarrasser de conditions sur les alias.

Une prochaine étape de ce travail pourrait consister à extraire non seulement le code des fonctions vers C, mais aussi leurs spécifications (par exemple vers le langage de spécifications ACSL[1]), afin de ne plus devoir inclure l'extraction dans la base de confiance. Le code C extrait pourrait ensuite être vérifié à l'aide d'outils existants de vérification de code C en tirant parti du fait que les algorithmes ont déjà été prouvés à l'aide de Why3.

Remerciements Je remercie Pascal Cuoq et Guillaume Melquiond pour leurs remarques et suggestions.

Références

[1] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. ACSL : ANSI/ISO C specification language, 2008. http://frama-c.com/acsl.html.

[2] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. https://hal.inria.fr/hal-00790310.

[3] Jean-Christophe Filliâtre. One logic to use them all. In *24th International Conference on Automated Deduction (CADE-24)*, volume 7898 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Lake Placid, USA, June 2013. Springer.

[4] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A pragmatic type system for deductive verification. Research report, Université Paris Sud, 2016. https://hal.archives-ouvertes.fr/hal-01256434v3.

[5] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. *Formal Methods in System Design*, 48(3) :152–174, 2016.

1. http://toccata.lri.fr/gallery/multiprecision.en.html
2. http://gmplib.org/

[6] 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.

[7] International Organization for Standardization. *ISO/IEC 9899 :1999 : Programming Languages – C*, 2000.

[8] K. Rustan M. Leino and Michał Moskal. Usable auto-active verification. In *Usable Verification Workshop*, Redmond, WA, USA, November 2010.

[9] 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*, Heidelberg, Germany, July 2017.