# Calcul de plus faible précondition, revisité en Why3

Claude Marché, Asma Tafat

▶ **To cite this version:**

Claude Marché, Asma Tafat. Calcul de plus faible précondition, revisité en Why3. JFLA - Journées francophones des langages applicatifs - 2013, Damien Pous and Christine Tasson, Feb 2013, Aussois, France. hal-00778791

**HAL Id: hal-00778791**

**https://inria.hal.science/hal-00778791v1**

Submitted on 21 Jan 2013

**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.

# Calcul de plus faible précondition, revisité en Why3

Claude Marché[1,2] & Asma Tafat[2,1]

[1] *INRIA Saclay – Île-de-France &* [2] *LRI / CNRS – Université Paris Sud*
*Centre Universitaire d'Orsay, Bâtiment 650 (PCRI), 91405 Orsay Cedex, France*

**Résumé**

Cet article a deux objectifs. D'une part, nous présentons une méthode originale de preuve de correction d'un calcul de plus faible précondition, fondée sur la notion de *sémantique bloquante*. La méthode imite au niveau des spécifications logiques la méthode classique de preuve de *type soundness*. D'autre part, cette preuve est réalisée formellement dans l'environnement de vérification déductive Why3, et nous illustrons, au fur et à mesure de l'article, les fonctionnalités avancées de cet environnement que nous avons utilisées. Le résultat constitue une présentation revisitée du calcul de plus faible précondition, et qui, bien qu'elle soit réalisée formellement, est facile à suivre, grâce en particulier au haut degré d'automatisation des preuves qui permet de se focaliser sur les points clés.

## 1. Introduction

Le système Why3 [7] est un environnement pour la *vérification déductive de programmes* : il permet de coder des programmes en les annotant avec des spécifications logiques, puis de générer les obligations de preuves qui garantissent leur correction fonctionnelle. Why3 peut ensuite soumettre ces obligations à une large collection de prouveurs : automatiques comme les *solveurs SMT* (Alt-Ergo [5], CVC3 [3], Z3 [9], etc.) ou interactifs comme les environnements de preuve Coq [4] ou PVS [12].

Dans le paysage des outils pour la vérification déductive, Why3 se veut à mi-chemin entre les environnements interactifs offrant des langages très expressifs mais un niveau faible d'automatisation des preuves (Coq, PVS, Isabelle/HOL, etc.), et les systèmes plus automatiques mais munis de langages plus pauvres (Spec#, VCC, Frama-C, KeY, etc.). Un objectif naturel est de récupérer le meilleur possible des deux extrêmes : un langage suffisamment expressif qui permet des preuves largement automatiques.

Un premier objectif de cet article est d'illustrer les capacités de Why3 sur une étude de cas, mettant en œuvre des objets complexes comme les types algébriques récursifs et les définitions inductives, de façon à illustrer la mise en place de l'automatisation des preuves dans un tel contexte. Une famille d'exemples mettant en œuvre de tels objets est naturellement formée des méthodes de calcul symbolique, en particulier les outils travaillant sur des langages : compilateurs, analyseurs statiques, etc. Notre choix s'est porté sur la formalisation d'un calcul de plus faible précondition, suite aux travaux de Herms *et al.* [10] sur la certification d'un générateur d'obligations de preuve avec Coq. Ces travaux sont fondés sur une notion de *sémantique bloquante* à grands pas, définie co-inductivement. Le second objectif de cet article est de présenter une approche originale du calcul de plus faible précondition. L'originalité réside dans la proposition d'une sémantique bloquante *à petits pas*, qui plus est formalisée dans le langage de Why3, significativement plus restreint que celui de Coq.

En plus des types algébriques et des prédicats inductifs, les fonctionnalités avancées de Why3 que nous allons utiliser concernent les méthodes fournies pour faire les preuves. Ainsi, nous allons utiliser

Cette étude a été partiellement financée par le projet U3CAT (`http://frama-c.com/u3cat/`) de l'Agence Nationale de la Recherche et le projet Hi-lite (`http://www.open-do.org/projects/hi-lite/`) du pôle de compétitivité Systematic de la région Île-de-France.

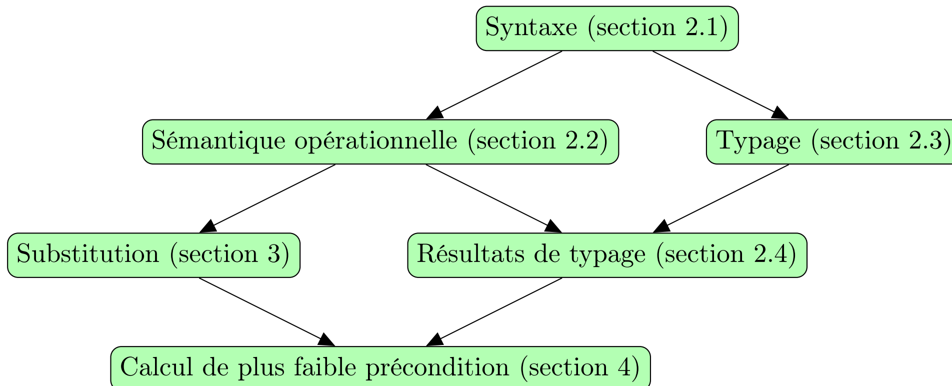


Figure 1 – Structure du développement

une *transformation* de Why3 qui permet de faire de la récurrence structurelle. Quand les énoncés seront trop complexes pour être prouvés automatiquement, nous appelons Coq, et depuis Coq nous utilisons une *tactique* « why3 » qui permet de terminer la preuve d'un sous-but Coq en rappelant des prouveurs automatiques à travers Why3. Ces fonctionnalités permettent au final de ne faire en Coq que les parties subtiles des preuves.

L'exemple que nous présentons est structuré en composants, nommés *théories* dans Why3, et l'article suit la même structure, représentée sur la figure 1. La section 2 définit le langage d'étude : sa syntaxe (section 2.1), sa sémantique opérationnelle (section 2.2) et ses règles de typage (section 2.3). La section 2.4 énonce et prouve les résultats classiques de préservation du typage par réduction. La section 3 est consacrée à la notion de substitution et à la problématique importante des variables fraîches. La section 4 présente alors le calcul de plus faible précondition, elle énonce et prouve les résultats de correction.

# 2. Formalisation d'un langage impératif jouet

Cette section formalise notre langage jouet qui *grosso-modo* ne contient que l'affectation, la séquence, une instruction conditionnelle et une boucle « tant que ».

## 2.1. Syntaxe

La première théorie que nous définissons décrit la syntaxe de notre langage. Traditionnellement, on définit cette syntaxe par des grammaires. Ici nous donnons directement les définitions Why3 sous forme de déclarations de types algébriques. Nous ne pouvons pas présenter la syntaxe de Why3 en détail dans cet article ; on pourra se référer au manuel [6] ou au cours de J.-C. Filliâtre aux JFLA 2012 [1]. La syntaxe étant proche de celle d'OCaml, nous espérons qu'elle sera suffisamment intuitive pour le lecteur.

Notre langage est muni de trois types de base qui correspondent à trois ensembles de valeurs.

```
type datatype = TYunit | TYint | TYbool      (** basic types unit, int and bool *)
type value = Vvoid | Vint int | Vbool bool   (** corresponding sets of values *)
```

Nous introduisons des types *abstraits* pour les identificateurs, en distinguant les identificateurs de variables mutables, de ceux des variables logiques.

1. http://why3.lri.fr/jfla-2012/

```
type mident  (** identifiers for mutable variables *)
type ident   (** identifiers for logic variables *)
```

Le langage des termes de la logique contient les valeurs, les variables logiques, la déréférenciation des variables mutables, et les opérations binaires $+$, $-$, $\times$ et $\leq$ (d'autres opérateurs pourraient être ajoutés sans difficulté supplémentaire).

```
type operator = Oplus | Ominus | Omult | Ole  (** operators +, -, *, ≤ *)
type term =                                   (** Terms *)
  | Tvalue value                              (** values *)
  | Tvar ident                                (** logic variables *)
  | Tderef mident                             (** access to mutable variables *)
  | Tbin term operator term                   (** binary operations *)
```

Le langage des formules contient les termes (booléens), les connecteurs classiques de conjonction, négation et implication, et la quantification universelle. Ce sont les seuls connecteurs dont on a besoin pour la suite, mais d'autres pourraient être ajoutés simplement. On ajoute également la liaison locale par `let` qui permettra une définition plus élégante du calcul de plus faible précondition. Notons ici que nous n'utilisons aucune méthode avancée comme les indices de De Bruijn. Les difficultés propres aux noms de variables seront traitées à la section 3.

```
type fmla =                        (** Formulas *)
  | Fterm term                     (** terms: atomic formulas *)
  | Fand fmla fmla                 (** conjunction *)
  | Fnot fmla                      (** negation *)
  | Fimplies fmla fmla             (** implication *)
  | Flet ident term fmla           (** local binding: let id = term in fmla *)
  | Fforall ident datatype fmla   (** universal quantification: forall id : ty, fmla *)
```

Enfin, la syntaxe des instructions de notre langage impératif est la suivante.

```
type stmt =                  (** Statements *)
  | Sskip                    (** no-op statement *)
  | Sassign mident term      (** assignment id := term *)
  | Sseq stmt stmt           (** sequence *)
  | Sif term stmt stmt       (** conditional statement *)
  | Sassert fmla             (** assertion statement *)
  | Swhile term fmla stmt    (** while loop with condition, invariant and body *)
```

Nous considérons l'affectation d'une variable mutable par un terme, la séquence (la constante `skip` représentant une séquence vide), l'instruction conditionnelle standard et la boucle *tant que*. De manière non classique, nous ajoutons directement dans la syntaxe des annotations logiques : invariant explicite pour les boucles, ainsi qu'une instruction d'affirmation (*assert* en anglais).

Cette première théorie ne contient que des définitions et pas de lemmes ; il n'y a donc rien à prouver.

## 2.2. Sémantique opérationnelle

Notre seconde théorie a pour but de définir une sémantique opérationnelle pour notre langage. Nous choisissons une version à petits pas. La première étape consiste à définir les environnements d'évaluation : pour les variables mutables d'une part et pour les variables logiques d'autre part.

Le type `env` dénote les environnements pour les variables mutables. C'est une table d'association qui à chaque `mident` associe une valeur. On utilise pour cela le type générique `map` de la bibliothèque standard de Why3.

```
use export Syntax
use map.Map as IdMap
type env = IdMap.map mident value     (** map global mutable variables to their value *)
function get_env (i:mident) (sigma:env) : value = IdMap.get sigma i
```

La fonction `get_env` fournit un raccourci pour accéder aux éléments d'un tel environnement.

Le type `stack` dénote les environnements pour les variables logiques : une pile de couples (`ident`, `value`). On utilise pour cela le type générique `list` de la bibliothèque standard de Why3.

```
use export list.List
type stack = list (ident, value)     (** map local immutable variables to their value *)
function get_stack (i:ident) (pi:stack) : value =
  match pi with
  | Nil → Vvoid
  | Cons (x,v) r → if x=i then v else get_stack i r
  end
```

La fonction `get_stack` fournit un raccourci pour accéder à un tel environnement ; elle est définie par récurrence sur la liste. Noter que cette fonction renvoie `void` si l'on accède à une variable absente de la liste. Ce cas ne se produira jamais sur des programmes bien typés tels que définis dans la section suivante. Enfin, on pose des lemmes, prouvés automatiquement, montrant un comportement similaire à celui des tables d'association.

```
lemma get_stack_eq: forall x:ident, v:value, pi:stack.
  get_stack x (Cons (x,v) pi) = v
lemma get_stack_neq: forall x i:ident, v:value, pi:stack.
  x ≠ i → get_stack i (Cons (x,v) pi) = get_stack i pi
```

Nous définissons ensuite une fonction auxiliaire d'évaluation des opérations binaires. Cette définition se fait par cas sur les valeurs. Comme en Why3 les fonctions doivent être totales, on complète la définition en renvoyant `void` dans les cas « mal typés » [2].

```
function eval_bin (x:value) (op:operator) (y:value) : value =
  match x,y with
  | Vint x,Vint y →
     match op with
     | Oplus  → Vint (x+y)
     | Ominus → Vint (x-y)
     | Omult  → Vint (x*y)
     | Ole    → Vbool (if x ≤ y then True else False)
     end
  | _,_ → Vvoid
  end
```

Nous pouvons maintenant définir l'évaluation des termes comme une fonction récursive totale retournant une `value`, et l'évaluation des formules par un prédicat récursif total. Ces deux fonctions prennent en argument un *état courant* d'un programme, donné par deux environnements $\Sigma$ et $\Pi$.

```
function eval_term (sigma:env) (pi:stack) (t:term) : value = match t with
  | Tvalue v       → v
  | Tvar id        → get_stack id pi
  | Tderef id      → get_env id sigma
  | Tbin t1 op t2  → eval_bin (eval_term sigma pi t1) op (eval_term sigma pi t2)
  end
predicate eval_fmla (sigma:env) (pi:stack) (f:fmla) = match f with
  | Fterm t            → eval_term sigma pi t = Vbool True
  | Fand f1 f2         → eval_fmla sigma pi f1 ∧ eval_fmla sigma pi f2
  | Fnot f             → not (eval_fmla sigma pi f)
  | Fimplies f1 f2     → eval_fmla sigma pi f1 → eval_fmla sigma pi f2
  | Flet x t f         → eval_fmla sigma (Cons (x,eval_term sigma pi t) pi) f
  | Fforall x TYint f  → forall n:int. eval_fmla sigma (Cons (x,Vint n) pi) f
  | Fforall x TYbool f → forall b:bool. eval_fmla sigma (Cons (x,Vbool b) pi) f
  | Fforall x TYunit f → eval_fmla sigma (Cons (x,Vvoid) pi) f
  end
```

2. Noter que `True` et `False` sont les deux constructeurs du type bool de Why3. Nous écrivons `if x ≤ y then True else False` et non pas simplement `x ≤ y` pour "transformer" cette proposition en booléen.

Noter que dans le cas des lieurs `Flet` et `Fforall`, la variable liée `x` est ajoutée à la pile. Dans la suite, on notera ces fonctions sous la forme $[\![t]\!]_{\Sigma,\Pi}$ et $[\![f]\!]_{\Sigma,\Pi}$.

Enfin, nous définissons la notion de formule *valide*, au sens où elle est vraie dans n'importe quel état.

```
predicate valid_fmla (p:fmla) = forall sigma:env, pi:stack. eval_fmla sigma pi p
```

L'exécution d'une instruction dans un état donné se définit classiquement par une relation de réduction en un pas, notée $\Sigma, \Pi, s \rightsquigarrow \Sigma', \Pi', s'$. Comme il y a des instructions qui ne réduisent pas, on ne peut pas formaliser cette relation par une fonction de Why3 [3]. Nous utilisons plutôt un *prédicat inductif* de Why3 [4]. Notre définition est alors une formalisation proche de l'écriture habituelle sous forme de règles d'inférence.

```
inductive one_step env stack stmt env stack stmt =
  | one_step_assign : forall sigma sigma':env, pi:stack, x:mident, t:term.
      sigma' = IdMap.set sigma x (eval_term sigma pi t) →
      one_step sigma pi (Sassign x t) sigma' pi Sskip

  | one_step_seq_noskip: forall sigma sigma':env, pi pi':stack, s1 s1' s2:stmt.
      one_step sigma pi s1 sigma' pi' s1'              →
      one_step sigma pi (Sseq s1 s2) sigma' pi' (Sseq s1' s2)

  | one_step_seq_skip: forall sigma:env, pi:stack, s:stmt.
      one_step sigma pi (Sseq Sskip s) sigma pi s

  | one_step_if_true: forall sigma:env, pi:stack, t:term, s1 s2:stmt.
      eval_term sigma pi t = Vbool True                →
      one_step sigma pi (Sif t s1 s2) sigma pi s1

  | one_step_if_false: forall sigma:env, pi:stack, t:term, s1 s2:stmt.
      eval_term sigma pi t = Vbool False               →
      one_step sigma pi (Sif t s1 s2) sigma pi s2

  | one_step_assert: forall sigma:env, pi:stack, f:fmla.
      eval_fmla sigma pi f                             →       (** blocking semantics *)
      one_step sigma pi (Sassert f) sigma pi Sskip

  | one_step_while_true: forall sigma:env, pi:stack, cond:term, inv:fmla, body:stmt.
      eval_fmla sigma pi inv                           →       (** blocking semantics *)
      eval_term sigma pi cond = Vbool True             →
      one_step sigma pi (Swhile cond inv body) sigma pi (Sseq body (Swhile cond inv body))

  | one_step_while_false: forall sigma:env, pi:stack, cond:term, inv:fmla, body:stmt.
        eval_fmla sigma pi inv                         →       (** blocking semantics *)
        eval_term sigma pi cond = Vbool False          →
        one_step sigma pi (Swhile cond inv body) sigma pi Sskip
```

Ainsi, le cas `one_step_while_true` correspond à la règle

$$\frac{[\![inv]\!]_{\Sigma,\Pi} \qquad [\![cond]\!]_{\Sigma,\Pi} = True}{\Sigma, \Pi, \texttt{while}\ cond\ \texttt{invariant}\ inv\ \texttt{do}\ body \quad \rightsquigarrow \quad \Sigma, \Pi, (body;\ \texttt{while}\ cond\ \texttt{invariant}\ inv\ \texttt{do}\ body)}$$

Nous utilisons une sémantique *bloquante* au sens proposé par Herms *et al.* [10] : l'exécution bloque si une assertion invalide est rencontrée, ou si un invariant de boucle n'est pas vérifié au moment où l'on teste la condition de cette boucle (comme indiqué par la règle ci-dessus). L'intérêt d'une telle définition est que l'on définit le fait *qu'un programme respecte ses spécifications* par le fait *qu'il s'exécute sans bloquer* [10], et ceci est valable en particulier pour un programme qui ne se termine pas.

3. Nous rappelons que les fonctions de Why3 sont toujours totales.

4. La notion de prédicat inductif de Why3 est similaire à celles de Coq ou de PVS : c'est la plus petite relation vérifiant les clauses données. Why3 vérifie les conditions habituelles de monotonie pour de telles définitions.

Il faut aussi remarquer qu'aucune des règles de réduction ne modifie la partie $\Pi$ de l'état. Ce serait le cas si on complétait le langage par une instruction « let ». On conserve néanmoins le paramètre $\Pi'$ pour garder la forme classique d'une réduction en un pas.

Un autre prédicat définit l'exécution en $n$ pas d'un programme, par une clôture réflexive-transitive.

```
inductive many_steps env stack stmt env stack stmt int =
  | many_steps_refl: forall sigma:env, pi:stack, s:stmt.
      many_steps sigma pi s sigma pi s 0
  | many_steps_trans: forall sigma1 sigma2 sigma3:env, pi1 pi2 pi3:stack,
                             s1 s2 s3:stmt, n:int.
      one_step sigma1 pi1 s1 sigma2 pi2 s2 → many_steps sigma2 pi2 s2 sigma3 pi3 s3 n →
      many_steps sigma1 pi1 s1 sigma3 pi3 s3 (n+1)
```

Remarquons que nous rendons explicite le nombre $n$ d'étapes d'exécution dans cette définition, afin par la suite de faire des preuves par récurrence sur $n$. Il faut noter que Why3 propose le type des entiers relatifs, mais pas celui des entiers naturels. Ainsi, pour procéder à des récurrences bien fondées, on pose le lemme élémentaire mais essentiel suivant.

```
lemma steps_non_neg: forall sigma1 sigma2:env, pi1 pi2:stack, s1 s2:stmt, n:int.
  many_steps sigma1 pi1 s1 sigma2 pi2 s2 n → n ≥ 0
```

Ce lemme doit être prouvé en Coq depuis l'environnement Why3 : il nécessite en effet une induction sur le prédicat `many_steps`. Mais cette preuve est très courte (une seule ligne) : (`induction 1; auto with zarith`).

Pour finir cette théorie, on définit la notion de réductibilité d'un programme dans un état donné.

```
predicate reducible (sigma:env) (pi:stack) (s:stmt) =
  exists sigma':env, pi':stack, s':stmt. one_step sigma pi s sigma' pi' s'
```

## 2.3. Typage

Cette théorie introduit le typage des programmes. On commence par une fonction totale qui renvoie le type d'une valeur quelconque.

```
function type_value (v:value) : datatype =
  match v with
    | Vvoid   → TYunit
    | Vint _  →  TYint
    | Vbool _ → TYbool
  end
```

Les types de chacun des opérateurs binaires est donné par un prédicat inductif simple.

```
inductive type_operator (op:operator) (ty1 ty2 ty: datatype) =
  | Type_plus  : type_operator Oplus TYint TYint TYint
  | Type_minus : type_operator Ominus TYint TYint TYint
  | Type_mult  : type_operator Omult TYint TYint TYint
  | Type_le    : type_operator Ole TYint TYint TYbool
```

Nous définissons les environnements de typage de manière très similaire aux environnements d'évaluation.

```
type type_stack = list (ident, datatype)  (** map local immutable variables to their type *)
function get_vartype (i:ident) (pi:type_stack) : datatype =
  match pi with
  | Nil → TYunit
  | Cons (x,ty) r → if x=i then ty else get_vartype i r
  end
type type_env = IdMap.map mident datatype  (** map global mutable variables to their type *)
function get_reftype (i:mident) (e:type_env) : datatype = IdMap.get e i
```

Les jugements de typage des termes, des formules et des instructions sont alors tout naturellement définis par de nouveaux prédicats inductifs.

```
inductive type_term type_env type_stack term datatype =
  | Type_value : forall sigma: type_env, pi:type_stack, v:value.
      type_term sigma pi  (Tvalue v) (type_value v)

  | Type_var : forall sigma: type_env, pi:type_stack, v: ident, ty:datatype.
      (get_vartype v pi = ty) →  type_term sigma pi (Tvar v) ty

  | Type_deref : forall sigma: type_env, pi:type_stack, v: mident, ty:datatype.
      (get_reftype v sigma = ty) → type_term sigma pi (Tderef v) ty

  | Type_bin : forall sigma: type_env, pi:type_stack, t1 t2 : term, op:operator,
      ty1 ty2 ty:datatype.
      type_term sigma pi t1 ty1 → type_term sigma pi t2 ty2 →
      type_operator op ty1 ty2 ty → type_term sigma pi (Tbin t1 op t2) ty
```

```
inductive type_fmla type_env type_stack fmla =
  | Type_term : forall sigma: type_env, pi:type_stack, t:term.
      type_term sigma pi t TYbool → type_fmla sigma pi (Fterm t)

  | Type_conj : forall sigma: type_env, pi:type_stack, f1 f2:fmla.
      type_fmla sigma pi f1 → type_fmla sigma pi f2 → type_fmla sigma pi (Fand f1 f2)

  | Type_neg : forall sigma: type_env, pi:type_stack, f:fmla.
      type_fmla sigma pi f → type_fmla sigma pi (Fnot f)

  | Type_implies : forall sigma: type_env, pi:type_stack, f1 f2:fmla.
      type_fmla sigma pi f1 → type_fmla sigma pi f2 →
      type_fmla sigma pi (Fimplies f1 f2)

  | Type_let : forall sigma: type_env, pi:type_stack, x:ident, t:term, f:fmla, ty:datatype.
      type_term sigma pi t ty → type_fmla sigma (Cons (x,ty) pi) f →
      type_fmla sigma pi (Flet x t f)

  | Type_forall : forall sigma: type_env, pi:type_stack, x:ident, f:fmla, ty:datatype.
      type_fmla sigma (Cons (x,ty) pi) f → type_fmla sigma pi (Fforall x ty f)
```

```
inductive type_stmt type_env type_stack stmt =
  | Type_skip : forall sigma: type_env, pi:type_stack. type_stmt sigma pi Sskip

  | Type_seq : forall sigma: type_env, pi:type_stack, s1 s2:stmt.
      type_stmt sigma pi s1 → type_stmt sigma pi s2 →
      type_stmt sigma pi (Sseq s1 s2)

  | Type_assigns : forall sigma: type_env, pi:type_stack, x:mident, t:term, ty:datatype.
      (get_reftype x sigma = ty) → type_term sigma pi t ty →
      type_stmt sigma pi (Sassign x t)

  | Type_if : forall sigma: type_env, pi:type_stack, t:term, s1 s2:stmt.
      type_term sigma pi t TYbool → type_stmt sigma pi s1 → type_stmt sigma pi s2 →
      type_stmt sigma pi (Sif t s1 s2)

  | Type_assert : forall sigma: type_env, pi:type_stack, p:fmla.
      type_fmla sigma pi p → type_stmt sigma pi (Sassert p)

  | Type_while : forall sigma: type_env, pi:type_stack, cond:term, body:stmt, inv:fmla.
      type_fmla sigma pi inv → type_term sigma pi cond TYbool →
      type_stmt sigma pi body → type_stmt sigma pi (Swhile cond inv body)
```

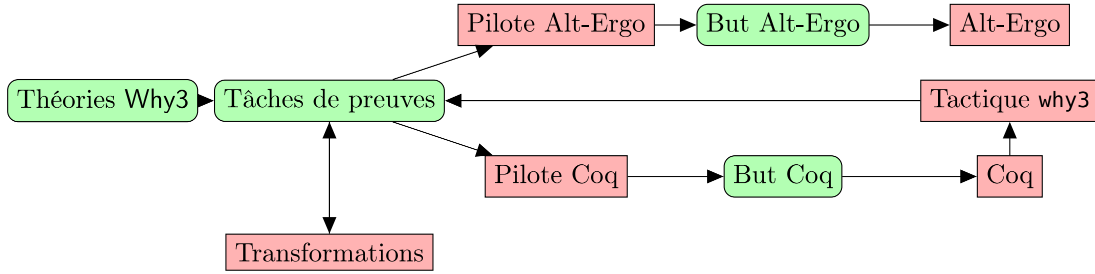


FIGURE 2 – Architecture Why3 et tactique `why3` de Coq.

### 2.4. Relations entre typage et sémantique opérationnelle

Cette théorie contient nos premiers résultats reliant le typage et l'exécution. On définit en premier lieu un prédicat de compatibilité entre environnements de typage et d'évaluation, c'est-à-dire qu'à tout identificateur de type $t$ est associée une valeur de type $t$.

```
predicate compatible_env (sigma:env) (sigmat:type_env) (pi:stack) (pit: type_stack) =
  (forall id: mident. type_value (get_env id sigma) = get_reftype id sigmat) ∧
  (forall id: ident. type_value (get_stack id pi) = get_vartype id pit)
```

On pose ensuite un lemme d'inversion, qui permet de donner le constructeur d'une valeur à partir de son type.

```
lemma type_inversion : forall v:value.
   match (type_value v) with
    | TYbool → exists b: bool. v = Vbool b
    | TYint  → exists n: int. v = Vint n
    | TYunit → v = Vvoid
  end
```

Un raisonnement par cas évident permet de prouver un tel lemme. Malgré tout, les prouveurs automatiques ne sont pas capables de résoudre ce but. On pourrait faire cette preuve en Coq, mais on peut aussi utiliser une fonctionnalité de Why3 : les *transformations* de buts. Ainsi, il existe une transformation pour faire une récursion structurelle. Ici, le type `value` n'étant pas récursif, cela revient à un raisonnement par cas. Les trois sous-buts engendrés sont alors prouvés automatiquement.

Le premier résultat important énonce que tout terme bien typé de type $t$ s'évalue en une valeur de type $t$.

```
lemma eval_type_term:
  forall t:term, sigma:env, pi:stack, sigmat:type_env, pit:type_stack, ty:datatype.
    compatible_env sigma sigmat pi pit →
    type_term sigmat pit t ty          → type_value (eval_term sigma pi t) = ty
```

Ce lemme doit se prouver par récurrence structurelle sur $t$. Là encore, au lieu de faire la preuve entièrement en Coq, on commence par appliquer la transformation Why3 pour la récurrence structurelle, ce qui produit 4 nouveaux sous-buts, correspondant au schéma standard de récursion. Sur ces 4 sous-buts, 3 sont prouvés automatiquement, le seul cas qu'il faut prouver en Coq étant celui des opérations binaires, qui se fait en 5 lignes : il s'agit d'abord d'appliquer le lemme `type_inversion` sur les sous-termes, puis la preuve Coq se termine facilement grâce à une autre fonctionnalité de Why3 : la *tactique* `why3` de Coq. Il s'agit d'une tactique chargée en tant que *plug-in* Coq, et qui permet d'appeler (par l'intermédiaire de Why3) les prouveurs automatiques, selon l'architecture de la figure 2. Il faut remarquer que cette tactique ne produit pas de terme de preuve, il faut donc faire confiance à la fois dans la façon dont elle reconstruit une tâche Why3 à partir du but Coq courant, et dans le prouveur

automatique appelé ensuite, de la même façon que l'on fait confiance aux prouveurs automatiques quand on les appelle directement depuis Why3.

Le second résultat est la préservation du bon typage par réduction.

```
lemma type_preservation : forall s1 s2:stmt, sigma1 sigma2:env, pi1 pi2:stack,
                                 sigmat:type_env, pit:type_stack.
   type_stmt sigmat pit s1 ∧ compatible_env sigma1 sigmat pi1 pit ∧
   one_step sigma1 pi1 s1 sigma2 pi2 s2 →
   type_stmt sigmat pit s2 ∧ compatible_env sigma2 sigmat pi2 pit
```

Ce lemme se prouve en Coq, par induction sur l'hypothèse `one_step`. Il y a 8 sous-cas, dont 7 sont prouvés automatiquement par la tactique `why3`. Le dernier sous-cas, celui de la séquence, nécessite d'inverser l'hypothèse de bon typage de la séquence avant d'appeler la tactique `why3`. Au total cette preuve de préservation du typage par réduction ne fait que trois lignes de Coq !

# 3. Substitutions, variables fraîches

Pour définir notre calcul de plus faible précondition, nous allons avoir besoin de substituer des variables. Nous consacrons une théorie spécifiquement à l'opération de substitution et à ses propriétés, qui s'expriment en particulier sous des hypothèses de fraîcheur des variables. Les problèmes de nommage et de substitution sont classiquement des points difficiles à traiter lorsque l'on formalise des langages avec lieurs, comme l'a montré le fameux défi *POPLmark* [1]. Nous proposons ici une approche basique que nous estimons facile à aborder, et suffisante pour notre étude.

On définit d'abord l'opération de substitution, où dans un premier temps on ne se soucie pas du problème de capture de variable. La seule opération qui nous intéresse ici est de substituer une variable mutable de programme par une variable logique.

```
(** substitution of a mutable variable [x] by a logic variable [v]
    warning: proper behavior only guaranteed if [v] is fresh *)
function msubst_term (t:term) (x:mident) (v:ident) : term =
  match t with
  | Tvalue _ | Tvar _ → t
  | Tderef y          → if x = y then Tvar v else t
  | Tbin t1 op t2     → Tbin (msubst_term t1 x v) op (msubst_term t2 x v)
  end
```

```
function msubst (f:fmla) (x:mident) (v:ident) : fmla =
  match f with
  | Fterm e         → Fterm (msubst_term e x v)
  | Fand f1 f2      → Fand (msubst f1 x v) (msubst f2 x v)
  | Fnot f          → Fnot (msubst f x v)
  | Fimplies f1 f2 → Fimplies (msubst f1 x v) (msubst f2 x v)
  | Flet y t f      → Flet y (msubst_term t x v) (msubst f x v)
  | Fforall y ty f → Fforall y ty (msubst f x v)
  end
```

On définit ensuite la notion de variable *fraîche*, c.-à-d. qui n'apparaît pas dans un terme ou une formule. Ce sont des définitions récursives naturelles.

```
(** [fresh_in_term id t] is true when [id] does not occur in [t] *)
predicate fresh_in_term (id:ident) (t:term) =
  match t with
  | Tvalue _ | Tderef _ → true
  | Tvar i       → id ≠ i
  | Tbin t1 _ t2 → fresh_in_term id t1 ∧ fresh_in_term id t2
  end
```

```
predicate fresh_in_fmla (id:ident) (f:fmla) =
  match f with
  | Fterm e                     → fresh_in_term id e
  | Fand f1 f2 | Fimplies f1 f2 → fresh_in_fmla id f1 ∧ fresh_in_fmla id f2
  | Fnot f                      → fresh_in_fmla id f
  | Flet y t f                  → id ≠ y ∧ fresh_in_term id t ∧ fresh_in_fmla id f
  | Fforall y ty f              → id ≠ y ∧ fresh_in_fmla id f
  end
```

Pour raisonner sur les substitutions dans la suite de notre formalisation, nous avons besoin de quelques lemmes généraux que nous présentons maintenant. Sans surprise, ces lemmes ont des hypothèses de fraîcheur des variables concernées.

Les deux premiers lemmes permettent de ramener l'évaluation d'un terme (resp. d'une formule) sur lequel on a appliqué une substitution, à l'évaluation de ce terme dans un état modifié. Autrement dit on a l'identité $[\![t[x \leftarrow v]]\!]_{\Sigma,\Pi} = [\![t]\!]_{\Sigma[x \leftarrow \Pi(v)],\Pi}$.

```
lemma eval_msubst_term: forall e:term, sigma:env, pi:stack, x:mident, v:ident.
  fresh_in_term v e →
  eval_term sigma pi (msubst_term e x v) =
    eval_term (IdMap.set sigma x (get_stack v pi)) pi e

lemma eval_msubst: forall f:fmla, sigma:env, pi:stack, x:mident, v:ident.
  fresh_in_fmla v f →
  (eval_fmla sigma pi (msubst f x v) ↔ eval_fmla (IdMap.set sigma x (get_stack v pi)) pi f)
```

Ces deux lemmes sont prouvés par récurrence structurelle sur le terme (resp. la formule), grâce à la transformation de Why3. Chacun des sous-buts est prouvé automatiquement, sauf 2 qui doivent être prouvés en Coq (cas `Flet` et `Fforall`), avec des preuves très simples : tactique `simpl` de Coq suivie de l'appel à la tactique `why3`.

Les 2 lemmes suivants permettent, lors de l'évaluation d'un terme ou d'une formule, de permuter l'ordre de deux identificateurs consécutifs dans la pile (s'ils sont différents), c'est-à-dire $[\![t]\!]_{\Sigma,\Pi_1\cdot(id_1,v_1)\cdot(id_2,v_2)\cdot\Pi_2} = [\![t]\!]_{\Sigma,\Pi_1\cdot(id_2,v_2)\cdot(id_1,v_1)\cdot\Pi_2}$. Ces lemmes s'écrivent [5]

```
lemma eval_swap_term: forall t:term, sigma:env, pi l:stack, id1 id2:ident, v1 v2:value.
  id1 ≠ id2 →
  eval_term sigma (l++(Cons (id1,v1) (Cons (id2,v2) pi))) t =
    eval_term sigma (l++(Cons (id2,v2) (Cons (id1,v1) pi))) t

lemma eval_swap_gen:
  forall f:fmla, sigma:env, pi l:stack, id1 id2:ident, v1 v2:value.
    id1 ≠ id2 →
    (eval_fmla sigma (l++(Cons (id1,v1) (Cons (id2,v2) pi))) f ↔
    eval_fmla sigma (l++(Cons (id2,v2) (Cons (id1,v1) pi))) f)
```

et se prouvent de nouveau par récurrence structurelle. Il n'y a qu'un seul sous-but qui n'est pas prouvé automatiquement, celui qui correspond au cas `Tvar` du lemme `eval_swap_term`. Ce cas est effectivement subtil, car il nécessite une nouvelle récurrence, cette fois sur la liste `l`. En Coq, on utilise la tactique `induction l`, et l'on termine les 2 sous-buts avec la tactique `why3`.

On pose également l'instance suivante de `eval_swap_gen` pour le cas `l=Nil`.

```
lemma eval_swap:
  forall f:fmla, sigma:env, pi:stack, id1 id2:ident, v1 v2:value.
    id1 ≠ id2 →
    (eval_fmla sigma (Cons (id1,v1) (Cons (id2,v2) pi)) f ↔
    eval_fmla sigma (Cons (id2,v2) (Cons (id1,v1) pi)) f)
```

5. Le symbole `++` désigne la concaténation des listes.

Bien qu'il suffise d'instancier le lemme précédent, ce lemme n'est pas prouvé automatiquement. Il faut le faire en Coq (en 1 ligne : `apply eval_swap_gen with (l:=Nil)`).

Enfin, les deux derniers lemmes permettent de retirer de la pile un identificateur qui n'apparaît pas dans le terme (resp. la formule) à évaluer, c'est-à-dire $[\![t]\!]_{\Sigma,(id,v)\cdot\Pi} = [\![t]\!]_{\Sigma,\Pi}$ si $id$ est frais.

```
lemma eval_term_change_free : forall t:term, sigma:env, pi:stack, id:ident, v:value.
  fresh_in_term id t → eval_term sigma (Cons (id,v) pi) t = eval_term sigma pi t

lemma eval_change_free : forall f:fmla, sigma:env, pi:stack, id:ident, v:value.
  fresh_in_fmla id f → (eval_fmla sigma (Cons (id,v) pi) f ↔ eval_fmla sigma pi f)
```

De nouveau, la preuve se fait par récurrence structurelle, les seuls sous-cas non prouvés automatiquement étant les cas du `Flet` et du `Fforall`, que l'on prouve en Coq, en utilisant en particulier le lemme `eval_swap`.

# 4. Calcul de la plus faible précondition

## 4.1. Définition du calcul

Le calcul de la plus faible précondition est une fonction qui à une instruction $s$ et une formule $Q$ associe une nouvelle formule $\mathsf{WP}(s, Q)$. Il s'agit d'une définition par récurrence sur $s$. La propriété habituelle attendue est que si l'on exécute $s$ dans un état qui vérifie $\mathsf{WP}(s, Q)$ et que cette exécution se termine, alors $Q$ est valide dans l'état final. En fait, nous énoncerons formellement une propriété plus forte à la section 4.3, sans hypothèse de terminaison.

```
function wp (s:stmt) (q:fmla) : fmla =
  match s with
  | Sskip               → q
  | Sassert f           → Fand f (Fimplies f q)    (** asymmetric and *)
  | Sseq s1 s2          → wp s1 (wp s2 q)
  | Sassign x t         → let id = fresh_from q in Flet id t (msubst q x id)
  | Sif t s1 s2         → (* (t → WP(s1,Q)) ∧ (not t → WP(s2,Q)) *)
      Fand (Fimplies (Fterm t) (wp s1 q)) (Fimplies (Fnot (Fterm t)) (wp s2 q))
  | Swhile cond inv body →
      (* inv ∧ forall effects, (cond ∧ inv → WP(body,inv)) ∧ (not cond ∧ inv → Q) *)
      Fand inv (abstract_effects body
                 (Fand (Fimplies (Fand (Fterm cond) inv) (wp body inv))
                   (Fimplies (Fand (Fnot (Fterm cond)) inv) q)))
  end
```

Notre définition suit le schéma classique d'un tel calcul. Pour le cas `Sassert f`, nous utilisons $f \wedge (f \to Q)$ au lieu de $f \wedge Q$ comme c'est le cas dans les outils en pratique[6]. Pour l'affectation `Sassign x t`, on substitue la variable mutable `x` par un symbole logique frais $id$, calculé par une fonction auxiliaire `fresh_from`, puis on lie localement $id$ à la valeur affectée `t`. Le cas de la boucle introduit classiquement une quantification sur les effets de bord du corps de cette boucle. Nous réalisons cette quantification en appelant une deuxième fonction auxiliaire `abstract_effects`.

Afin de séparer les difficultés, nous n'allons pas détailler d'implémentation de ces fonctions auxiliaires, nous allons seulement les axiomatiser. La première fonction est ainsi axiomatisée par :

```
function fresh_from (f:fmla) : ident
axiom fresh_from_fmla: forall f:fmla. fresh_in_fmla (fresh_from f) f
```

Autrement dit, pour toute formule $f$, `fresh_from f` renvoie un identificateur arbitraire frais dans $f$ ; peu importe comment il est calculé.

La fonction qui abstrait les effets de bord est également axiomatisée. Elle a pour type

6. Permet que $f$ soit en hypothèse des autres obligations de preuve.

```
function abstract_effects (s:stmt) (f:fmla) : fmla
```

et son comportement attendu est, pour une instruction $s$ et une formule $P$ données, de renvoyer la formule $\forall w, f$ où $w = w_1, \ldots, w_k$ est l'ensemble des variables affectées par $s$. Notre axiomatisation de ce comportement est donnée par les quatre hypothèses suivantes.

La première hypothèse formalise la propriété de spécialisation : si $\Sigma, \Pi \models \forall w, f$ alors $\Sigma, \Pi \models f$.

```
axiom abstract_effects_specialize : forall sigma:env, pi:stack, s:stmt, f:fmla.
  eval_fmla sigma pi (abstract_effects s f) → eval_fmla sigma pi f
```

La deuxième hypothèse formalise la distributivité de la quantification sur la conjonction : si $\Sigma, \Pi \models (\forall w, P) \land (\forall w, Q)$ alors $\Sigma, \Pi \models \forall w, P \land Q$.

```
axiom abstract_effects_distrib_conj : forall s:stmt, p q:fmla, sigma:env, pi:stack.
  eval_fmla sigma pi (abstract_effects s p) ∧ eval_fmla sigma pi (abstract_effects s q) →
    eval_fmla sigma pi (abstract_effects s (Fand p q))
```

La troisième hypothèse est plus subtile : elle exprime une propriété de la quantification sur les deux termes d'une implication.

```
axiom abstract_effects_monotonic : forall s:stmt, p q:fmla.
  valid_fmla (Fimplies p q) → forall sigma:env, pi:stack.
    eval_fmla sigma pi (abstract_effects s p) → eval_fmla sigma pi (abstract_effects s q)
```

Autrement dit si $\models P \to Q$ alors $\models (\forall w, P) \to (\forall w, Q)$. Cette propriété peut être vue comme une conséquence de deux propriétés : d'une part si $\models f$ alors $\models \forall w, f$, et d'autre part si $\forall w, (P \to Q)$ alors $(\forall w, P) \to (\forall w, Q)$. Noter qu'il est essentiel de parler de validité dans tous les états dans cette hypothèse, car pour $\Sigma$ et $\Pi$ fixés, $\Sigma, \Pi \models P \to Q$ n'implique pas $\Sigma, \Pi \models (\forall w, P) \to (\forall w, Q)$. Un contre-exemple est : supposons $\Sigma$ tel que $\Sigma(x) = 42$ alors $true \to x = 42$ est vraie, mais pas $(\forall x, true) \to (\forall x, x = 42)$.

Jusqu'à présent, les hypothèses sur `abstract_effects` expriment abstraitement que celle-ci se comporte comme une quantification universelle, sans dire précisément sur quoi. Nous posons maintenant une dernière hypothèse qui indirectement précise sur quoi on quantifie, en posant que si $w = w_1, \ldots, w_k$ est l'ensemble des variables affectées par $s$, alors la formule que l'on abstrait dans notre calcul est *invariante* par le calcul de WP à travers $s$.

```
axiom abstract_effects_writes :
  forall sigma:env, pi:stack, body:stmt, cond:term, inv q:fmla.
    let f =
      (abstract_effects body (Fand
        (Fimplies (Fand (Fterm cond) inv) (wp body inv))
        (Fimplies (Fand (Fnot (Fterm cond)) inv) q)))
    in
    eval_fmla sigma pi f → eval_fmla sigma pi (wp body f)
```

En fait, cette façon abstraite de décrire le comportement de `abstract_effects` est précisément la propriété que l'on cherche à obtenir quand on introduit la quantification universelle dans le calcul de plus faible précondition d'une boucle : la formule $f$ ci-dessus doit être quantifiée, pour devenir vraie (car invariante) à chaque itération de la boucle.

## 4.2. Propriétés de base du calcul

Il existe dans la littérature des études théoriques avancées sur la famille des *transformateurs de prédicats* en général, dont le calcul de plus faible précondition fait partie [2]. Des propriétés sont énoncées souvent sans forcément montrer leur utilité. Pour notre étude, nous allons énoncer et prouver deux propriétés qui servent ensuite à la preuve de correction.

La première de ces propriétés est classiquement nommée *monotonie* du calcul. Elle s'énonce mathématiquement sous la forme : pour toute instruction $s$ et toutes formules $P$ et $Q$, si $\models P \to Q$ alors $\models \mathsf{WP}(s, P) \to \mathsf{WP}(s, Q)$, ce qui donne en Why3 :

```
lemma monotonicity: forall s:stmt, p q:fmla.
  valid_fmla (Fimplies p q) → valid_fmla (Fimplies (wp s p) (wp s q))
```

Notez que, comme précédemment, la quantification sur tous les états est essentielle : $\Sigma, \Pi \models P \to Q$ n'entraîne pas $\Sigma, \Pi \models \mathsf{WP}(s, P) \to \mathsf{WP}(s, Q)$, avec un contre-exemple similaire : si $\Sigma(x) = 42$ alors $(true \to x = 42)$ mais $\mathsf{WP}(x := 7, true) = true$ n'implique pas $\mathsf{WP}(x := 7, x = 42) = (7 = 42)$. La preuve de cette propriété de monotonie se fait par récurrence sur la structure de $s$. En Why3, après application de la transformation de récurrence structurelle, cela donne 6 sous-buts. Seuls les cas de l'affectation, de l'instruction conditionnelle et de la boucle ne sont pas prouvés automatiquement. La preuve du cas `Sassigns` se fait en 2 lignes : la tactique `why3` permet de prouver le but après `unfold valid_fmla; simpl` (la preuve automatique fait appel aux lemmes sur la substitution et les variables fraîches). Le cas du `Sif` est très simple (`simpl` suivi de la tactique `why3`). En revanche le cas du `Swhile` est légèrement plus difficile : il faut utiliser l'hypothèse (3).

La seconde propriété qui va nous servir est la *distributivité sur la conjonction* : si $\Sigma, \Pi \models \mathsf{WP}(s, P)$ et $\Sigma, \Pi \models \mathsf{WP}(s, Q)$ alors $\Sigma, \Pi \models \mathsf{WP}(s, P \wedge Q)$.

```
lemma distrib_conj: forall s:stmt, sigma:env, pi:stack, p q:fmla.
  eval_fmla sigma pi (wp s p) ∧ eval_fmla sigma pi (wp s q) →
  eval_fmla sigma pi (wp s (Fand p q))
```

La réciproque est vraie, mais ne nous servira pas. La preuve se fait là encore par récurrence structurelle sur $s$. Les sous-cas affectation, séquence et boucle ne sont pas prouvés automatiquement. Les preuves se terminent en Coq avec, dans le cas de la boucle, utilisation des hypothèses (3 et (4), et dans le cas de la séquence, utilisation de la propriété précédente de monotonie : cela correspond en effet à une partie « intelligente » de la preuve, où l'on pose comme lemme intermédiaire que $\models \mathsf{WP}(s_1, \mathsf{WP}(s_2, P) \wedge \mathsf{WP}(s_2, Q)) \to \mathsf{WP}(s_1, \mathsf{WP}(s_2, P \wedge Q))$ [11].

## 4.3. Correction du calcul de plus faible précondition

Nous suivons l'approche classique de preuve de *type soundness*. Nous énonçons deux lemmes : d'une part la préservation de la WP par réduction ; d'autre part le *progrès*, c'est-à-dire que la validité de $\mathsf{WP}(s, Q)$ entraîne la réductibilité de $s$ (si $s \neq$ `Sskip`).

```
lemma wp_preserved_by_reduction: forall sigma sigma':env, pi pi':stack, s s':stmt.
  one_step sigma pi s sigma' pi' s' →
  forall q:fmla. eval_fmla sigma pi (wp s q) → eval_fmla sigma' pi' (wp s' q)
```

Ce premier lemme se prouve par induction sur l'hypothèse `(one_step sigma pi s sigma' pi' s')`. Ceci doit se faire en Coq, et après utilisation de la tactique `induction` de Coq, il y a 8 sous-buts, dont 6 peuvent être prouvés avec la tactique `simpl` suivi de la tactique `why3`. Les 2 cas restants concernent la réduction de la boucle. Pour les prouver, il faut commencer par utiliser l'hypothèse (1) pour expliciter le fait que la formule $(cond \wedge inv \to \mathsf{WP}(body, inv)) \wedge (\neg cond \wedge inv \to Q)$ est vraie dans l'état courant, puis utiliser la tactique `why3` pour terminer ces 2 sous-cas. À noter que cet appel à la tactique `why3` utilise l'hypothèse (4) et la propriété `distrib_conj`.

Le lemme de progrès s'énonce comme suit.

```
lemma progress: forall s:stmt, sigma:env, pi:stack,
  sigmat: type_env, pit: type_stack, q:fmla.
    compatible_env sigma sigmat pi pit ∧ type_stmt sigmat pit s ∧
    eval_fmla sigma pi (wp s q) ∧ s ≠ Sskip → reducible sigma pi s
```

Ce lemme se prouve par récurrence structurelle sur $s$. Seul le sous-cas `Sskip` est ensuite prouvé automatiquement, la raison étant que la conclusion `reducible` est quantifiée existentiellement. Il faut alors dans chaque cas passer en Coq et conclure en quelques lignes avec la tactique `exists`. Pour le cas de la boucle, nous utilisons une fois de plus la propriété `distrib_conj`.

Il peut paraître étrange que ce lemme de progrès mentionne une formule $Q$ quelconque, qui ne joue aucun rôle dans la conclusion. Ainsi, il pourrait sembler suffisant d'énoncer ce lemme pour $Q = true$. Le problème serait qu'il ne serait pas assez général pour être prouvable par récurrence. Ainsi, pour prouver le progrès d'une séquence $s_1; s_2$ sachant que $\mathsf{WP}(s_1; s_2, true)$, on se ramène au progrès de $s_1$ sachant $\mathsf{WP}(s_1, \mathsf{WP}(s_2, true))$ et la formule $\mathsf{WP}(s_2, true)$ est *a priori* quelconque.

Le résultat principal est maintenant le suivant : pour tout programme $s$, tout état $\Sigma, \Pi$ et toute formule $Q$, si $\Sigma, \Pi \models \mathsf{WP}(s, Q)$ alors soit $s$ va s'exécuter indéfiniment, soit il se réduit finiment vers `Sskip`, et alors dans ce cas $Q$ est vraie dans l'état final. Comme il n'est pas pratique d'énoncer le fait que l'exécution est infinie, on formule cela en disant que si $s$ s'exécute en un nombre quelconque de pas jusqu'à $s'$ et que $s'$ ne peut plus se réduire, alors $s' = \mathtt{Sskip}$. Tout ceci devant être énoncé uniquement pour un programme bien typé, nous obtenons en Why3 :

```
lemma wp_soundness: forall n :int, sigma sigma':env, pi pi':stack, s s':stmt,
  sigmat: type_env, pit: type_stack, q:fmla.
    compatible_env sigma sigmat pi pit ∧ type_stmt sigmat pit s ∧
    many_steps sigma pi s sigma' pi' s' n ∧ not (reducible sigma' pi' s') ∧
    eval_fmla sigma pi (wp s q) → s' = Sskip ∧ eval_fmla sigma' pi' q
```

Ce résultat se prouve par récurrence sur $n$, ce que l'on effectue en appelant Coq. Pour $n = 0$, le lemme de progrès garantit que $s'$ ne peut être que `Sskip`. Dans le cas $n > 0$, on sait que $s$ se réduit en un pas vers un $s_0$, le lemme de préservation de la WP par réduction permettant d'appliquer l'hypothèse de récurrence sur $s_0$.

Au total, l'ensemble des preuves que nous avons dû faire en Coq représente 142 lignes de tactiques [7]. Ce nombre est à rapprocher du nombre de lignes de spécification que nous avons écrites : 390, soit plus de 2 fois plus. On voit que le haut degré d'automatisation offert par l'environnement Why3 permet d'infirmer la croyance habituelle qu'une preuve de comportement complexe d'un programme nécessite significativement plus de lignes de preuve que de lignes de code.

## 5. Conclusions

Nous avons présenté une technique de preuve de correction d'un calcul de plus faible précondition, qui imite la technique classique de preuve de *type soundness* d'un langage, en montrant à la fois que la WP est préservée par réduction et que la validité de la WP entraîne la réductibilité. Comme la satisfaction des annotations d'un programme est par définition l'absence de blocage (sémantique bloquante), nous obtenons une méthode de vérification qui est garantie correcte, y compris sur des programmes non terminants. La définition par sémantique bloquante est proposée par Herms *et al.* [10] pour le cas d'une sémantique à grands pas, la correction du calcul étant prouvée par co-induction. Nous utilisons une sémantique bloquante sur une sémantique à petit pas : notre approche est donc originale à notre connaissance. Une démarche de preuve similaire a été utilisée par Conchon et Filliâtre [8] dans un contexte différent : preuve de correction d'une procédure de décision de la sûreté d'utilisation de structures de données semi-persistantes.

La preuve obtenue par cette approche n'est pas particulièrement difficile, en fait c'est plutôt le contraire : nous prétendons que c'est une approche naturelle, simple à comprendre et qui plus est facilement automatisable. C'est une méthode que nous prévoyons d'utiliser pour enseigner les bases

7. Les détails des preuves peuvent être consultés dans un rapport de recherche [11] ainsi que sur la galerie de programmes prouvés Toccata`http://toccata.lri.fr/gallery/WP_revisited.en.html`.

de la preuve de programmes [8].

Au cours de cette étude, nous avons identifié plusieurs fonctionnalités intéressantes de l'environnement Why3 : structuration en théories, types algébriques, prédicats inductifs, appels directs de prouveurs automatiques, transformation pour prouver par récurrence structurelle, appel à Coq pour les preuves difficiles, utilisation de la tactique `why3` pour terminer les preuves Coq. Au final, le nombres de lignes de preuves manuelles est petit : 142 lignes de Coq pour 390 lignes de code, ce qui représente un degré très satisfaisant d'automatisation. Néanmoins, il faut remarquer qu'au cours du développement de cet exemple, nous avons dû faire provisoirement plus de preuves Coq, afin d'identifier les bons lemmes pour arriver à remplacer certaines preuves par des preuves automatiques. La tactique `why3` est d'ailleurs très utile dans une telle phase de développement des preuves.

Nous avons identifié aussi quelques faiblesses de Why3 qui feraient l'objet d'extensions intéressantes. Les prouveurs automatiques sont assez faibles à partir du moment où une hypothèse met en jeu un prédicat inductif : une transformation pour inverser une telle hypothèse, ou même raisonner dessus par induction, serait très utile. Par exemple, un lemme comme `steps_non_neg` pourrait être prouvé sans appeler Coq. Dans le même ordre d'idée, une transformation pour prouver un lemme par récurrence sur les entiers serait utile.

On a remarqué une faiblesse assez générale des prouveurs SMT : régulièrement ils ne prouvent pas un but qui se prouve en quelques lignes de Coq. Par exemple, ces prouveurs ne savent pas faire aussi bien qu'un simple `eexists; eauto` de Coq. Nous avons testé également des prouveurs spécialisés pour le premier ordre (type « TPTP » : Vampire [13], Spass [15], Eprover [14]) sans succès. Nous avons aussi remarqué que les prouveurs automatiques ne savent pas traiter des buts qui se prouvent pourtant très facilement avec la tactique `simpl` de Coq suivi d'un nouvel appel à la tactique `why3`. Ceci met en évidence une faiblesse dans le support des calculs par les prouveurs SMT.

Un manque d'un autre ordre est la possibilité de poser des énoncés sur des fonctions du langage de programmation de Why3. En effet, dans cette étude nous n'avons défini que des fonctions pures dans la logique de Why3, pas dans son langage de programmation. Or si une fonction comme `wp` était écrite dans le langage de programmation, alors on ne pourrait parler de son comportement uniquement dans sa post-condition, et donc il ne serait pas possible d'énoncer une propriété comme `wp_soundness`.

L'étude elle-même mériterait d'être étendue. En premier lieu, les fonctions `fresh_from` et `abstract_effects` mériteraient d'être réalisées. L'extension la plus significative serait de traiter un langage avec plusieurs fonctions.

À plus long terme, on peut sérieusement penser à utiliser cette approche pour développer du code correct par construction. En effet, une extension de Why3 actuellement en cours de réalisation est la possibilité d'extraire du code OCaml. Par rapport à une approche toute en Coq, nous gagnerions un haut degré d'automatisation.

**Remerciements** Les auteurs tiennent à remercier Jean-Christophe Filliâtre et Paolo Herms pour l'idée de la sémantique bloquante pour définir la validité des annotations, Levs Gondelmans qui a réalisé la transformation de récurrence structurelle de Why3, Jean-Christophe Filliâtre et Andrei Paskevich pour la réalisation de la tactique `why3` de Coq, ainsi que les autres auteurs de Why3, Guillaume Melquiond et François Bobot, sans qui cette étude n'aurait pas pu être menée à bien. Enfin, nous remercions Jean-Christophe ainsi que Xavier Urbain pour leur relecture extrêmement attentive de cet article, aussi bien sur le fond que sur la forme grammaticale et typographique.

8. Voir `http://www.lri.fr/~marche/MPRI-2-36-1/`

# Références

[1] B. Aydemir, A. Bohannon, M. Fairbairn, J. Foster, B. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses : The POPLmark challenge. In *Proceedings of the Eighteenth International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005)*, number 3603 in Lecture Notes in Computer Science, pages 50–65. Springer, 2005.

[2] R.-J. Back and J. von Wright. *Refinement calculus - a systematic introduction.* Undergraduate texts in computer science. Springer, 1999.

[3] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302, Berlin, Germany, July 2007. Springer.

[4] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development.* Springer-Verlag, 2004.

[5] F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.

[6] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 platform, version 0.80.* LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.80 edition, Oct. 2012. `https://gforge.inria.fr/docman/view.php/2990/8186/manual-0.80.pdf`.

[7] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.

[8] S. Conchon and J.-C. Filliâtre. Semi-Persistent Data Structures. In *17th European Symposium on Programming (ESOP'08)*, Budapest, Hungary, Apr. 2008.

[9] L. de Moura and N. Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[10] P. Herms, C. Marché, and B. Monate. A certified multi-prover verification condition generator. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software : Theories, Tools, Experiments (4th International Conference VSTTE)*, volume 7152 of *Lecture Notes in Computer Science*, pages 2–17, Philadelphia, USA, Jan. 2012. Springer.

[11] C. Marché and A. Tafat. Weakest precondition calculus, revisited using Why3. Research report, INRIA, Dec. 2012.

[12] The PVS system. `http://pvs.csl.sri.com/`.

[13] A. Riazanov and A. Voronkov. Vampire. In H. Ganzinger, editor, *16th International Conference on Automated Deduction*, volume 1632 of *Lecture Notes in Artificial Intelligence*, pages 292–296, Trento, Italy, July 1999. Springer.

[14] S. Schulz. System description : E 0.81. In D. A. Basin and M. Rusinowitch, editors, *Second International Joint Conference on Automated Reasoning*, volume 3097 of *Lecture Notes in Computer Science*, pages 223–228. Springer, 2004.

[15] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. Spass version 3.5. In R. A. Schmidt, editor, *22nd International Conference on Automated Deduction*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer, 2009.