Double WP : Vers une preuve automatique d'un compilateur

Martin Clochard, Léon Gondelman

Double WP : Vers une preuve automatique d'un compilateur

Martin Clochard[1] & Léon Gondelman[1] *

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

Résumé

Nous présentons une expérience qui vise à certifier un compilateur de manière la plus automatique possible. Nous avons mené cette expérience en utilisant l'environnement de preuve de programme Why3, ce qui nous permet de décharger les obligations de preuve à l'aide de prouveurs, notamment automatiques. Nous avons étudié deux approches différentes. La première, une approche directe, s'est montrée peu satisfaisante car très manuelle.

Si l'on voit la preuve d'un compilateur comme la vérification que le code généré satisfait une spécification (dérivée du code source), on peut ramener la correction du compilateur à la correction du code qu'il produit. Notre deuxième approche consiste donc à exploiter des méthodes de preuve de programmes, proches de celles employées par Why3 lui-même, au niveau du code généré. Une telle méthode rend la preuve du compilateur quasi-automatique.

0. Introduction

La motivation de cet article est la suivante : à quel point peut-on automatiser la preuve d'un compilateur ? Avec d'un côté les progrès récents dans la preuve formelle des compilateurs, et de l'autre côté l'essor des prouveurs automatiques, il semble qu'il s'agisse d'une question d'actualité.

À l'heure actuelle, l'exemple le plus impressionnant de compilateur certifié est Compcert [10], développé avec l'assistant de preuve Coq. Majoritairement manuelle, cette preuve a également nécessité un effort considérable. Il est donc naturel de chercher à augmenter le degré d'automatisation d'une telle preuve. Nous avons donc décidé de mener une expérience qui vise à certifier un compilateur de manière la plus automatique possible. À cette fin, nous avons choisi un exemple jouet, présenté par Xavier Leroy lors d'un cours à l'OPLSS (Oregon Programming Languages Summer School) [1] sur la vérification mécanisée des compilateurs, qui effectue la traduction d'un langage impératif simple vers une machine virtuelle. Bien que relativement modeste, ce choix suffit à illustrer la complexité des problèmes rencontrés. Un autre intérêt est que ce compilateur est accompagné d'une preuve Coq qui nous fournit un élément de comparaison.

Pour augmenter le degré d'automatisation, la solution qui vient immédiatement à l'esprit est d'employer les prouveurs automatiques. Un cadre qui semble bien adapté est l'outil de vérification déductive de programmes Why3, qui permet de spécifier et de prouver des programmes via une large gamme de prouveurs externes. Nous avons donc choisi ce cadre-là pour mener notre expérience. Nous avons d'abord tenté une approche directe, dans l'esprit de la preuve Coq.

*. Ce travail est en partie soutenu par Bware (ANR-12-INSE-0010, `http://bware.lri.fr/`), projet de l'Agence Nationale de la Recherche (ANR).

1. `http://pauillac.inria.fr/~xleroy/courses/Eugene-2011/`

Cependant, elle s'est montrée peu satisfaisante : les prouveurs automatiques s'avèrent incapables d'effectuer les preuves sans aide manuelle conséquente.

Si l'on voit la preuve d'un compilateur comme la vérification que quel que soit le code source, le code généré satisfait une spécification (dérivée du code source), on peut ramener la correction du compilateur à la correction du code qu'il produit. L'idée principale est donc d'utiliser le code source comme spécification du code généré. L'approche que nous présentons dans cet article consiste à exploiter des méthodes de preuve de programmes, proches de celles employées par Why3 lui-même (calcul de plus faible précondition), au niveau du code généré. Concrètement, on introduit à l'intérieur de Why3 un langage de spécification pour la machine virtuelle ainsi que des combinateurs permettant d'assembler non seulement les parties du code mais également leur spécifications. Une telle approche rend la preuve du compilateur quasi-automatique. Le développement Why3 présenté dans cet article peut être trouvé à l'adresse suivante : `http://toccata.lri.fr/gallery/double_wp.fr.html`.

1. Why3 : outil de vérification déductive de programmes

L'outil Why3 propose un langage riche, appelé WhyML, pour la spécification et l'écriture de programmes vérifiés. Il repose sur l'utilisation de prouveurs externes, qu'ils soient automatiques ou interactifs, pour obtenir les preuves des lemmes auxiliaires ainsi que des conditions de vérification.

Le fragment logique de WhyML [2], utilisé pour annoter les programmes, est une extension de la logique du premier ordre comprenant notamment des types polymorphes à la ML, des types algébriques, des définitions (co-)inductives et récursives, ainsi que certaines constructions d'ordre supérieur [4]. Les constructions comme le filtrage et la liaison peuvent être employées directement dans les formules et les termes. Comme la majorité des prouveurs employés ne supportent pas toutes les constructions du langage, Why3 utilise durant le processus de génération des obligations de preuve une série de transformations pour éliminer celles qui ne sont pas supportées. Les transformations font partie de la base de confiance de Why3. Elles peuvent également être employées explicitement par l'utilisateur pour changer la forme des obligations générées, *a priori* dans le but de faciliter la tâche aux prouveurs automatiques. Le développement présenté dans cet article repose sur des transformations, dont certaines développées par les auteurs.

Le langage de programmation lui-même [7] est un dialecte de ML avec un certain nombre de restrictions pour rendre la preuve automatique viable. Par exemple, les procédures d'ordre supérieur ne sont pas acceptées. Pour décrire le comportement attendu d'un programme, les définitions de fonctions WhyML sont annotées par des contrats, sous la forme de pré- et postconditions. Les boucles sont également munies d'invariants. Dans le but d'assurer la terminaison, les définitions récursives ainsi que les boucles peuvent également être munies de variants, c'est-à-dire des valeurs qui décroissent à chaque itération pour un ordre bien fondé. Il est également possible d'ajouter des assertions vérifiées statiquement à des emplacements arbitraires du programme. L'outil Why3 utilise ces annotations pour générer des conditions de vérification via un calcul de plus faible précondition.

Le langage WhyML permet également d'écrire du code fantôme [6], c'est-à-dire des calculs et données qui servent uniquement à aider la vérification et peuvent être retirés du programme sans changer son comportement observable. Un usage typique du code fantôme est de propager les témoins de certaines propriétés existentielles.

Pour une présentation plus détaillée de Why3 et WhyML, nous invitons le lecteur à se référer à la page web du projet `http://why3.lri.fr`, qui propose une introduction détaillée ainsi qu'une grande collection d'exemples.

2. Compiler un mini-langage impératif vers une machine virtuelle

Cette section décrit la syntaxe et la sémantique du langage source et du langage cible du mini-compilateur. Nous y présentons également le compilateur ainsi que l'énoncé de sa correction.

2.0. Langage source

Il s'agit d'un langage impératif simple, communément appelé Imp. Chaque programme de Imp est composé d'une séquence de commandes qui servent à manipuler l'état global de l'exécution à travers des variables globales. On définit l'état et les variables avec les types suivants :

```
type id = Id int           (* les noms des variables *)
type state = map id int    (* l'état global (équivalent à id -> int) *)
```

Les variables, dénotées par des entiers, correspondent toujours à des valeurs entières. Outre les variables, à l'intérieur d'une commande on manipule des expressions arithmétiques ou booléennes :

```
type aexpr =
 | Anum int           (*  n ∈ ℤ  *)
 | Avar id            (*    X    *)
 | Aadd aexpr aexpr   (* a1 + a2 *)
 | Asub aexpr aexpr   (* a1 - a2 *)
 | Amul aexpr aexpr   (* a1 * a2 *)
```

```
type bexpr =
 | Btrue              (*   vrai   *)
 | Bfalse             (*   faux   *)
 | Band bexpr bexpr   (* b1 ∧ b2 *)
 | Bnot bexpr         (*   ¬b     *)
 | Beq aexpr aexpr    (* a1 = a2 *)
 | Ble aexpr aexpr    (* a1 ≤ a2 *)
```

Enfin, les commandes de Imp sont représentées par le type suivant :

```
type com =
  | Cskip              (*         SKIP          *)
  | Cassign id aexpr   (*        X := a         *)
  | Cseq com com       (*        c1 ; c2        *)
  | Cif bexpr com com  (*  IF b THEN c1 ELSE c2 *)
  | Cwhile bexpr com   (*   WHILE b DO c DONE   *)
```

Voici comment on peut écrire la factorielle en Imp :

```
X := 1; WHILE ¬(Y ≤ 0) DO X := X * Y; Y := Y - 1 DONE
```

La sémantique du langage Imp est une sémantique opérationnelle standard à grands pas. Nous la formalisons en Why3 de la même façon que dans le développement Coq, à savoir par des fonctions d'évaluation pour les expressions et un prédicat inductif pour les commandes :

```
function aeval  (st: state) (a: aexpr) : int  = ...
function beval  (st: state) (b: bexpr) : bool = ...

(* exécuter cmd dans l'état mi donne l'état mf *)
inductive ceval (mi: state) (cmd: com) (mf: state) = ...
```

Voici par exemple la règle qui donne la sémantique de la séquence :

```
| E_Seq : forall cmd1 cmd2 m0 m1 m2.
  ceval m0 cmd1 m1 → ceval m1 cmd2 m2 → ceval m0 (Cseq cmd1 cmd2) m2
```

Les lecteurs intéressés trouveront les définitions complètes en appendice.

Enfin, remarquons que cette sémantique est déterministe. Cette propriété est prouvée en Why3 en utilisant une transformation d'induction (voir section 5) et les prouveurs automatiques.

```
lemma ceval_deterministic :
    forall c mi mf1 mf2. ceval mi c mf1 →  ceval mi c mf2 → mf1 = mf2
```

2.1. Langage cible

Il s'agit d'un langage bas niveau interprété par une machine virtuelle. Chaque programme est composé d'une séquence d'instructions. La machine virtuelle est équipée d'une pile et d'une mémoire :

```
type pos = int                              (* position de la tête de lecture *)
type stack = list int                       (* pile d'entiers                 *)
type machine_state = VMS pos stack state (* état de la machine virtuelle   *)
```

À chaque étape, la machine procède en exécutant l'instruction qui se trouve au niveau de la tête de lecture, puis avance celle-ci d'un cran. Le jeu d'instructions est décrit en Why3 par le type suivant :

```
type ofs   = int
type instr =
  | Iconst int   (* met un entier sur la pile                          *)
  | Ivar id      (* met la valeur de la variable sur la pile           *)
  | Isetvar id   (* dépile n, assigne la variable à n                  *)
  | Ibranch ofs  (* saute le nombre d'instructions donné               *)
  | Iadd         (* dépile deux valeurs, empile leur somme             *)
  | Isub         (* dépile n2, dépile n1, empile n1 - n2               *)
  | Imul         (* dépile deux valeurs, empile leur produit           *)
  | Ibeq ofs     (* dépile n2, dépile n1, effectue Ibranch ssi n1 = n2 *)
  | Ibne ofs     (* dépile n2, dépile n1, effectue Ibranch ssi n1 ≠ n2 *)
  | Ible ofs     (* dépile n2, dépile n1, effectue Ibranch ssi n1 ≤ n2 *)
  | Ibgt ofs     (* dépile n2, dépile n1, effectue Ibranch ssi n1 > n2 *)
  | Ihalt        (* arrêt de la machine                                *)

type code = list instr
```

Note : Le jeu d'instructions que nous présentons est légèrement différent de celui utilisé dans le développement Coq. Comme nous utilisons des entiers relatifs, nous avons fusionné les instructions de saut avant/arrière en une seule (`Ibranch`).

Voici l'exemple de la factorielle réécrit en langage cible :

```
(*  X = Id 0, Y = Id 1 *)
[ Iconst 1; Isetvar X; Ivar Y   ; Iconst 0     ; Ible 9;
  Ivar X  ; Ivar Y   ; Imul     ; Isetvar X    ; Ivar Y;
  Iconst 0; Isub     ; Isetvar Y; Ibranch (-12); Ihalt ]
```

Un autre exemple de programme correspondant à une boucle infinie :

```
[ Ibranch (-1); Ihalt ]
```

La sémantique de la machine est une sémantique opérationnelle à petits pas ce que nous formalisons en Why3 par un prédicat inductif. Celui-ci représente le changement d'état lors d'une étape de calcul de la machine virtuelle. Ce prédicat, dont la définition se trouve en appendice, reflète la sémantique informelle donnée en commentaire.

```
inductive transition (c: code) (msi msj: machine_state) = ...

(* Clôture réflexive transitive *)
inductive transition_star (c: code) (msi msj: machine_state) = ...
```

2.2. Compilateur

Le compilateur consiste en trois fonctions de programme qui compilent respectivement les trois catégories syntaxiques du langage Imp :

```
let rec compile_aexpr (a: aexpr) : code = ...
let rec compile_bexpr (b: bexpr) (cond: bool) (ofs: ofs) : code = ...
let rec compile_com   (cmd: com) : code = ...
```

Le schéma de compilation d'une expression arithmétique consiste à mettre sa valeur au sommet de la pile. Voici par exemple comment on compile `Asub a1 a2` :

```
(* ++ représente la concaténation *)
compile_aexpr a1 ++ compile_aexpr a2 ++ Cons Isub Nil
```

Les expressions booléennes sont traduites vers des branchements : si une telle expression s'évalue à `cond`, l'exécution du code compilé effectue un saut de `ofs`.

Enfin, la compilation d'un programme `Imp` consiste à ajouter l'instruction d'arrêt à la compilation de sa commande.

```
let compile_program (prog : com) : code = compile_com prog ++ Cons Ihalt Nil
```

2.3. Énoncé de la correction du compilateur

Intuitivement, un compilateur est correct si, que l'on exécute le programme de départ P ou le programme compilé $\widehat{P}$, on obtient le même résultat. Concrètement, nous allons montrer un résultat de *simulation en avant*, c'est-à-dire que si P termine dans un état donné, l'exécution de $\widehat{P}$ avec les mêmes conditions initiales termine dans le même état. On exprime cela en donnant à `compile_program` la postcondition suivante :

```
let compile_program (prog : com) : code
  ensures { forall mi mf: state.
   ceval mi prog mf → vm_terminates result[2] mi mf }
```

Ici, `vm_terminates` exprime la terminaison de la machine, c'est-à-dire que la tête de lecture est positionnée sur une instruction `Ihalt`, et que la pile, utilisée pour les calculs intermédiaires, est vide.

```
(* le code c se trouve à la position p dans le code c_glob *)
inductive codeseq_at (c_glob: code) (p: pos) (c: code) =
| codeseq_at_intro : forall c1 c2 c3. codeseq_at (c1 ++ c2 ++ c3) (length c1) c2

predicate vm_terminates (c:code) (mi mf: state) =
  exists p. codeseq_at c p (Cons Ihalt Nil) ∧
    transition_star c (VMS 0 Nil mi) (VMS p Nil mf)
```

Remarquons au passage que nous nous intéressons uniquement aux comportements terminants. L'énoncé de correction choisi ne donne aucune garantie si le programme de départ diverge.

3. Preuve du compilateur : l'approche naïve

Commençons par essayer de prouver la correction pour la compilation des expressions arithmétiques. On exprime cela en donnant à `compile_aexpr` la postcondition suivante :

```
let rec compile_aexpr (a : aexpr) : code
  ensures { forall c: code, p: pos, m: state, s: stack.
    codeseq_at c p result →
    transition_star c (VMS p s m)
                      (VMS (p + length result) (Cons (aeval m a) s) m) }
```

c'est-à-dire que, quel que soit l'endroit où se trouve le code compilé, son exécution déplace la tête de lecture et met la valeur de l'expression arithmétique au sommet de la pile.

Les deux premiers cas, à savoir la compilation des constantes et des variables, sont prouvés automatiquement, car l'énoncé de correction correspond exactement aux règles de transition pour les instructions respectives.

Ce n'est malheureusement pas le cas pour les opérations binaires. Focalisons-nous sur le cas de la soustraction :

```
| Asub a1 a2 → compile_aexpr a1 ++ compile_aexpr a2 ++ Cons Isub Nil
```

2. `result` représente le résultat renvoyé par la fonction

Prouver la correction de cette traduction revient à montrer l'existence d'une séquence de transitions partant de l'état `VMS p s m` vers l'état

```
VMS (p + length a1 + length a2 + 1) (Cons (aeval m a1 - aeval m a2) s) m
```

On obtient cette séquence en appliquant l'énoncé de correction sur les sous-expressions, ce qui nous donne les états intermédiaires suivants :

```
VMS (p + length a1)              (Cons (aeval m a1) s)                      m
VMS (p + length a1 + length a2) (Cons (aeval m a2) (Cons (aeval m a1) s)) m
```

Cependant, nous avons observé que parmi la douzaine de prouveurs automatiques que nous avons essayés, aucun n'est parvenu à trouver ces états [3]. On pourrait penser que la difficulté survienne de la nécessité d'exhiber des instances du prédicat `codeseq_at` pour chaque transition intermédiaire. Or, quelques changements pour contourner cette difficulté n'ont pas amélioré la situation. Il est donc plus que probable que le problème réside dans la construction de ces états intermédiaires.

Une première solution est d'utiliser un prouveur interactif, comme Coq, mais cela n'apporterait évidemment rien en termes d'automatisation vis-à-vis d'une preuve effectuée dans un tel environnement. Une autre possibilité serait d'aider les prouveurs automatiques en exhibant les états intermédiaires via les assertions de Why3. Dans le cas de la soustraction, on obtient :

```
let c1 = compile_aexpr a1 in let c2 = compile_aexpr a2 in
let isub = Cons Isub Nil in let c12 = c1 ++ c2 in
let res = c12 ++ isub in
  assert { forall c: code, p: pos, m: state, s: stack.
    codeseq_at c p res →
    let state1 = VMS p s m in  let s2 = Cons (aeval m a1) s in
    let state2 = VMS (p + length c1) s2 m in
    let state3 = VMS (p + length c12) (Cons (aeval m a2) s2) m in
    let state4 = VMS (p + length res) (Cons (aeval m a) s) m in
    transition_star c state1 state2 && codeseq_at c (p + length c1) c2 &&
    transition_star c state2 state3 && transition_star c state3 state4 &&
    transition_star c state1 state4 };
res
```

Cependant, mettre une assertion revient ici à placer une preuve explicite dans le code Why3, et n'apporte donc rien non plus à l'automatisation de la preuve.

À l'état actuel de la preuve automatique, il semble que l'on ne puisse pas prouver directement l'énoncé de correction. Ne serait-ce que pour le cas simple des expressions arithmétiques, une telle approche nécessite une quantité trop importante de travail manuel pour mériter le nom de preuve automatique. Nous allons donc aborder la vérification du compilateur sous un angle différent.

4. Double WP

Dans cette section, nous présentons une autre approche de vérification du compilateur. Celle-ci consiste à plonger dans Why3 une logique de programme, la logique de Hoare, pour la machine virtuelle. Pour rendre la preuve plus automatique nous introduisons ensuite un calcul de plus faible précondition. Nous illustrons ensuite comment cette approche s'applique à la preuve du compilateur.

4.0. Triplets de Hoare

Nous définissons une logique de Hoare adaptée au code non structuré de la machine virtuelle. L'idée est d'utiliser les pré- et postconditions pour exprimer le fait que le code généré se comporte de la même manière que le code source. On commence donc par définir la forme des triplets de Hoare en Why3 :

3. dans un délai de deux minutes

```
type pre  = pos → machine_state → bool
type post = pos → machine_state → machine_state → bool
type hl   = { code: code; ghost pre: pre; ghost post: post }
```

Comme on peut le remarquer, les pré- et postconditions sont paramétrées par une position. Celle-ci indique la position à laquelle se trouvent les instructions spécifiées par rapport au programme global. On remarque également que la postcondition parle de deux états machine, le premier étant l'état initial. Notons enfin que ces annotations font partie du code fantôme et n'interviendront donc pas à l'exécution du compilateur.
Note : Au lieu de paramétrer les spécifications par une position *absolue*, nous aurions pu donner les positions d'une manière *relative* dans les spécifications, autrement dit comme si les instructions spécifiés se trouvaient à la position zéro. Néanmoins, il serait alors nécessaire d'introduire explicitement des décalages de la tête de lecture dans *toutes* les annotations lorsque le code est placé à une autre position que zéro. Nous avons choisi la première solution car elle nous a paru moins intrusive.

Le sens que nous attribuons à ces triplets correspond à la notion de *correction totale*. Autrement dit, quel que soit l'état initial, si la précondition est vérifiée, on atteindra un état final où la postcondition est vérifiée. L'énoncé ci-dessous exprime bien la correction totale, car la machine virtuelle est déterministe.

```
predicate contextual_irrelevance (c: code) (p: pos) (ms1 ms2: machine_state) =
  forall c_glob. codeseq_at c_glob p c → transition_star c_glob ms1 ms2

predicate hl_correctness (cs: hl) =
  forall p ms. cs.pre p ms →
    exists ms'. cs.post p ms ms' ∧ contextual_irrelevance cs.code p ms ms'
```

Armés de ces triplets, nous pouvons ré-attaquer la preuve du compilateur. Commençons par reformuler l'énoncé de sa correction. Dans l'énoncé ci-dessous, `\x1...xn.e` dénote $\lambda x1 \ldots xn.e$.

```
function com_pre (cmd: com) : pre =
  \p ms. let VMS p' _ m = ms in p = p' ∧ exists m'. ceval m cmd m'

function com_post (cmd: com) (len: pos) : post =
  \p0 ms ms'. let VMS p s m = ms in let VMS p' s' m' = ms' in
    p' = p + len ∧ s' = s ∧ ceval m cmd m'

let rec compile_com (cmd: com) : hl
  ensures  { result.pre = com_pre cmd }
  ensures  { result.post = com_post cmd result.code.length }
  ensures  { hl_correctness result } = ...
```

Cette fois, à l'issue de la compilation, nous obtenons des triplets dont la validité est assurée par la spécification de `compile_com`. L'énoncé de correction initial est la conséquence immédiate de cette validité, dont on peut se persuader en déroulant la définition de `hl_correctness`. Nous voyons donc un compilateur certifié comme un compilateur produisant du code certifié.

Pour établir efficacement la validité de tels triplets, il est naturel d'établir des règles de raisonnement. Par conséquent, on attribue une spécification à chaque instruction machine. Par exemple, voici la spécification de l'instruction de soustraction :

```
constant ibinop_pre : pre =
  \p ms. exists n1 n2 s m. ms = VMS p (Cons n2 (Cons n1 s)) m

function ibinop_post (op : int → int → int) : post =
  \p ms ms'. forall n1 n2 s m. ms  = VMS  p    (Cons n2 (Cons n1 s)) m →
                               ms' = VMS (p+1) (Cons (op n1 n2) s)   m

let isubf () : hl
  ensures { result.pre = ibinop_pre ∧ result.post = ibinop_post (\x y. x - y) }
```

```
    ensures { result.code.length = 1  ∧ hl_correctness result } = ...
```

La précondition des opérations binaires requiert que deux opérandes soient présents au sommet de la pile, et la postcondition exprime que ceux-ci ont été remplacés par le résultat de l'opération.

Cela ne suffit cependant pas pour vérifier les triplets générés par le compilateur. Il faut également donner sous une forme ou une autre des règles de combinaison des triplets.

4.1. Calcul de WP

Revenons à la compilation des expressions arithmétiques. Comme pour les commandes, nous donnons d'abord la spécification des triplets générés.

```
function trivial_pre : pre = \p ms. let VMS p' _ _ = ms in p = p'
function aexpr_post (a: aexpr) (len: pos) : post  =
  \p ms ms'. let VMS _ s m = ms in ms' = VMS (p + len) (Cons (aeval m a) s) m

let rec compile_aexpr (a: aexpr) :  hl
  ensures { result.pre  = trivial_pre ∧ hl_correctness result }
  ensures { result.post = aexpr_post a result.code.length } = ...
```

Dans les cas où la compilation génère un triplet comportant une seule instruction (`Anum`, `Avar`), l'application directe d'une règle d'affaiblissement suffit. Ce n'est pas aussi immédiat pour la compilation des opérateurs binaires où une simple application de la règle de séquence ne suffit pas. Il faudrait combiner celle-ci avec des règles d'affaiblissement et de "frame", ce qui demanderait encore un nombre d'annotations trop important pour une preuve automatique. Nous avons donc employé une technique connue pour contourner ce problème, à savoir prouver le code compilé en utilisant un calcul similaire au *calcul de plus faible précondition (Weakest Precondition Calculus)*. L'intérêt principal de ce calcul est de pouvoir appliquer et combiner les règles de raisonnement d'une manière automatique. Il est intéressant de remarquer que c'est également la technique employée par Why3 lui-même pour vérifier notre compilateur.

Comme pour les triplets, nous introduisons d'abord une forme d'annotation du code qui correspond aux *transformateurs de prédicats en arrière (backward predicate transformers)*.

```
type wp_trans = pos → (machine_state → bool) → (machine_state → bool)

type wp = { wcode : code ; ghost wp : wp_trans }

predicate wp_correctness (code: wp) =
  forall p post ms. (code.wp p post) ms →
    exists ms'. post ms' ∧ contextual_irrelevance code.wcode p ms ms'
```

Étant donnée une propriété Q portant sur les états machine, un tel transformateur (de type `wp_trans`) renvoie une condition suffisante sur l'état de départ pour que la machine finisse par arriver dans un état vérifiant Q.

Maintenant que nous avons deux formes différentes d'annotations, il est nécessaire d'introduire des convertisseurs de l'une vers l'autre. Le premier convertisseur (`$`) correspond exactement au calcul de plus faible précondition pour un contrat.

```
function towp_wp (pr: pre) (ps: post) : wp_trans =
  \p q ms. pr p ms && forall ms'. ps p ms ms' → q ms'

let ($_) (c: hl) : wp
  requires { hl_correctness c }
  ensures  { result.wcode.length = c.code.length }
  ensures  { result.wp = towp_wp c.pre c.post ∧ wp_correctness result }
= { wcode = c.code; wp = towp_wp c.pre c.post }
```

Le convertisseur dans l'autre sens correspond alors à la vérification d'un tel contrat vis-à-vis de la précondition calculée, comme l'indique sa précondition.

```
let hoare (ghost pre: pre) (c: wp) (ghost post: post) : hl
  requires { wp_correctness c }
  (* correction du contrat *)
  requires { forall p ms. pre p ms → (c.wp p (post p ms)) ms }
  ensures  { result.pre = pre ∧ result.post = post }
  ensures  { result.code.length = c.wcode.length ∧ hl_correctness result }
= { code = c.wcode; pre = pre; post = post }
```

Nous introduisons ensuite un combinateur correspondant à l'exécution en séquence de deux listes d'instructions consécutives.

```
function seq_wp (l1: int) (w1: wp_trans) (w2: wp_trans) : wp_trans =
  \p q ms. w1 p (w2 (p + l1) q) ms
```

Cette fonction logique, que l'on utilise dans le combinateur ci-dessous, correspond à la règle de calcul usuelle pour la plus faible précondition d'une séquence. Comme nous l'avons mentionné au début de cette section, le choix des positions absolues nous épargne des décalages explicites de la tête de lecture dans la condition suffisante calculée.

```
let (~) (s1: wp) (s2: wp) : wp
  requires { wp_correctness s1 ∧ wp_correctness s2 }
  ensures  { result.wcode.length = s1.wcode.length + s2.wcode.length }
  ensures  { result.wp = seq_wp s1.wcode.length s1.wp s2.wp }
  ensures  { wp_correctness result }
= { wcode = s1.wcode ++ s2.wcode; wp = seq_wp s1.wcode.length s1.wp s2.wp }
```

En employant ces définitions, nous réécrivons le compilateur des expressions arithmétiques sous la forme suivante :

```
let rec compile_aexpr (a:aexpr) :  hl
  ensures { result.pre = trivial_pre ∧ hl_correctness result }
  ensures { result.post = aexpr_post a result.code.length }
  variant { a } =
  let c = match a with
    | Anum n     → $ iconstf n
    | Avar x     → $ ivarf x
    | Aadd a1 a2 → $ compile_aexpr a1 ~ $ compile_aexpr a2 ~ $ iaddf ()
    | Asub a1 a2 → $ compile_aexpr a1 ~ $ compile_aexpr a2 ~ $ isubf ()
    | Amul a1 a2 → $ compile_aexpr a1 ~ $ compile_aexpr a2 ~ $ imulf ()
    end in hoare trivial_pre c (aexpr_post a c.wcode.length)
```

Expliquons en détail le schéma de preuve. Les résultats intermédiaires de la compilation sont d'abord combinés en tant que transformateurs de prédicat. Il suffit ensuite de vérifier la compatibilité du transformateur résultant vis-à-vis des pré- et postconditions données. Concrètement, cela consiste à montrer que la précondition implique la condition suffisante obtenue en appliquant le transformateur à la postcondition voulue. Remarquons que dans chaque cas, il s'agit de la seule obligation de preuve non triviale, à la fois au sens logique et vis-à-vis des prouveurs automatiques.

Comparé à l'approche naïve, le problème des états intermédiaires est résolu, car ils sont introduits automatiquement par le combinateur de séquence. Cependant, les prouveurs automatiques peinent à vérifier les obligations de preuve telles quelles, à cause des nombreuses définitions d'ordre supérieur. En pratique, nous contournons ce problème en appliquant systématiquement la transformation `compute`, qui simplifie autant que possible la forme des buts grâce à des règles de réécriture précisées par l'utilisateur. Les règles de réécriture que nous avons employées déroulent les définitions des objets d'ordre supérieurs impliqués. Cela a pour effet de calculer effectivement les conditions suffisantes obtenues à partir du code généré, en éliminant entièrement les objets susmentionnés.

Après cette transformation, la totalité des obligations de preuve pour `compile_aexpr` est vérifiée automatiquement en moins de 5 secondes.

4.2. Compilation des expressions booléennes

Encouragés par ce succès, continuons la vérification du compilateur avec ces méthodes. La preuve de la compilation des expressions booléennes n'est pas fondamentalement différente de celle pour les expressions arithmétiques. Introduisons d'abord l'énoncé de correction. Le code généré doit effectuer un saut si et seulement si l'expression booléenne s'évalue au paramètre supplémentaire `cond` :

```
function bexpr_post (b: bexpr) (cond: bool) (out_t: ofs) (out_f: ofs) : post =
  \p ms ms'. let VMS _ s m = ms in if beval m b = cond
      then ms' = VMS (p + out_t) s m
      else ms' = VMS (p + out_f) s m

let rec compile_bexpr (b: bexpr) (cond: bool) (ofs: ofs) :  hl
  ensures { result.pre = trivial_pre ∧ hl_correctness result }
  ensures { let len = result.code.length in
            result.post = bexpr_post b cond (len + ofs) len } = ...
```

À l'exception du cas de la conjonction, la preuve s'effectue automatiquement comme pour les expressions arithmétiques. La particularité de la conjonction tient au fait que le code généré pour `Band b1 b2` s'exécute d'une manière paresseuse, c'est-à-dire que le code correspondant à `b2` n'est exécuté que si `b1` est vraie. Or, il ne s'agit plus d'une exécution en séquence, mais d'une exécution conditionnelle. Nous devons donc introduire un nouveau combinateur qui reflète ce comportement.

```
function fork_wp (w: wp_trans) (cond: pre) : wp_trans =
  \p q ms. (cond p ms → w p q ms) ∧ (not cond p ms → q ms)

let (%) (s: wp) (ghost cond: pre) : wp
  requires { wp_correctness s }
  ensures  { result.wp = fork_wp s.wp cond }
  ensures  { result.wcode.length = s.wcode.length ∧ wp_correctness result }
= { wcode = s.wcode; wp = fork_wp s.wp cond }
```

Ce combinateur permet d'ignorer le code sur lequel il s'applique lorsque la condition est fausse. Nous pouvons alors compiler la conjonction de la manière suivante :

```
function exec_cond (b: bexpr) (cond: bool) : pre =
  \p ms. let VMS _ _ m = ms in beval m b = cond
        ⋮

| Band b1 b2  →  let c2 = $ compile_bexpr b2 cond ofs % exec_cond b1 true in
    let len = length c2.wcode in let ofs = if cond then len else ofs + len in
    $ compile_bexpr b1 false ofs ~ c2
```

Ce cas est alors lui aussi prouvé automatiquement.

4.3. Compilation des commandes

Pour finir, il nous reste à vérifier la correction de la compilation des commandes, dont l'énoncé est donné dans la partie 4.0. Comme on pouvait s'y attendre, le cas de la boucle `WHILE` est non trivial. De manière plus surprenante, le cas de la conditionnelle ne se déduit pas directement des combinateurs de séquence et d'exécution conditionnelle. En effet, pour une commande `Cif b c1 c2`, le code de la branche compilée en dernier, par exemple `c2`, n'est exécuté que si le test est faux. Or, l'état dans lequel ce test est évalué peut *a priori* être différent de l'état à l'entrée du code correspondant à `c2`. Il nous manque donc une information (l'état de départ) pour pouvoir écrire les conditions d'exécution.

La solution que nous proposons est d'employer des variables *auxiliaires*, autrement dit paramétrer les triplets de Hoare (et les transformateurs de prédicats) par des données

supplémentaires. Comme en général on ne connaît pas la nature des variables auxiliaires, nous représentons leur structure par un paramètre de type. Nous redéfinissons donc les triplets (resp. transformateurs) de la manière suivante :

```
type pre 'a  = 'a → pos → machine_state → bool
type post 'a = 'a → pos → machine_state → machine_state → bool
type hl 'a  = { code: code; ghost pre: pre 'a; ghost post: post 'a}

type wp_trans 'a = 'a → pos → (machine_state → bool) → (machine_state → bool)
type wp 'a = { wcode : code; ghost wp: wp_trans 'a}
```

La correction des triplets (resp. transformateurs) est maintenant exprimée en quantifiant universellement sur les variables auxiliaires. Au vu de cette nouvelle définition, il serait possible de mettre la position parmi les variables auxiliaires. En pratique, il est plus simple de la traiter séparément du fait de son omniprésence.

Ce changement fait, on peut adapter le combinateur de séquence afin de "photographier" l'état initial.

```
function seq_wp
  (l1: int) (w1: wp_trans 'a) (w2: wp_trans ('a, machine_state)) : wp_trans 'a
= \x p q ms. w1 x p (w2 (x, ms) (p+l1) q) ms

let (~) (s1: wp 'a) (s2: wp ('a, machine_state)) : wp 'a
  requires { wp_correctness s1 ∧ wp_correctness s2 }
  ensures  { result.wcode.length = s1.wcode.length + s2.wcode.length }
  ensures  { result.wp = seq_wp s1.wcode.length s1.wp s2.wp }
  ensures  { wp_correctness result } = ...
```

Maintenant, l'état initial peut être utilisé pour spécifier la seconde partie du code via les variables auxiliaires. En particulier, cela permet de vérifier la compilation de la conditionnelle automatiquement.

Cas de la boucle. Comme aucun des combinateurs que nous avons définis ne permet d'exprimer la notion d'exécution répétée, nous devons en introduire un. Notre définissons ce combinateur comme une règle habituelle pour la boucle dans la logique de Hoare, à savoir une règle logique entre deux triplets paramétrée par un variant et un invariant. Définir ce combinateur directement sur les transformateurs de prédicats serait également possible, mais étant donnée la quantité d'information supplémentaire (invariant et variant) nécessaire, donner aussi la postcondition n'était pas problématique. Nous nous sommes donc contentés de la version avec les triplets.

Commençons par expliquer le variant. Il s'agit dans notre cas d'une relation bien fondée sur les états qui vérifient l'invariant. Nous devons donc d'abord définir ce qu'est une relation bien fondée. Nous utilisons pour cela le prédicat usuel d'accessibilité, qui caractérise l'ensemble des éléments pour lesquels la relation est bien fondée :

```
inductive acc ('a → 'a → bool) 'a =
  | Acc : forall r, x:'a. (forall y. r y x → acc r y) → acc r x
```

Outre le variant, nous devons également spécifier le comportement de la boucle à chaque itération.

```
function loop_progress (inv post:pre 'a) (var:post 'a) : post 'a =
  \x p ms ms'. (inv x p ms' ∧ var x p ms' ms)  ∨  post x p ms'
```

Cette spécification requiert qu'à chaque itération, l'exécution progresse vers la postcondition. Autrement dit, à l'issue de l'itération, soit la postcondition est établie, soit l'invariant de la boucle est préservé et l'état devient "plus petit" au sens du variant. Nous pouvons alors définir le combinateur de boucle.

```
function forget_old (post:pre 'a) : post 'a = \x p ms . post x p
```

```
let make_loop_hl (c:hl 'a) (ghost inv post: pre 'a) (ghost var: post 'a) : hl 'a
  requires { hl_correctness c }
  requires { forall x p ms. inv x p ms → acc (var x p) ms }
  requires { c.pre = inv }
  requires { c.post = loop_progress inv post var }
  ensures  { result.pre = inv ∧ result.post = forget_old post }
  ensures  { result.code.length = c.code.length ∧ hl_correctness result }
= { code = c.code ; pre = inv ; post = forget_old post }
```

Pour des raisons de simplicité, l'état initial n'apparaît pas explicitement dans cette définition. Cela ne constitue pas un problème, car cet état peut être passé via les variables auxiliaires.

Utilisons maintenant ce combinateur pour vérifier le cas de la boucle `Cwhile b c0`. Pour cela, nous définissons d'abord les paramètres effectifs du combinateur.

```
(* c = Cwhile b c0 *)
function loop_invariant (c: com) : pre ('a, machine_state) =
  \x p msi. let VMS _ s0 m0 = snd x in let VMS pi si mi = msi in
    pi = p ∧ s0 = si ∧ exists mf. ceval m0 c mf ∧ ceval mi c mf
(* c = Cwhile b c0 *)
function loop_post (c: com) (len: pos) : pre ('a, machine_state) =
  \x p msf. let VMS _ s0 m0 = snd x in let VMS pf sf mf = msf in
    pf = p + len ∧ s0 = sf ∧ ceval m0 c mf
(* c = c0, test = b *)
function loop_variant (c: com) (test: bexpr) : post 'a =
  \x p msj msi. let VMS pj sj mj = msj in let VMS pi si mi = msi in
     pj = pi ∧ sj = si ∧ ceval mi c mj ∧ beval mi test
```

Ici, l'invariant spécifie que la boucle évalue les états intermédiaires et l'état initial en le même état final. Le variant correspond lui à une itération du corps de la boucle. La compilation de la boucle consiste donc à générer le code pour une itération, puis à appliquer le combinateur de boucle avec les paramètres ci-dessus au résultat.

```
| Cwhile test body  → let code_body = compile_com body in
    let body_length = length code_body.code + 1 in
    let code_test = compile_bexpr test false body_length in
    let ofs = length code_test.code + body_length in
    let wp_while = $ code_test ~
        ($ code_body ~ $ ibranchf (- ofs)) % exec_cond test true in
    let ghost inv  = loop_invariant cmd in
    let ghost var  = loop_variant body test  in
    let ghost post = loop_post cmd ofs in
    let hl_while = hoare inv wp_while (loop_preservation inv post var) in
    $ inil () ~ $ make_loop_hl hl_while inv post var
```

Qu'en est-il de la vérification des buts générés ? Malheureusement, une obligation de preuve résiste aux prouveurs automatiques. De manière peu surprenante, il s'agit de la vérification que le variant est bien fondé, ce qui nécessite une preuve par induction non triviale sur la sémantique du langage source. Nous avons donc effectué cette preuve en Coq en une quinzaine de lignes. Cependant, c'est le seul recours à un prouveur interactif dans la totalité du développement.

5. Transformation d'induction

Comme nous l'avons vu, certaines de nos obligations de preuve, tels que le déterminisme du langage source ou la correction du combinateur de boucle, nécessitent un raisonnement par induction sur les prédicats définis d'une manière inductive. Or, de tels raisonnements sont actuellement au-delà des capacités des prouveurs automatiques. L'une des solution proposée par Why3 est d'utiliser un prouveur interactif pour faire la preuve manuellement. C'est dommage, car

expliciter la structure de l'induction suffit en général pour revenir dans le fragment logique traité efficacement par les prouveurs automatiques.

Comme nous l'avons mentionné dans la section 1, Why3 permet d'appliquer des transformations sur les obligations de preuves. Nous avons alors développé une nouvelle transformation pour effectuer des preuves par induction. Plus précisément, cette transformation, appliquée à un but, renvoie un ensemble de buts qui correspond à la structure d'une preuve par induction sur l'un des prédicats inductifs présents parmi les hypothèses du but initial. Ce prédicat est choisi comme étant le premier rencontré, sauf si l'utilisateur l'indique explicitement dans le fichier source de Why3.

Montrons comment fonctionne cette transformation sur un exemple extrait du développement de notre compilateur, à savoir la clôture réflexive transitive. On définit cette clôture par un prédicat inductif.

```
type parameter
type state
predicate transition parameter state state

inductive transition_star parameter (x y: state) =
  | Refl: forall p x. transition_star p x x
  | Step: forall p x y z.
      transition p x y → transition_star p y z → transition_star p x z

lemma transition_star_transitive: forall p s1 s2 s3.
  transition_star p s1 s2 → transition_star p s2 s3 →
  transition_star p s1 s3
```

Pour montrer qu'il s'agit effectivement d'une relation transitive, il faut raisonner par induction sur le prédicat `transition_star`. En appliquant cette nouvelle transformation, nous obtenons les deux buts suivants[4] :

```
goal transition_star_transitive_Refl : forall x s3: state,  p: parameter.
  transition_star p x s3 → transition_star p x s3

goal transition_star_transitive_Step : forall x y z s3: state, p: parameter.
  transition p x y →
  transition_star p y z ∧ (transition_star p z s3 → transition_star p y s3) →
  transition_star p z s3 → transition_star p x s3
```

Ces buts sont alors prouvés instantanément par les prouveurs automatiques.

6. Travaux connexes

Nous avons illustré notre approche pour le compilateur d'un langage impératif simple vers une machine virtuelle. Cet exemple n'est pas nouveau dans le domaine de la preuve formelle : la vérification d'un compilateur similaire est décrite dans le développement Coq sur lequel nous nous sommes basés, ou encore dans un développement HOL [9]. À la différence de ce que nous présentons dans cet article, ces développement appliquent une approche plus directe.

L'idée principale de notre approche est de combiner, dans le but de vérifier un compilateur, le plongement d'une logique de programme dans un outil de preuve formelle avec les techniques du calcul de plus faible précondition pour du code non structuré. La méthode du plongement d'une logique de programme a déjà été utilisée dans le but de vérifier des programmes spécifiées via cette logique, par exemple avec Ynot [12]. Les techniques de vérification du code non structuré ont également été étudiées auparavant. On peut notamment citer le travail de Barnett et Leino [1] sur le calcul de plus faible précondition, ou également le développement Bedrock [3]. La combinaison de ces deux techniques dans le cadre de vérification de compilateur a déjà été employée par

4. modulo quelques simplifications triviales pour des raisons de lisibilité.

Myreen [11]. Cette approche a été également envisagée par Jensen et al. dans leur travail sur la vérification du code assembleur x86 [8]. Une différence importante entre ces travaux et le nôtre est l'emploi dans notre cas du calcul de plus faible précondition pour automatiser la preuve.

Les méthodes de preuve de programmes ont été également employées pour la validation d'un compilateur. Il ne s'agit pas dans ce cas de vérifier le compilateur lui-même, mais *a posteriori* le code généré. C'est par exemple l'approche utilisée par l'outil GeneAuto [5].

7. Conclusion et perspectives

Le point de départ de notre expérience était la preuve Coq du compilateur mentionnée en introduction, plus précisément la partie concernant la simulation en avant. Avons-nous gagné en automatisation vis-à-vis de cette preuve ? En termes de la preuve proprement dite, nous avons réussi à la rendre quasi-automatique. En revanche, notre approche induit un coût non négligeable en spécifications. Concrètement, notre développement représente un peu moins de 500 lignes de codes (preuve Coq incluse), ce qui est comparable à la partie correspondante du développement Coq (un peu plus de 400 lignes). Cependant, la formalisation de la logique de Hoare et des spécifications des instructions de la machine virtuelle, qui représente 60% de notre développement, est indépendante du compilateur proprement dit. Elle pourrait être réutilisée telle quelle pour un autre compilateur vers cette même machine, à la différence du développement Coq.

Perspectives Pour la correction du compilateur, nous avons montré un résultat particulier, à savoir la simulation en avant dans le cas des exécutions terminantes. Une extension intéressante serait de prouver la correction pour d'autres types de résultats. Par exemple, il serait possible d'essayer d'appliquer l'approche que nous avons présentée pour les exécutions divergentes. Une autre direction possible serait d'essayer de montrer la simulation en arrière. Nous menons en ce moment une expérience à ce sujet qui suggère qu'il s'agit principalement de changer l'interprétation des triplets de Hoare. Il serait également intéressant d'essayer d'appliquer cette approche à un cas où le schéma de compilation nécessite une traduction non triviale entre les états source et cible.

De manière orthogonale à ces extensions, il serait également intéressant d'enrichir notre bibliothèque de combinateurs. Cela pourrait nous permettre de traiter la compilation d'un langage source plus élaboré, ou un schéma de compilation différent. Par exemple, si l'on étend le langage source avec des procédures, il serait utile d'introduire un combinateur de sélection.

Enfin, la formalisation des triplets de Hoare et des combinateurs semble relativement indépendante du système de transition sous-jacent (dans notre cas, la machine virtuelle). Il serait donc intéressant d'essayer de généraliser cette partie du travail, par exemple sous forme d'une bibliothèque Why3, pour pouvoir ensuite l'appliquer à des systèmes de transitions de nature différente.

Remerciements Nous remercions Jean-Christophe Filliâtre, Claude Marché et Andrei Paskevich pour leurs conseils et leurs suggestions. Nous remercions également les reviewers anonymes pour leurs remarques.

Références

[1] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '05, pages 82–87, New York, NY, USA, 2005. ACM.

[2] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3 : Shepherd your herd of provers. In *Boogie*, pages 53–64, August 2011.

[3] A. Chlipala. Mostly-automated verification of low-level programs in computational separation logic. *SIGPLAN Not.*, 46(6) :234–245, June 2011.

[4] M. Clochard, J.-C. Filliâtre, C. Marché, and A. Paskevich. Formalizing semantics with an automatic program verifier. In D. Giannakopoulou and D. Kroening, editors, *6th Working Conference on Verified Software : Theories, Tools and Experiments (VSTTE)*, volume 8471 of *Lecture Notes in Computer Science*, pages 37–51, Vienna, Austria, July 2014. Springer.

[5] A. Dieumegard and M. Pantel. Vérification d'un générateur de code par génération d'annotations (short paper). In *Conférence en Ingénierie du Logiciel (CIEL), Rennes, France, 19/06/2012-21/06/2012*, page (en ligne), http ://www.irisa.fr/, 2012. IRISA.

[6] J.-C. Filliâtre, L. Gondelman, and A. Paskevich. The spirit of ghost code. In A. Biere and R. Bloem, editors, *26th International Conference on Computer Aided Verification*, volume 8859 of *Lecture Notes in Computer Science*, pages 1–16, Vienna, Austria, July 2014. Springer.

[7] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *ESOP*, volume 7792 of *LNCS*, pages 125–128. Springer, Mar. 2013.

[8] J. B. Jensen, N. Benton, and A. Kennedy. High-level separation logic for low-level code. *SIGPLAN Not.*, 48(1) :301–314, Jan. 2013.

[9] G. Klein, H. Loetzbeyer, T. Nipkow, and R. Sandner. IMP — A WHILE-language and its Semantics, 2008.

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

[11] M. O. Myreen. Verified just-in-time compiler on x86. *SIGPLAN Not.*, 45(1) :107–118, Jan. 2010.

[12] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot : Reasoning with the awkward squad. In *Proceedings of ICFP'08*, 2008.

A. Appendice

A.1. Semantique du langage source

```
function aeval  (st: state) (e: aexpr) : int = match e with
  | Anum n      → n
  | Avar x      → st[x]
  | Aadd e1 e2  → aeval st e1 + aeval st e2
  | Asub e1 e2  → aeval st e1 - aeval st e2
  | Amul e1 e2  → aeval st e1 * aeval st e2
  end

function beval (st:state) (b:bexpr) : bool = match b with
  | Btrue      → true
  | Bfalse     → false
  | Bnot b'    → notb (beval st b')
  | Band b1 b2 → andb (beval st b1) (beval st b2)
  | Beq a1 a2  → aeval st a1 = aeval st a2
  | Ble a1 a2  → aeval st a1 ≤ aeval st a2
  end

inductive ceval (mi: state) (cmd: com) (mf: state) =
  | E_Skip : forall m. ceval m Cskip m
  | E_Ass  : forall m a x. ceval m (Cassign x a) m[x ← aeval m a]
  | E_Seq  : forall cmd1 cmd2 m0 m1 m2.
      ceval m0 cmd1 m1 → ceval m1 cmd2 m2 → ceval m0 (Cseq cmd1 cmd2) m2
  | E_IfTrue : forall m0 m1 cond cmd1 cmd2. beval m0 cond →
      ceval m0 cmd1 m1 → ceval m0 (Cif cond cmd1 cmd2) m1
  | E_IfFalse : forall m0 m1 cond cmd1 cmd2. not beval m0 cond →
      ceval m0 cmd2 m1 → ceval m0 (Cif cond cmd1 cmd2) m1
```

```
  | E_WhileEnd : forall cond m body. not beval m cond →
      ceval m (Cwhile cond body) m
  | E_WhileLoop : forall mi mj mf cond body. beval mi cond →
      ceval mi body mj → ceval mj (Cwhile cond body) mf →
      ceval mi (Cwhile cond body) mf
```

A.2. Semantique du langage cible

```
inductive transition (c: code) (msi msj: machine_state) =
  | trans_const : forall c p n. codeseq_at c p (iconst n) →
      forall s m. transition c (VMS p s m) (VMS (p + 1) (push n s) m)
  | trans_var : forall c p x. codeseq_at c p (ivar x) →
      forall s m. transition c (VMS p s m) (VMS (p + 1) (push m[x] s) m)
  | trans_set_var: forall c p x. codeseq_at c p (isetvar x) →
      forall n s m. transition c (VMS p (push n s) m) (VMS (p + 1) s m[x←n])
  | trans_add : forall c p. codeseq_at c p iadd →
      forall n1 n2 s m. transition c (VMS p (push n2 (push n1 s)) m)
                                     (VMS (p + 1) (push (n1 + n2) s) m)
  | trans_sub : forall c p. codeseq_at c p isub →
      forall n1 n2 s m. transition c (VMS p (push n2 (push n1 s)) m)
                                     (VMS (p + 1) (push (n1 - n2) s) m)
  | trans_mul : forall c p. codeseq_at c p imul →
      forall n1 n2 s m. transition c (VMS p (push n2 (push n1 s)) m)
                                     (VMS (p + 1) (push (n1 * n2) s) m)
  | trans_beq: forall c p1 ofs. codeseq_at c p1 (ibeq ofs) →
      forall s m n1 n2. transition c (VMS p1 (push n2 (push n1 s)) m)
                  (VMS (if n1 = n2 then p1 + 1 + ofs else p1 + 1) s m)
  | trans_bne: forall c p1 ofs. codeseq_at c p1 (ibne ofs) →
      forall s m n1 n2. transition c (VMS p1 (push n2 (push n1 s)) m)
                   (VMS (if n1 = n2 then p1 + 1 else p1 + 1 + ofs) s  m)
  | trans_ble: forall c p1 ofs. codeseq_at c p1 (ible ofs) →
      forall s m n1 n2. transition c (VMS p1 (push n2 (push n1 s)) m)
                  (VMS (if n1 ≤ n2 then p1 + 1 + ofs else p1 + 1)  s m)
  | trans_bgt: forall c p1 ofs. codeseq_at c p1 (ibgt ofs) →
      forall s m n1 n2. transition c (VMS p1 (push n2 (push n1 s)) m)
                  (VMS (if n1 ≤ n2 then p1 + 1 else p1 + 1 + ofs) s  m)
  | trans_branch: forall c p ofs. codeseq_at c p (ibranch ofs) →
      forall s m. transition c (VMS p s m) (VMS (p + 1 + ofs) s m)
```

A.3. Compilateur

```
function aexpr_post (a:aexpr) (len:pos) : post 'a =
  \x p ms ms'. let VMS _ s m = ms in ms' = VMS (p+len) (push (aeval m a) s) m
meta rewrite_def function aexpr_post

let rec compile_aexpr (a:aexpr) :  hl 'a
  ensures { result.pre = trivial_pre ∧ hl_correctness result }
  ensures { result.post = aexpr_post a result.code.length }
  variant { a }
  = let c = match a with
    | Anum n     → $ iconstf n
    | Avar x     → $ ivarf x
    | Aadd a1 a2 → $ compile_aexpr a1 ~ $ compile_aexpr a2 ~  $ iaddf ()
    | Asub a1 a2 → $ compile_aexpr a1 ~ $ compile_aexpr a2 ~  $ isubf ()
    | Amul a1 a2 → $ compile_aexpr a1 ~ $ compile_aexpr a2 ~  $ imulf ()
    end in
```

```
    hoare trivial_pre c (aexpr_post a c.wcode.length)

function bexpr_post (b:bexpr) (cond: bool) (out_t:ofs) (out_f:ofs) : post 'a =
  \x p ms ms'. let VMS _ s m = ms in if beval m b = cond
      then ms' = VMS (p + out_t) s m
      else ms' = VMS (p + out_f) s m
meta rewrite_def function bexpr_post

function exec_cond (b1:bexpr) (cond:bool) : pre 'a =
  \x p ms. let VMS _ _ m = ms in beval m b1 = cond
meta rewrite_def function exec_cond

let rec compile_bexpr (b:bexpr) (cond:bool) (ofs:ofs) :  hl 'a
  ensures { result.pre = trivial_pre ∧ hl_correctness result }
  ensures { result.post =
            bexpr_post b cond (result.code.length + ofs) result.code.length }
  variant { b }
= let c = match b with
  | Btrue      → $ if cond then ibranchf ofs else inil ()
  | Bfalse     → $ if cond then inil () else ibranchf ofs
  | Bnot b1    → $ compile_bexpr b1 (not cond) ofs
  | Band b1 b2 →
    let c2  = $ compile_bexpr b2 cond ofs % exec_cond b1 true in
    let ofs = if cond then length c2.wcode else ofs + length c2.wcode in
    $ compile_bexpr b1 false ofs ~ c2
  | Beq a1 a2 → $ compile_aexpr a1 ~ $ compile_aexpr a2 ~
                $ if cond then ibeqf ofs else ibnef ofs
  | Ble a1 a2 → $ compile_aexpr a1 ~ $ compile_aexpr a2 ~
                $ if cond then iblef ofs else ibgtf ofs
  end in
  let ghost post = bexpr_post b cond (c.wcode.length + ofs) c.wcode.length in
  hoare trivial_pre c post

function com_pre (cmd:com) : pre 'a =
  \x p ms. let VMS p' _ m = ms in p = p' ∧ exists m'. ceval m cmd m'
meta rewrite_def function com_pre

function com_post (cmd:com) (len:pos) : post 'a =
  \x p ms ms'. let VMS p s m = ms in let VMS p' s' m' = ms' in
    p' = p + len ∧ s' = s ∧ ceval m cmd m'
meta rewrite_def function com_post

function exec_cond_old (b1:bexpr) (cond:bool) : pre ('a,machine_state) =
  \x p ms. let VMS _ _ m = snd x in beval m b1 = cond
meta rewrite_def function exec_cond_old

(* Invariant for loop compilation: any intermediate state
   would evaluate to the same final state as the initial state. *)
function loop_invariant (c:com) : pre ('a,machine_state) =
  \x p msi. let VMS _ s0 m0 = snd x in let VMS pi si mi = msi in
    pi = p ∧ s0 = si ∧ exists mf. ceval m0 c mf ∧ ceval mi c mf
meta rewrite_def function loop_invariant

function loop_post (c : com) (len: pos) : pre ('a,machine_state) =
  \x p msf. let VMS _ s0 m0 = snd x in let VMS pf sf mf = msf in
    pf = p + len ∧ s0 = sf ∧ ceval m0 c mf
```

```
meta rewrite_def function loop_post

function loop_variant (c:com) (test:bexpr) : post 'a =
  \x p msj msi. let VMS pj sj mj = msj in let VMS pi si mi = msi in
     pj = pi ∧ sj = si ∧ ceval mi c mj ∧ beval mi test
meta rewrite_def function loop_variant

let rec compile_com (cmd: com) : hl 'a
  ensures  { result.pre = com_pre cmd ∧ hl_correctness result }
  ensures  { result.post = com_post cmd result.code.length }
  variant  { cmd }
= let res = match cmd with
  | Cskip              → $ inil ()
  | Cassign x a        → $ compile_aexpr a  ~ $ isetvarf x
  | Cseq cmd1 cmd2     → $ compile_com cmd1 ~ $ compile_com cmd2
  | Cif cond cmd1 cmd2 → let code_false = compile_com cmd2 in
    let code_true = $ compile_com cmd1 ~ $ ibranchf code_false.code.length in
    $ compile_bexpr cond false code_true.wcode.length ~
    (code_true % exec_cond cond true) ~
    ($ code_false % exec_cond_old cond false)
  | Cwhile test body  → let code_body = compile_com body in
    let body_length = length code_body.code + 1 in
    let code_test = compile_bexpr test false body_length in
    let ofs = length code_test.code + body_length in
    let wp_while = $ code_test ~
        ($ code_body ~ $ ibranchf (- ofs)) % exec_cond test true in
    let ghost inv  = loop_invariant cmd in
    let ghost var  = loop_variant body test  in
    let ghost post = loop_post cmd ofs in
    let hl_while = hoare inv wp_while (loop_progress inv post var) in
    $ inil () ~ $ make_loop_hl hl_while inv post var
  end in
  hoare (com_pre cmd) res (com_post cmd res.wcode.length)

(* Get back to natural specification for the compiler. *)
let compile_com_natural (com: com) : code
  ensures { forall c p s m m'. ceval m com m' → codeseq_at c p result →
    transition_star c (VMS p s m) (VMS (p + length result) s m') }
= let res = compile_com com : hl unit in
  assert { forall c p s m m'. ceval m com m' → codeseq_at c p res.code →
    res.pre () p (VMS p s m) && (forall ms'. res.post () p (VMS p s m) ms' →
    ms' = VMS (p + length res.code) s m') };
  res.code

(* Insert the final halting instruction. *)
let compile_program (prog : com) : code
  ensures { forall  mi mf: state.
    ceval mi prog mf → vm_terminates result mi mf }
= compile_com_natural prog ++ ihalt
```