Calcul des Constructions Inductives
D.E.A. Programmation: Sémantique, Preuves et Langages
2003-2004

Calcul des Constructions Inductives

Christine Paulin-Mohring, Benjamin Werner,

Bruno Barras, Hugo Herbelin,

Jean-Christophe Filliâtre, Claude Marché

Table des matières

Chapitre 7

Preuve de programmes impératifs

7.1 Logique de Hoare classique

On considère un langage PASCAL très simplifié, avec des variables globales entières, des expressions entières et booléennes, et les instructions d'affectation, de test et de boucle while:

$$\begin{array}{rcl} e & ::= & n \mid x \mid e\ op\ e \\ op & ::= & + \mid - \mid * \mid / \mid = \mid \neq \mid < \mid > \mid \leq \mid \geq \mid and \mid or \\ i & ::= & skip \mid x := e \mid i;i \mid if\ e\ then\ i\ else\ i \mid while\ e\ do\ i\ done \end{array}$$

Exemple 1 *Appelons ISQRT le programme suivant, sur trois variables n, count et sum :*

```
count := 0; sum := 1;
while sum <= n do count := count + 1; sum := sum + 2 * count + 1 done
```

Affirmation: à la fin de l'exécution de ce programme, count est la racine carré de n, arrondie à l'entier inférieur.

7.1.1 Sémantique opérationnelle

Un *état* d'un programme est une table d'association E qui à chaque variable x du programme associe sa valeur courante $E(x)$. La valeur d'une expression bien typée e dans un état E est définie par

$$\begin{array}{rcl} E(n) & = & n \\ E(x) & = & E(x) \\ E(e_1\ op\ e_2) & = & E(e_1)\ op\ E(e_2) \end{array}$$

La sémantique opérationnelle de ce langage est définie par les règles de transition (sur toute instruction bien typée) :

$$\begin{array}{lcl} E & \underset{x:=e}{\longrightarrow} & E\{x = E(e)\} \\ E_1 & \underset{i_1;i_2}{\longrightarrow} & E_3 \text{ si } E_1 \underset{i_1}{\longrightarrow} E_2 \text{ et } E_2 \underset{i_2}{\longrightarrow} E_3 \\ E_1 & \underset{if\ e\ then\ i_1\ else\ i_2}{\longrightarrow} & E_2 \text{ si } E_1(e) = true \text{ et } E_1 \underset{i_1}{\longrightarrow} E_2 \\ E_1 & \underset{if\ e\ then\ i_1\ else\ i_2}{\longrightarrow} & E_2 \text{ si } E_1(e) = false \text{ et } E_1 \underset{i_2}{\longrightarrow} E_2 \\ E_1 & \underset{while\ e\ do\ i}{\longrightarrow} & E_3 \text{ si } E_1(e) = true,\ E_1 \underset{i}{\longrightarrow} E_2 \text{ et } E_2 \underset{while\ e\ do\ i}{\longrightarrow} E_3 \\ E & \underset{while\ e\ do\ i}{\longrightarrow} & E \text{ si } E(e) = false \end{array}$$

7.1.2 Logique de Hoare

Un triplet de Hoare est un triplet noté $\{P\}i\{Q\}$ où P et Q sont des assertions logiques et i une instruction. Ces assertions logiques sont des formules du premier ordre, avec comme formules atomiques les expressions de notre langage. Remarque importante: il y a ainsi identification entre les variables du programme et les variables de la logique.

On dit qu'un triplet de Hoare $\{P\}i\{Q\}$ est valide si pour tous états E_1 et E_2 tels que $E_1 \underset{i}{\longrightarrow} E_2$ et P est vraie dans E_1, Q est vraie dans E_2.

Exemples de triplets valides: $\{x = 1\}x := x + 2\{x = 3\}$, $\{x = y\}x := x + y\{x = 2y\}$.

Exemple 2 *Sur le programme $ISQRT$, on souhaiterait montrer la validité du triplet*

$$\{n \geq 0\}ISQRT\{count * count \leq n \wedge n < (count + 1) * (count + 1)\}$$

Logique de Hoare: ensemble de règles de déduction sur les triplets:

$$\frac{}{\{P\}skip\{P\}} \qquad \frac{\{P \wedge e = true\}i_1\{Q\} \qquad \{P \wedge e = false\}i_2\{Q\}}{\{P\}if\ e\ then\ i_1\ else\ i_2\{Q\}}$$

$$\frac{}{\{P[x \leftarrow e]\}x := e\{P\}} \qquad \frac{\{I \wedge e = true\}i\{I\}}{\{I\}while\ e\ do\ i\{I \wedge e = false\}}$$

$$\frac{\{P\}i_1\{Q\} \qquad \{Q\}i_2\{R\}}{\{P\}i_1; i_2\{R\}} \qquad \frac{\{P'\}i\{Q'\} \qquad P \to P' \qquad Q' \to Q}{\{P\}i\{Q\}}$$

Proposition 1 *Cet ensemble de règles est correct: tout triplet dérivable est valide.*

Preuve: pas de difficulté (cf chapitre **??**).

Difficulté: prouver un triplet à partir de ces règles demande de « deviner » les bonnes annotations intermédiaires, par exemple pour la séquence, mais aussi pour la règle d'affaiblissement. Ainsi, on ne peut pas prouver le programme de la racine carrée sans réfléchir: il faut en particulier trouver un invariant de boucle adéquat. L'équivalent, du point de vue théorique, de cette difficulté est: a-t-on complétude de la logique de Hoare, c.-à-d. peut-on prouver tous les triplets valides ?

7.1.3 Complétude, et calcul de plus faible précondition

Pour i et Q fixés, l'ensemble des P tels que $\{P\}i\{Q\}$ est valide, s'il est non vide, possède-t-il un élément *minimal* P_0 au sens où pour tout P tels que $\{P\}i\{Q\}$ est valide, P implique P_0.

Calcul de WP:

$$\begin{array}{rcl} WP(x := e, Q) & = & Q\{x \leftarrow e\} \\ WP(i_1; i_2, Q) & = & WP(i_1, WP(i_2, Q)) \\ WP(if\ e\ then\ i_1\ else\ i_2, Q) & = & (e = true \to WP(i_1, Q)) \wedge (e = false \to WP(i_2, Q)) \\ WP(while\ e\ do\ i, Q) & = & \text{pas de formule simple !} \end{array}$$

Exemple 3 $WP(x := x + y, x = 2y) \equiv x + y = 2y$

Proposition 2 *L'ensemble des règles de logique de Hoare est* relativement *complet: Tout triplet valide $\{P\}i\{Q\}$ est dérivable (en particulier, on peut trouver des invariants pour les boucles while de i).*

Preuve: le *relativement* exprime ici une hypothèse qui est que la logique dans laquelle on exprime les annotations est suffisamment expressive, en particulier pour exprimer les invariants de boucle nécessaires à l'aide de point-fixe [Cou90].

```
let isqrt = fun (n : int) ->
  { n >= 0 }     (* pré-condition *)
  begin
    let count = ref 0 in
    let sum = ref 1 in
    begin
    while !sum <= n do
      { invariant count >= 0 and n >= count*count and sum = (count+1)*(count+1)
        variant   n - sum }  (* invariant et variant de boucle *)
      count := !count + 1;
      sum := !sum + 2 * !count + 1
    done;
   !count
    end
  end
  { result >= 0 and result * result <= n and n < (result+1)*(result+1) }
    (* post-condition *)
```

FIG. 7.1 – Calcul de la racine carrée en Why

7.1.4 Difficultés

De nombreux travaux ont fait suite à la logique de Hoare originale [Cou90], pour étendre le formalisme et résoudre des difficultés, par exemple:

- Référer, dans une post-condition Q de $\{P\}i\{Q\}$, à la valeur d'une variable avant l'exécution de i.
- Avoir des effets de bord dans les expressions.
- Traiter l'appel de sous-programmes.
- Prouver la terminaison des programmes (terminaison des boucles while).
- Supporter les `break`, `continue`, les exceptions.
- Avoir des structures de données complexes: tableaux, structures, pointeurs, objets, etc.
- Travailler sur des entiers bornés, et vérifier le non-débordement des opérations sur les entiers.

7.2 Transformation fonctionnelle: la méthode Why

Objectif général: établir la validité d'un triplet de Hoare, non pas avec les règles de déduction précédente, mais avec une technique fondée sur le calcul des constructions inductives. Cet objectif a été développé par Jean-Christophe Filliâtre [Fil03a] et implantée dans un outil logiciel appelée Why [Fil03b]. Les programmes traités par Why ne sont pas en PASCAL, ni en C ou autre langage de programmation existant, mais dans une syntaxe spécifique, qui a été conçue pour la preuve de ces programmes.

En Why, toutes les difficultés mentionnées ci-dessus sont traitées, exceptés la possibilité d'avoir des structures de données complexes et les entiers bornés. Dans la section suivante, nous verrons à la fois comment traiter les structures de données complexes, et comment faire des preuves sur des programmes écrits dans des langages de programmation impératifs standards: Java ou C en l'occurrence.

La figure 7.1 donne de nouveau le programme qui calcule la racine carrée, écrit cette fois sous forme d'une fonction Why. On remarque que:

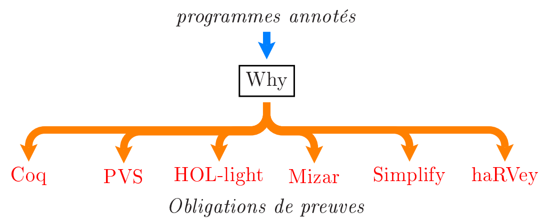

FIG. 7.2 – Approche multi-prouveur de Why

- Le langage Why est un langage proche de Caml
- En particulier, il n'y a pas de distinction entre instruction et expression. Les triplets à la Hoare s'applique sur des expressions, et dans une post-condition on peut utiliser le mot clé **result** pour parler du résultat de l'expression.
- L'ajout d'un variant, pour garantir la terminaison

Le travail de l'outil Why consiste à produire, à partir d'un programme annoté, des *obligations de preuves* dont la validité assure la correction du programme. Avec Why, ces obligations de preuves sont des formules en logique du premier ordre, exprimable dans la syntaxe de différents démonstrateurs existants, aussi bien des démonstrateurs interactifs comme Coq, PVS, HOL-light ou Mizar, que des démonstrateurs automatiques comme Simplify ou haRVey.

Néanmoins, la sortie pour Coq possède une particularité supplémentaire: la *validation*, qui est un programme fonctionnel Coq équivalent au programme de départ. Il s'agit là de l'idée directrice de l'approche Why : un programme impératif peut être traduit en un programme fonctionnel par ajout de paramètre. Sur l'exemple de ISQRT, on écrirait quelque chose comme

```
isqrt(n) = isqrt2(n,0,1)
isqrt2(n,count,sum) =
  if sum<=n then isqrt2(n,count+1,sum + 2 * (count+1)+1) else count
```

On est ramené alors à la preuve d'un programme fonctionnel. Il s'agit là d'une instance de l'approche par plongement superficiel (*shallow embedding* en anglais) opposé à plongement profond (*deep embedding*), tel qu'expliqué dans le chapitre **??**.

7.2.1 Le langage Why

C'est un langage fonctionnel auquel sont rajoutés quelques traits impératifs. Les types de base sont: `int`, `bool`, `float` et `unit` (habité par la constante `void`). Comme d'habitude avec les langages fonctionnels, il n'y a pas de distinction syntaxiques entre expressions et instructions: les instructions sont les expressions de type `unit`.

Le noyau fonctionnel est constitué des expressions suivantes :

- les constantes entières, booléennes, flottantes et `void` ;
- les variables ;
- l'application, notée sous forme curryfiée $(e\ e_1\ \dots\ e_n)$;
- les opérations primitives (unaires et binaires) $+$, -, etc. ;
- les définitions locales `let` $v = e_1$ `in` e_2 ;
- les conditionnelles `if` e_1 `then` e_2 `else` e_3 ;
- l'abstraction notée `fun` $(x_1 : t_1)\ \cdots\ (x_n : t_n) \to e$.

Les traits impératifs sont introduits par

- les définitions de références locales `let` $v =$ `ref` e_1 `in` e_2 ;

- la déréférenciation $!v$;
- l'affectation: $v := e$;
- la séquence $e_1; ...; e_n$;
- la boucle `while` e_1 `do` e_2 `done` ;

Remarque: pour simplifier on ne traite pas les définitions récursives ici, ni la levée et le rattrapage d'exception. Le langage complet est décrit dans le manuel utilisateur.

En Why, toute expression e peut être annotée, avec la notation des triplets de Hoare $\{pre\}\ e\ \{post\}$. Les boucles *while* doivent être annotée par

while e do { invariant inv variant var } e done

pour spécifier un invariant de boucle, ainsi qu'un *variant*, une expression censée décroitre à chaque itération de la boucle, assurant ainsi la terminaison. Les annotations *pre*, *post* et *inv* sont des formules de logique du premier ordre, leur variables sont les variables du programme. De plus, dans les *post* on peut utiliser le mot-clé result pour référer à la valeur résultat de l'expression, et v@ pour désigner la valeur de v avant l'exécution de l'expression.

7.2.2 Typage avec effets

Comme un langage de programmation classique, le langage Why possède des règles de typage, et un programme doit être correctement typé pour que la génération des obligations de preuves puissent se faire. Comme ce langage est proche de Caml, le type en est assez proche aussi, mais on lui ajoute une spécificité qui est le typage des effets, qui consiste à préciser quelles sont les références qui sont lues et celles qui sont écrites.

Voici un exemple de programme très simple, qui va nous servir pour illustrer les mécanismes de Why:

```
parameter montant : int ref

let crediter = fun (s:int) ->
  { s >= 0 }
  montant := !montant + s
  { montant = s + montant@ }
```

En tant que programme « Caml », on sait très bien inférer que `crediter` a le type `int -> unit`. Ce qu'on va faire en plus c'est de calculer ses effets, pour déterminer dans cet exemple que la référence `montant` est à la fois lue et écrite.

Un *type avec effet* est un triplet (type, variables lues, variables écrites). Voici un extrait des règles de typage de Why (cf [Fil03a] pour le reste).

$$\frac{x : t \in \Gamma \qquad t \text{ non ref}}{\gamma \vdash x : (t, \emptyset, \emptyset)} \qquad \frac{x : t\ ref \in \Gamma}{\Gamma \vdash !x : (t, \{x\}, \emptyset)}$$

$$\frac{\Gamma \vdash e_1 : (t_2 \rightarrow (t_1, R, W), R_1, W_1) \qquad \Gamma \vdash e_2 : (t_2, R_2, W_2) \qquad t_2 \text{ non ref}}{\Gamma \vdash (e_1\ e_2) : (t_1, R_1 \cup R_2 \cup R, W_1 \cup W_2 \cup W)}$$

$$\frac{\Gamma, x : t \vdash e : (t', R, W)}{\Gamma \vdash fun(x : t) \rightarrow e : (t \rightarrow (t', R, W), \emptyset, \emptyset)}$$

$$\frac{x : t\ ref \in \Gamma \qquad \Gamma \vdash e : (t, R, W)}{\Gamma \vdash x := e : (unit, \{x\} \cup R, \{x\} \cup W)}$$

La règle de typage de l'application ci-dessus interdit d'appliquer une fonction à une référence. Une règle spéciale existe dans ce cas:

$$\frac{\Gamma \vdash e_1 : (t_2\ ref \rightarrow (t_1, R, W), R_1, W_1) \qquad \Gamma \vdash r : (t_2\ ref, R_2, W_2) \qquad r \notin (t_1, R, W)}{\Gamma \vdash (e_1\ r) : (t_1[x \leftarrow r], R_1 \cup R_2 \cup R[x \leftarrow r], W_1 \cup W_2 \cup W[x \leftarrow r])}$$

Il est fondamental de remarquer que cette règle interdit les *alias*, c.-à-d. de référencer la même chose avec deux noms différents. C'est une condition essentielle pour garantir la correction de la méthode Why. Voici un exemple typique:

```
let incr2 = fun (x:int ref) (y:int ref) ->
  { true } begin x := !x + 1; y := !y + 1 end { x = x@ + 1 and y = y@ + 1 }

parameter t : int ref

let test = { true } (incr2 t t) { t = t@ + 1 }
```

Why signale une erreur de typage sur l'application `(incr2 t t)`. Si c'était accepté, alors au vu de la post-condition de `incr2`, la post-condition `t = t@ + 1` serait prouvable, or elle est fausse, c'est `t=t@ + 2` qui est vrai.

Exercice 1 *Dériver le jugement de typage*

$$montant : int\ ref \vdash crediter : (int \rightarrow (unit, \{montant\}, \{montant\}), \emptyset, \emptyset))$$

On peut utiliser cette approche de typage avec effets pour définir des programmes Why de façon modulaire: un programme peut très faire appel à une autre fonction dont le code n'est pas donné, cette fonction doit seulement être spécifiée par son type avec effets, ainsi qu'avec un pré et une post-condition. Par exemple on peut écrire :

```
parameter montant : int ref

parameter crediter : s:int ->
  { s >= 0 }
  unit reads montant writes montant
  { montant = s + montant@ }

let test = fun (tt:unit) ->
  { true }
  begin
    (crediter 50); (crediter 80)
  end
  { montant = montant@ + 130 }
```

7.2.3 Calcul de plus faible précondition

Avec l'inférence de type avec effets, on est capable d'associer à chaque sous-expression d'un programme un type avec effets. L'étape suivante consiste à leur associer aussi une pré et une post-condition: de deux choses l'une: soit l'expression considérée est déjà annotée (une boucle while, un identificateur de fonction, ou une expression explicitement annotée par l'utilisateur), soit on lui détermine une annotation par calcul de plus faible pré-condition, d'une manière très similaire à celui de la logique de Hoare classique, mais adaptée au langage Why.

Voici un extrait des formules de calcul de WP, les règles complètes sont dans le manuel de Why [Fil02] et dans [Fil99] .

$$\begin{array}{rcl}
WP(x,Q) & = & Q[result \leftarrow x] \\
WP(!x,Q) & = & Q[result \leftarrow x] \\
WP(x := e,Q) & = & WP(e, Q[result \leftarrow tt, x \leftarrow result, x@ \leftarrow x]) \\
WP(e_1;e_2,Q) & = & WP(e_1, WP(e_2,Q)) \\
WP(if\ e_1\ then\ e_2\ else\ e_3, Q) & = & WP(e_1, if\ result\ then\ WP(e_2,Q)\ else\ WP(e_3,Q) \\
WP(let\ x = e_1\ in\ e_2, Q) & = & WP(e_1, WP(e_2,Q)[x \leftarrow result])
\end{array}$$

La règle pour l'application est l'une des plus complexe:

$$\begin{array}{rl}
WP((f\ e_1\ \cdots\ e_n),Q) = & WP(e_1, WP(e_2, \ldots, WP(e_n, \\
& (P_f \wedge \forall w_1, \ldots, w_k, result, Q_f \to Q)[x_n \leftarrow result]) \ldots)[x_1 \leftarrow result])
\end{array}$$

si f a le type avec effets annoté $x_1 : t_1 \cdots x_n : t_n \to \{P_f\}(t,R,W)\{Q_f\}$, et les expressions $e_1, \ldots, e_n$ sont pures, c.-à-d. ne modifient aucune variable. Cette dernière condition fait qu'il peut arriver que l'outil Why peut parfois ne pas parvenir à générer les obligations de preuves, auquel cas il faut simplifier l'application concernée en introduisant des `let...in...`

Exercice 2 *Annoter le code de* `crediter`.

7.2.4 Traduction fonctionnelle

Il s'agit maintenant du cœur même de la méthode Why. Nous en donnons ici un aperçu, les détails étant dans[Fil99, Fil03a].

Le but est d'engendrer un programme fonctionnel Coq équivalent au programme Why complètement annoté. Ce programme comportera des « trous » pour les preuves des annotations logiques : les obligations de preuve. Celle-ci peuvent naturellement être prouvées dans Coq, pour obtenir alors un programme fonctionnel Coq certifié, équivalent au programme de départ, que l'on appelle la *validation* Why.

Interprétation fonctionnelle des types avec effets

Tout type avec effets annoté $T = \{P\}(t,\vec{r},\vec{w})\{Q\}$ est interprété en un type Coq $\overline{T}$:

$$\forall \vec{x}, P(\vec{x}) \to \exists \vec{y}, \exists r, Q(\vec{x},\vec{y},r)$$

La notation $P(\vec{x})$ désigne la formule P où les occurrences des variables lues $\vec{r}$ sont substituées par les $\vec{x}$, et la notation $Q(\vec{x},\vec{y},r)$ désigne la formule Q où, pour chaque variable v de $\vec{w}$, les occurrences de v sont remplacées par le y correspondant, les $v@$ par le x correspondant, pour chaque variable v de $\vec{r}$ qui n'est pas dans $\vec{w}$, chaque occurrence de v est remplacée par le x correspondant, et enfin `result` est remplacé par r. Il s'agit là d'exprimer le fait que les x désignent les anciennes valeurs des variables modifiables, alors que les y désignent les nouvelles valeurs.

En pratique, le $\exists$ est utilisé est celui dans Set, noté avec des accolades. Par exemple, le type avec effet de `crediter` :

$$s : int \to \{s >= 0\}(unit, \{montant\}, \{montant\})\{montant = s + montant@\}), \emptyset, \emptyset)$$

est interprété par le type Coq

```
forall (s: Z), forall (montant: Z), forall (H: s >= 0),
  { montant0: Z, result: unit | montant0 = s + montant }
```

Notons également que les types de base `int`, `bool` et `float` sont interprétés par leurs représentations mathématiques Coq (`Z`, `bool` et `R`) et donc en particulier les entiers ne sont pas bornés.

Interprétation fonctionnelle des programmes

On donne maintenant des règles de traduction d'une expression e de type avec effets $T = \{P\}(t, \{r_1, \ldots, r_k\}, \{w_1, \ldots, w_l\})\{Q\}$ dans un environnement Γ, en un terme Coq à trous $\overline{e}$ de type $\overline{T}$ dans l'environnement $\overline{\Gamma}$ obtenu en enlevant toutes les variables de type ref.

- si $e \equiv x$ variable:
  $$\overline{e} = \lambda\vec{x_0}, \lambda p : P(\vec{x_0}), (\emptyset, x, ?_1)$$
  où $?_1 : Q(\vec{x_0}, \emptyset, x)$
- si $e \equiv !x$:
  $$\overline{e} = \lambda\vec{x_0}, \lambda p : P(\vec{x_0}).(\emptyset, x_{0,i}, ?_1)$$
  où $x_{0,i}$ est la variable correspondant à x, et $?_1 : Q(\vec{x_0}, \emptyset, x_{0,i})$
- si $e \equiv x := e_1$, où e_1 a le type $\{P_1\}(t_1, \vec{r_1}, \vec{w_1})\{Q_1\}$
  $$\overline{e} = \lambda\vec{x_0}, \lambda p : P(\vec{x_0}), let(\vec{x_1}, v, q_1) = (\overline{e_1}\ \vec{x_0}\ ?_1)\ in\ (\vec{x_1} \oplus \{x \leftarrow v\}, tt, ?_2)$$
  où $?_1 : P_1(\vec{x_0})$ et $?_2 : Q(\vec{x_0}, \vec{x_1} \oplus \{x \leftarrow v\}, tt)$, où $\vec{x_1} \oplus \{x \leftarrow v\}$ désigne le vecteur de variables $\vec{x_1}$ où la variable $x_{1,i}$ correspondant à x est remplacée par v.
- si $e \equiv fun(x : t_1) \to e_1$ et t_1 n'est pas un type ref:
  $$\overline{e} = \lambda\vec{x_0}, \lambda p : P(\vec{x_0}), (\emptyset, \lambda x.\overline{e_1}, ?_1)$$
  où $?_1 : Q(\vec{x_0}, \emptyset, \lambda x.\overline{e_1})$.
- si $e \equiv (e_2\ e_1)$: alors e_2 a un type avec effets annoté de la forme
  $$\{P_2\}(x : t_1 \to \{P'\}(t, R', W')\{Q'\}, R_2, W_2)\{Q_2\}$$
  on distingue deux cas suivant e_1:
  - t_1 non ref, e_1 a le type $\{P_1\}(t_1, R_1, W_1)\{Q_1\}$:
    $$\begin{array}{rl}\overline{e} = \lambda\vec{x_0}, \lambda p : P(\vec{x}), & let\ (\vec{x_1}, a, q_1) = (\overline{e_1}\ \vec{x_0}\ ?_1)\ in \\ & let\ (\vec{x_2}, f, q_2) = (\overline{e_2}\ (\vec{x_0} \oplus \vec{x_1})\ ?_2)\ in \\ & let\ (\vec{x_3}, v, q') = (f\ a\ (\vec{x_0} \oplus \vec{x_1} \oplus \vec{x_2})\ ?_3)\ in \\ & (\vec{x_1} \oplus \vec{x_2} \oplus \vec{x_3}, v, ?_4)\end{array}$$
    où $?_1 : P_1(\vec{x_0})$, $?_2 : P_2(\vec{x_0} \oplus \vec{x_1})$, $?_3 : P'(\vec{x_0} \oplus \vec{x_1} \oplus \vec{x_2})$ et $?_4 : Q(\vec{x_0}, \vec{x_1} \oplus \vec{x_2} \oplus \vec{x_3}, v)$, et la notation $\vec{x} \oplus \vec{y}$ désigne le vecteur de variables obtenus en ajoutant/surchargeant les variables de $\vec{y}$ aux variables de $\vec{x}$.
  - $t_1 = t'_1\ ref$, alors e_1 est une variable r et
    $$\begin{array}{rl}\overline{e} = \lambda\vec{x_0}, \lambda p : P(\vec{x_0}), & let\ (\vec{x_1}, f, q_2) = (\overline{e_2}\ \vec{x_0}\ ?_1)\ in \\ & let\ (\vec{x_2}, v, q') = (f\ r\ (\vec{x_0} \oplus \vec{x_1})\ ?_2\ in \\ & (\vec{x_1} \oplus \vec{x_2}, v, ?_3)\end{array}$$
    où $?_1 : P_2(\vec{x_0}$, $?_2 : P'(\vec{x_0} \oplus \vec{x_1})$ et $?_3 : Q(\vec{x_0}, \vec{x_1} \oplus \vec{x_2}, v)$.

Remarques: les interprétations de l'application évaluent les arguments de l'application de droite à gauche. L'interprétation des boucles while [Fil99, Fil03a] utilise `well_founded_induction` tel qu'expliqué au chapitre **??**.

Exemple 4 *La validation à trous de* `crediter` *est*

```
Definition crediter :=
  (fun (s: Z) (montant: Z) (Pre: s >= 0) =>
    let (result1, Post) :=
      exist (fun (result: Z) => result = (s + montant))
            (montant + s) (PO1 s montant Pre)
      (* : { result: Z | result = s + montant } *)
```

```
  in
    exist_2 (fun (montant1: Z) (result0: unit) => montant1 = s + montant)
            result1 tt Post)
    (* : { result1: Z, result2:unit | result1 = s + montant } *)
```

où

```
PO1 (s:Z) (montant:Z) (Pre:s>=0) : montant + s = s + montant
```

Proposition 3 (Correction [Fil99, Fil03a]) *Pour toute expression e typée avec effets annotée, si les obligations de preuve de $\overline{e}$ sont prouvables, alors e vérifie sa spécification (c.à-d. que sa post-condition est valide quand sa pré-condition est satisfaite).*

Pour la preuve de cette proposition, il faut définir formellement la sémantique de Why [Fil99, Fil03a].

7.3 Traitement des structures données complexes et application à d'autres langages de programmation

Why ne gère pas explicitement de structures de données complexes. Néanmoins, il a une approche modulaire, dans le sens où de même qu'il accepte des programmes en paramètre avec uniquement leurs spécifications, il accepte des types abstraits, et des prédicats et fonctions logiques abstraites sur ces types.

7.3.1 Exemple d'un programme avec un tableau: le drapeau hollandais

Il s'agit d'un exemple célèbre: un programme de tri linéaire quand il n'y a que trois valeurs (comme les couleurs du drapeau hollandais !) dû à Dijkstra [Dij76].

On introduit un type abstrait Why avec des axiomes :

```
parameter BLUE, WHITE, RED : color
logic iscolor : color -> prop
axiom color_elim : forall c:color. iscolor(c) -> c=BLUE or c=WHITE or c=RED
```

puis un premier programme Why uniquement spécifié, pour le test d'égalité des couleurs:

```
parameter eq_color : c1:color -> c2:color ->
  { } bool { if result then c1=c2 else c1<>c2 }
```

On introduit ensuite une axiomatisation des tableaux: on les représente en logique comme des tableaux « fonctionnels », c.-à-c. où la mise à jour d'une case de tableau retourne un nouveau tableau:

```
logic acc : colorarray, int -> color  (* acc(t,i) represente t[i] *)
logic length : colorarray -> int      (* longueur d'un tableau    *)
logic update : colorarray, int, color -> colorarray (* mise à jour t[i] := c *)
axiom length_pos: forall t:colorarray. 0 <= length(t)

axiom length_up:
  forall t:colorarray.
    forall i:int. forall v:color.
       length(update(t,i,v)) = length (t)
```



```
axiom acc_up_eq :
  forall t:colorarray.
    forall i:int. forall v:color.
       acc(update(t,i,v),i) = v

axiom acc_up_neq :
  forall t:colorarray.
    forall i:int. forall j:int. forall v:color.
       i <> j -> acc(update(t,i,v),j) = acc(t,j)
```

et on a les programme WHY suivant sur les tableaux:

```
parameter length_ : t:colorarray -> { } int { result = length(t) }

parameter acc_ : t:colorarray -> i:int ->
  { 0 <= i < length(t) }
  color
  { result=acc(t,i) }

parameter update_ : t:colorarray ref -> i:int -> v:color ->
  { 0 <= i < length(t) }
  unit reads t writes t
  { t = update(t@,i,v) }
```

noter les préconditions sur les bornes des tableaux utilisés.

On peut maintenant commencer par un petit programme qui echange deux éléments d'un tableau:

```
let swap = fun (t : colorarray ref) (i:int) (j:int) ->
  { 0 <= i < length(t) and 0 <= j < length(t) }
  let ti = (acc_ !t i) in
  let tj = (acc_ !t j) in
  begin
  (update_ t i tj);
  (update_ t j ti)
  end
  { length(t) = length(t@) and
    acc(t,i) = acc(t@,j) and
    acc(t,j) = acc(t@,i) and
    forall k:int. i <> k and j <> k -> acc(t,k) = acc(t@,k) }
```

Pour écrire le programme de tri de Dijkstra, on introduit un predicat `monochrome` :

```
logic monochrome : colorarray, int, int, color -> prop

axiom mon1 :
  forall t:colorarray.
    forall i:int. forall j:int. forall c:color.
      monochrome(t,i,j,c) -> (forall k:int. i <= k < j -> acc(t,k)=c)
axiom mon2 :
  forall t:colorarray.
    forall i:int. forall j:int. forall c:color.
      (forall k:int. i <= k < j -> acc(t,k)=c) -> monochrome(t,i,j,c)
```

et le programme principal est alors

```
let flag = fun (t : colorarray ref) ->
  { forall k:int. 0 <= k < length(t) -> iscolor(acc(t,k)) }
  begin
  let b = ref 0 in
  let i = ref 0 in
  let r = ref (length_ !t) in
  while !i < !r do
   { invariant
       (forall k:int. 0 <= k < length(t) -> iscolor(acc(t,k)))
       and 0 <= b and b <= i and i <= r and r <= length(t)
       and monochrome(t, 0, b, BLUE)
       and monochrome(t, b, i, WHITE)
       and monochrome(t, r, length(t), RED)
    variant r - i }
    let c = (acc_ !t !i) in
    if (eq_color c BLUE)
    then
      begin (swap t !b !i); b := !b + 1; i := !i + 1 end
    else
     if (eq_color c WHITE)
     then i := !i + 1
     else
       begin r := !r - 1; (swap t !r !i) end
   done
  end
  { exists r:int. exists b:int.
         monochrome(t, 0, b, BLUE)
       and monochrome(t, b, r, WHITE)
       and monochrome(t, r, length(t), RED) }
```

Notons que les onze obligations de preuves engendrées pour ces deux programmes (`swap` et `flag`) sont prouvées entièrement automatiquement par Simplify.

7.3.2 Programmes Java et C

L'approche ci-dessus pour les tableaux simples peut être étendue à toute structure de données complexes, ce qui permet de traiter des « vrais » langages comme Java ou C: pour ceux-ci, la structure de la mémoire est modélisée avec les opérations logiques abstraites et des axiomes. L'outil Krakatoa [MPMU04] suit ce principe, et permet de traduire automatiquement un programme Java en un programme Why avec une telle modélisation. L'outil Caduceus, en cours de développement, fait de même avec les programmes C.

Remarquons aussi que cette approche pour modéliser les structures de données complexes peut être aussi utilisée pour traiter les entiers bornés: il suffit d'interpréter les opérations arithmétiques comme l'addition par une autre fonction logique. On a même le choix de permettre les débordements et faire des calculs modulo 2^{32}, ou bien imposer le non-débordement, en introduisant une fonction d'addition sur les entiers non bornés mais avec précondition de la forme

```
parameter bounded_add : x:int -> y:int ->
  { -2^31 <= x+y < 2^31 } int { result = x+y }
```

Bibliographie

[Cou90] Patrick Cousot. Methods and logics for proving programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 841–993. North-Holland, 1990.

[Dij76] Edsger W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.

[Fil99] J.-C. Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999.

[Fil02] Jean-Christophe Filliâtre. The why verification tool, 2002. `http://why.lri.fr/`.

[Fil03a] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[Fil03b] J.-C. Filliâtre. Why: a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003. `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz`.

[MPMU04] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.