École Jeunes Chercheurs en Programmation (EJCP)

Vérification déductive de programmes avec Why3

Jean-Christophe Filliâtre

Juin 2012

Informations pratiques

Ce cours a été écrit en utilisant la version 0.72 de Why3 (mai 2012). Le matériel relatif à ce cours (transparents, exercices, code source décrit dans ce document, instructions pour installer Why3, etc.) est disponible à cette adresse :

`http://why3.lri.fr/ejcp-2012/`

En complément de ce cours, on pourra consulter la documentation de Why3, ainsi que les nombreux exemples de preuve de programmes disponibles sur `http://why3.lri.fr`.

Coordonnées de l'auteur :

Jean-Christophe Filliâtre

LRI - bâtiment 650
Université Paris-Sud
91405 Orsay Cedex
France

Email : `filliatr@lri.fr`
Web : `http://www.lri.fr/~filliatr/`

Table des matières

1 Introduction **5**

2 Logique **9**
2.1 Premier contact avec Why3 9
2.2 Parler à l'oreille des démonstrateurs 15
2.3 Syntaxe et sémantique 16
2.3.1 Syntaxe 16
2.3.2 Sémantique 16

3 Preuve de programmes **19**
3.1 Introduction 19
3.2 Structures de données 25
3.2.1 Tableaux 25
3.2.2 Types algébriques 28
3.2.3 Modélisation 32
3.3 Syntaxe et sémantique 36
3.3.1 Syntaxe 36
3.3.2 Typage 38
3.3.3 Plus faibles préconditions 39

Chapitre 1

Introduction

On fête cette année le 42e anniversaire de la fonction 91 de John McCarthy [8]. Cette fonction récursive, s'il est nécessaire de le rappeler, est ainsi définie :

$$f(n) = \begin{cases} n - 10 & \text{si } n > 100, \\ f(f(n+11)) & \text{sinon.} \end{cases}$$

On peut se poser alors la question de savoir si cette fonction renvoie toujours la valeur 91, comme son nom semble le suggérer, ou, à défaut, sous quelles conditions. On peut également se poser la question de sa terminaison, qui n'a rien d'évident, ou encore de son équivalence avec le code impératif suivant :

```
e ← 1
while e > 0 do
  if n > 100 then
    n ← n − 10
    e ← e − 1
  else
    n ← n + 11
    e ← e + 1
return n
```

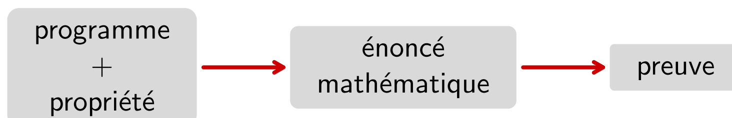
Il s'agit là de questions ayant trait à la *preuve de programmes*. Si les deux programmes ci-dessus sont donnés sous forme de pseudo-code, on peut bien évidemment se poser des questions similaires pour des programmes écrits dans des langages de programmation usuels. Ainsi, on peut chercher à montrer que le programme Java suivant trie effectivement un tableau de booléens

```
int i = 0, j = a.length - 1;
while (i < j)
  if (!a[i]) i++;
  else if (a[j]) j-;
  else swap(a, i++, j-);
```

ou encore que le programme C suivant, bien que volontairement obscurci, calcule effectivement le nombre de solutions du problème des N reines :

```
t(a,b,c){int d=0,e=a&~b&~c,f=1;if(a)for(f=0;d=(e-=d)&-e;f+=t(a-d,(b+d)*2,(
c+d)/2));return f;}main(q){scanf("%d",&q);printf("%d\n",t(~(~0«q),0,0));}
```

Ce cours est une introduction au système Why3 [3], un ensemble d'outils qui permettent notamment la *vérification déductive* de programmes. Par vérification déductive, on entend l'idée d'exprimer le fait qu'un programme satisfait une propriété par un énoncé mathématique, dont on cherche ensuite à construire une preuve, ce qui peut être schématisé ainsi :

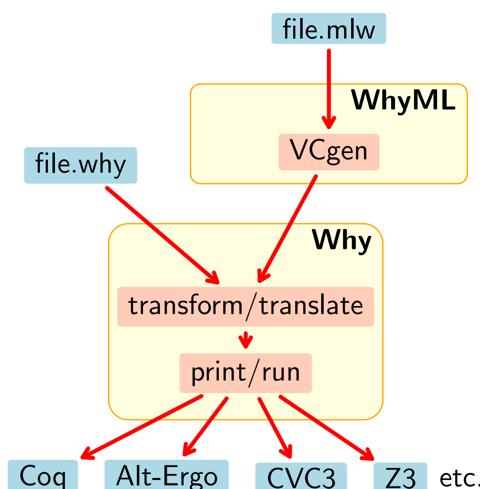


Cette idée est aussi vieille que l'informatique [15, 7], mais elle prend tout son sens aujourd'hui grâce au progrès technologique des *démonstrateurs automatiques* [1]. Ceux-ci permettent en effet de décharger l'utilisateur d'une grande partie de la preuve, généralement fastidieuse, voire même parfois de l'intégralité de la preuve.

La plate-forme Why est développée depuis une dizaine d'années dans l'équipe ProVal (LRI / INRIA), avec l'idée centrale d'offrir une interface commune vers de multiples démonstrateurs, non seulement automatiques mais aussi interactifs. Durant ces dernières années, la plate-forme a notamment été utilisée pour la preuve

— de programmes Java, avec l'outil Krakatoa [10] ;
— de programmes C, d'abord avec l'outil Caduceus [6] puis aujourd'hui avec le greffon Jessie de Frama-C [12] ;
— de programmes probabilistes [2] ;
— de programmes cryptographiques [1] ;
— d'algorithmes.

Depuis février 2010 est développée une nouvelle version de cette plate-forme, sous le nom Why3, par François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond et Andrei Paskevich. Très schématiquement, on peut présenter l'architecture de Why3 comme un système à deux étages, de la façon suivante :



La partie basse fournit un langage logique appelé Why et se charge de le traduire vers le langage natif de différents démonstrateurs (Coq, Alt-Ergo, CVC3, Z3, etc.). Cette partie est détaillée dans le chapitre 2. La partie haute fournit un langage de programmation,

1. et notamment ceux de la famille SMT, d'où la notion de *révolution SMT*

appelé WhyML, où les programmes sont spécifiés dans le langage Why. La correction d'un programme donné est une formule logique transmise à la partie basse de l'outil, pour être prouvée à l'aide des démonstrateurs. Cette partie est détaillée dans le chapitre 3.

Chapitre 2

Logique

Avant de parler de preuve de programmes avec Why3, on va commencer par introduire sa logique.

2.1 Premier contact avec Why3

Dans un fichier `demo_logic.why`, définissons le type algébrique des listes polymorphes. Pour cela, on introduit une théorie `List` contenant la déclaration du type `list 'a`[1].

```
theory List
  type list 'a = Nil | Cons 'a (list 'a)
end
```

On peut passer ce fichier sur la ligne de commande de Why3, de la manière suivante :

```
  why3 demo_logic.why
```

Le contenu du fichier `demo_logic.why` est alors vérifié, puis ré-imprimé sur la sortie standard. Ajoutons maintenant la définition d'un prédicat récursif `mem` indiquant la présence d'un élément `x` dans une liste `l`.

```
  predicate mem (x: 'a) (l: list 'a) = match l with
    | Nil -> false
    | Cons y r -> x = y \/ mem x r
  end
```

Le système vérifie automatiquement la terminaison du prédicat `mem`, en cherchant un sous-ensemble de ses arguments qui garantit une décroissance structurelle lexicographique. Ici, cet ensemble se limite à l'argument `l`. Si on avait malencontreusement écrit `mem x l` à la place de `mem x r`, alors la définition de `mem` aurait été rejetée par le système :

```
  why3 demo_logic.why
  File "demo_logic.why", line 10, characters 2-6:
  Cannot prove the termination of mem
```

Ajoutons enfin une troisième déclaration à la théorie `List`, à savoir un but visant à montrer que l'entier 2 appartient à la liste [1 ;2 ;3]. On l'écrit ainsi :

1. La variable de type α s'écrit `'a` dans la syntaxe concrète, comme en OCaml.

```
goal G1: mem 2 (Cons 1 (Cons 2 (Cons 3 Nil)))
```

Il est alors possible de chercher à prouver ce but avec l'un des démonstrateurs connus de Why3. Supposons par exemple que le démonstrateur SMT Alt-Ergo est installé et connu de Why3 (ce que l'on peut vérifier avec `why3 -list-provers`). Alors on peut passer le but `G1` à Alt-Ergo avec l'option `-P` de Why3.

```
why3 -P alt-ergo demo_logic.why
demo_logic.why List G1 : Valid (0.03s)
```

Comme on le voit ci-dessus, le but est validé par le démonstrateur, sans surprise. Une alternative à la ligne de commande `why3` consiste à utiliser l'interface graphique `why3ide`.

```
why3ide demo_logic.why
```

On peut alors lancer interactivement les différents démonstrateurs connus de Why3, y compris les démonstrateurs interactifs tels que Coq.

Définissons maintenant la notion de longueur d'une liste. On le fait dans une seconde théorie `Length`, dont les deux premières déclarations consistent à importer la théorie `List` précédente, ainsi que la théorie de l'arithmétique entière de la bibliothèque standard de Why3, à savoir `int.Int`.

```
theory Length
  use import List
  use import int.Int
```

Dans la théorie précédente, nous n'avions pas eu besoin d'importer l'arithmétique, car seules des constantes entières avaient été utilisées ; ici, nous aurons besoin également de l'addition et de la relation d'ordre sur les entiers. On peut alors définir une fonction récursive `length` sur les listes, de manière analogue à la définition du prédicat `mem`.

```
  function length (l: list 'a) : int = match l with
    | Nil -> 0
    | Cons _ r -> length r + 1
  end
```

Là encore, le système vérifie automatiquement la terminaison de cette définition récursive. On énonce alors un lemme affirmant que la longueur d'une liste est toujours positive ou nulle.

```
  lemma length_nonnegative: forall l:list 'a. length(l) >= 0
```

Cette fois, le démonstrateur automatique n'est pas capable de prouver ce résultat. En l'occurrence, il échoue rapidement avec le message `Unknown` qui signifie qu'il ne peut statuer quant à la validité du but.

```
why3 -P alt-ergo demo_logic.why
demo_logic.why List G1 : Valid (0.01s)
demo_logic.why Length length_nonnegative : Unknown: Unknown (0.01s)
```

En effet, la preuve du lemme `length_nonnegative` nécessite une récurrence, ce qui est hors de portée des démonstrateurs SMT. On peut se reporter sur un assistant de preuve tel que Coq pour faire cette preuve, par exemple en l'appelant depuis l'interface graphique `why3ide`. En revanche, Alt-Ergo est capable de prouver le but suivant, en se servant du lemme `length_nonnegative`.

```
goal G3: forall x: int, l: list int. length (Cons x l) > 0
```

C'est là la différence entre les déclarations `goal` et `lemma` : la seconde introduit un but qui pourra être utilisé par la suite, c'est-à-dire qui se retrouvera dans le contexte logique des buts ultérieurs. On peut le vérifier dans l'interface graphique en examinant le contexte du but `G3`.

Cherchons maintenant à définir la notion de liste triée, en commençant par le cas d'une liste d'entiers. On introduit pour cela une nouvelle théorie, `SortedList`, contenant la déclaration d'un prédicat inductif `sorted` [2].

```
theory SortedList
  use import List
  use import int.Int

  inductive sorted (list int) =
  | sorted_nil:
      sorted Nil
  | sorted_one:
      forall x: int. sorted (Cons x Nil)
  | sorted_two:
      forall x y: int, l: list int.
      x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
end
```

Une telle déclaration définit `sorted` comme le plus petit prédicat satisfaisant les trois "axiomes" `sorted_nil`, `sorted_one` et `sorted_two`. On peut maintenant énoncer un but affirmant que la liste [1 ;2 ;3] est triée

```
goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
```

et il sera aisément déchargé par les démonstrateurs automatiques. Cependant, définir `sorted` sur les listes d'entiers uniquement, et qui plus est en se limitant à la relation d'ordre `<=`, n'est pas très satisfaisant. Autant que possible, on préférerait définir `sorted` de manière générique, une fois pour toutes. La logique de Why3 étant une logique du premier ordre, il n'est pas possible de passer la relation d'ordre en argument du prédicat `sorted`. Il existe cependant une alternative. On modifie la théorie `SortedList` pour y utiliser non pas les entiers mais un type `t` et une relation binaire sur `t` non interprétés.

2. On aurait pu définir également le prédicat `sorted` de manière récursive, de la manière suivante :

```
predicate sorted (l: list int) = match l with
  | Nil | Cons _ Nil -> true
  | Cons x (Cons y _ as r) -> x <= y /\ sorted r
end
```

Mais le but est ici de présenter les prédicats inductifs.

```
theory SortedList
  use import List
  type t
  predicate (<=) t t
```

Pour bien faire, on ajoute trois axiomes exprimant qu'il s'agit là d'une relation d'ordre.

```
  axiom le_refl:  forall x: t. x <= x
  axiom le_asym:  forall x y: t. x <= y -> y <= x -> x = y
  axiom le_trans: forall x y z: t. x <= y -> y <= z -> x <= z
```

La définition de `sorted` est alors la même qu'auparavant, à ceci près que le type `int` est remplacé par `t` et la relation `<=` sur les entiers par la relation `<=` sur le type `t`.

```
  inductive sorted (list t) =
  | sorted_nil:
      sorted Nil
  | sorted_one:
      forall x: t. sorted (Cons x Nil)
  | sorted_two:
      forall x y: t, l: list t.
      x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
end
```

Supposons maintenant que l'on souhaite travailler avec des listes d'entiers. On se place dans une nouvelle théorie qui importe les entiers et les listes.

```
theory SortedIntList
  use import int.Int
  use import List
```

Pour utiliser la théorie générique `SortedList` ci-dessus, on va la *spécialiser* sur le type `int` et la relation `<=` sur les entiers. Pour cela, on utilise la commande `clone` plutôt que `use`, de la manière suivante.

```
  clone import SortedList with type t = int, predicate (<=) = (<=),
    lemma le_refl, lemma le_asym, lemma le_trans
```

Cette commande réalise une copie de la théorie `SortedList`, tout en effectuant la substitution du type `t` par `int` et du prédicat non interprété `<=` par le prédicat `<=` sur les entiers (qui portent ici le même nom, mais ce n'est pas une nécessité). Cette commande introduit donc dans le contexte la déclaration d'un *nouveau* prédicat inductif `sorted`, prenant un argument de type `list int`. On peut l'utiliser pour déclarer par exemple le but suivant.

```
  goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
```

On note par ailleurs que la déclaration `clone` ci-dessus indiquait également `lemma le_refl`, `lemma le_asym`, et `lemma le_trans`. Cela signifie que ces trois axiomes doivent être démontrés[3]. Ils deviennent donc autant de buts. On le vérifie notamment en examinant l'ensemble des buts contenus dans le fichier `demo_logic.why`.

3. Le système ne peut inférer seul les axiomes qui deviennent démontrables lors d'une commande `clone`. C'est donc à l'utilisateur de le préciser.

```
why3 -P alt-ergo demo_logic.why
...
demo_logic.why SortedIntList le_refl : Valid (0.00s)
demo_logic.why SortedIntList le_asym : Valid (0.00s)
demo_logic.why SortedIntList le_trans : Valid (0.00s)
demo_logic.why SortedIntList sorted123 : Valid (0.00s)
```

Comme on le comprend sur cet exemple, le concept du clonage de théories permet d'écrire des théories génériques et de les spécialiser plus tard de diverses façons. C'est analogue à la paramétricité dans les langages de programmation, par exemple aux classes génériques de Java ou aux foncteurs de ML. C'est cependant plus souple, dans la mesure où l'on ne fixe pas à l'avance les paramètres. Tout symbole non interprété qui n'est pas substitué pendant le clonage devient un nouveau symbole non interprété. Une théorie est donc paramétrée de multiples façons simultanément.

Dans l'exemple ci-dessus, on peut faire apparaître une autre théorie générique, à savoir celle définissant la notion de relation d'ordre. On y place le type `t`, la relation `<=` et les trois axiomes exprimant que `<=` est une relation d'ordre.

```
theory Order
  type t
  predicate (<=) t t

  axiom le_refl:  forall x: t. x <= x
  axiom le_asym:  forall x y: t. x <= y -> y <= x -> x = y
  axiom le_trans: forall x y z: t. x <= y -> y <= z -> x <= z
end
```

On peut alors la cloner à l'intérieur de la théorie `SortedList`, pour obtenir exactement la même chose qu'auparavant.

```
theory SortedList
  use import List
  clone export Order
  ...
end
```

L'intérêt ici sera de pouvoir réutiliser la théorie `Order` dans d'autres contextes. La figure 2.1 récapitule cette nouvelle conception de la théorie `SortedList` et son utilisation sur des listes d'entiers. D'une manière générale, toute la bibliothèque standard de Why3 est construite de cette façon. Outre la factorisation des définitions logiques, l'intérêt du découpage en petites théories est un meilleur contrôle du contexte logique de chaque but. Avec les démonstrateurs SMT, notamment, limiter la taille du contexte peut avoir un impact significatif sur les performances.

Récapitulatif. La logique de Why3 est une *logique du premier ordre polymorphe*, avec des types algébriques (mutuellement) récursifs, des symboles de fonctions/prédicats (mutuellement) récursifs, des prédicats (mutuellement) inductifs, et des constructions `let-in`,

```
theory List
  type list 'a = Nil | Cons 'a (list 'a)
end

theory Order
  type t
  predicate (<=) t t

  axiom le_refl:  forall x: t. x <= x
  axiom le_asym:  forall x y: t. x <= y -> y <= x -> x = y
  axiom le_trans: forall x y z: t. x <= y -> y <= z -> x <= z
end

theory SortedList
  use import List
  clone export Order

  inductive sorted (list t) =
  | sorted_nil:
      sorted Nil
  | sorted_one:
      forall x: t. sorted (Cons x Nil)
  | sorted_two:
      forall x y: t, l: list t.
      x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))
end

theory SortedIntList
  use import int.Int
  use import List

  clone import SortedList with type t = int, predicate (<=) = (<=),
    lemma le_refl, lemma le_asym, lemma le_trans

  goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
end
```

FIGURE 2.1 – Théorie générique des listes triées.

`match-with`, et `if-then-else`. La syntaxe et la sémantique de cette logique sont données à la fin de ce chapitre, section 2.3. Les déclarations logiques sont de quatre natures différentes :

- déclaration de type ;
- déclaration de fonction ou prédicat ;
- déclaration de prédicat inductif ;
- déclaration d'axiome, lemme ou but.

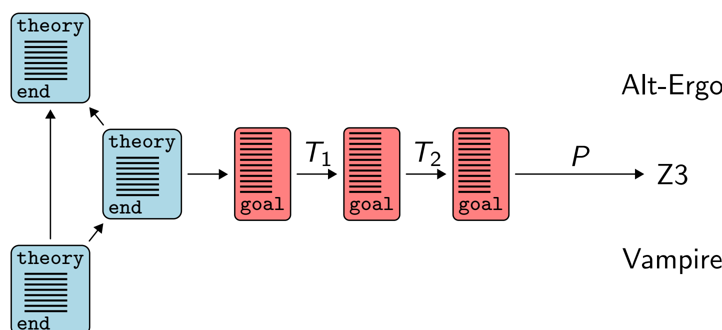
Enfin, les déclarations logiques sont organisées en *théories*. Une théorie T_1 peut être

- utilisée (`use`) dans une théorie T_2. Dans ce cas les symboles de T_1 sont *partagés*, les axiomes de T_1 restent des axiomes, les lemmes de T_1 deviennent des axiomes, et les buts de T_1 sont ignorés.
- ou clonée (`clone`) par une autre théorie T_2. Dans ce cas, les déclarations de T_1 sont *copiées* ou *remplacées*, les axiomes de T_1 restent des axiomes ou deviennent des lemmes/buts, les lemmes de T_1 deviennent des axiomes, et les buts de T_1 sont ignorés.

2.2 Parler à l'oreille des démonstrateurs

L'un des intérêts du système Why3 est d'offrir une technologie pour parler à l'oreille des démonstrateurs. Ces derniers sont nombreux et varient par leurs langages logiques, leurs théories prédéfinies ou leurs systèmes de types. Why3 offre un langage commun et un outil unique pour les utiliser.

Cette technologie est organisée autour de la notion de *tâche*. Une tâche est un contexte logique, c'est-à-dire une liste de déclarations, suivi d'un but, c'est-à-dire une formule. Les tâches sont extraites des différentes théories. Étant donnés une tâche et un démonstrateur cible, on applique une série de transformations à la tâche, pour l'amener jusque dans la logique du démonstrateur. Ainsi, si le démonstrateur est Z3, on appliquera notamment une transformation éliminant les prédicats inductifs car c'est là une notion étrangère à Z3. Une fois toutes les transformations appliquées, il ne reste plus qu'à traduire la tâche obtenue dans la syntaxe native du démonstrateur, à l'aide d'un *pretty-printer*. On peut donc schématiser le parcours d'une tâche de la façon suivante.



Le parcours d'une tâche est piloté par un fichier (appelé *driver* dans la terminologie Why3). Ce fichier indique notamment les transformations à appliquer, le format de sortie du démonstrateur (c'est-à-dire sa syntaxe de sortie et ses symboles et axiomes prédéfinis), et le diagnostique des messages du démonstrateur. Un tel fichier peut être construit par l'utilisateur, par exemple pour ajouter le support d'un nouveau démonstrateur ou mener

des expériences avec un démonstrateur déjà supporté. Des *drivers* sont fournis avec Why3 pour les démonstrateurs suivants : Alt-Ergo, CVC3, Yices, E-prover, Gappa, Simplify, Spass, Vampire, VeriT, Z3, Coq.

D'autre part, le système Why3 offre une interface de programmation (API) pour le langage OCaml. Elle permet notamment de construire des termes, des formules, des déclarations, des tâches, des théories (en garantissant à chaque fois leur bonne formation), et d'appeler des démonstrateurs sur des tâches. Elle permet également d'étendre le système en ajoutant, sous forme de greffons OCaml, des langages d'entrée, des transformations, et des *pretty-printers*. Ainsi le support d'un nouveau démonstrateur est-il grandement facilité.

Plus de détails sont donnés dans l'article *Why3 : Shepherd your herd of provers* [4] ainsi que dans le manuel de Why3 [3].

2.3 Syntaxe et sémantique

On donne ici rapidement quelques éléments de syntaxe et de sémantique de la logique de Why3. Pour plus de détails, on pourra consulter les articles décrivant cette logique [4, 5].

2.3.1 Syntaxe

La figure 2.2 présente la syntaxe abstraite de la logique de Why3. De manière usuelle en logique du premier ordre, cette syntaxe distingue les termes et les formules. Il est important de noter que les formules et les termes se font mutuellement référence, une formule apparaissant dans un terme de la forme if f then t_1 else t_2. Il est également important de noter que les formules ne sont pas des termes de type bool.

2.3.2 Sémantique

La logique du premier ordre est définie formellement dans de nombreux ouvrages. On trouvera par exemple une définition de la logique du premier ordre simplement typée dans l'ouvrage de Manzano *Extensions of first-order logic* [9]. En revanche, la *logique du premier ordre polymorphe* implémentée par Why3 n'a rien de standard. Informellement, le polymorphisme de Why3 est un polymorphisme à la Hindley-Milner [11], au niveau de chaque déclaration. Cela signifie que chaque déclaration polymorphe est un *schéma* pour l'ensemble (potentiellement infini) de toutes ses instances monomorphes. Ainsi la déclaration polymorphe de la fonction `length` sur les listes

```
function length: list 'a : int
```

doit être comprise comme la déclaration d'une infinité de symboles de fonctions :

```
function length_int:      list int        : int
function length_bool:     list bool       : int
function length_list_int: list (list int) : int
...
```

De même, un axiome polymorphe tel que

```
axiom length_nonneg: forall l: list 'a. length l >= 0
```

Termes

t	$::=$	x	variable
	$\mid$	c	constante
	$\mid$	$s\ t \dots t$	application de fonction
	$\mid$	$\mathsf{if}\ f\ \mathsf{then}\ t\ \mathsf{else}\ t$	expression conditionnelle
	$\mid$	$\mathsf{match}\ t\ \mathsf{with}\ p \to t, \dots, p \to t\ \mathsf{end}$	filtrage

Formules

f	$::=$	$s\ t \dots t$	application de prédicat
	$\mid$	$\forall x : \tau . f$	quantification universelle
	$\mid$	$\exists x : \tau . f$	quantification existentielle
	$\mid$	$f \wedge f$	conjonction
	$\mid$	$f \vee f$	disjonction
	$\mid$	$f \to f$	implication
	$\mid$	$f \leftrightarrow f$	équivalence
	$\mid$	$\mathsf{not}\ f$	négation
	$\mid$	true	
	$\mid$	false	
	$\mid$	$\mathsf{if}\ f\ \mathsf{then}\ f\ \mathsf{else}\ f$	expression conditionnelle
	$\mid$	$\mathsf{match}\ t\ \mathsf{with}\ p \to f, \dots, p \to f\ \mathsf{end}$	filtrage

Motifs

p	$::=$	x	variable
	$\mid$	$s\ t \dots t$	application de constructeur
	$\mid$	$_$	motif universel
	$\mid$	$p \mid p$	motif ou
	$\mid$	$p\ \mathsf{as}\ x$	lieur

FIGURE 2.2 – Syntaxe abstraite de la logique de Why3.

doit être compris comme l'ensemble infini des axiomes

```
axiom length_nonneg1: forall l: list int.        length l >= 0
axiom length_nonneg2: forall l: list bool.       length l >= 0
axiom length_nonneg3: forall l: list (list int). length l >= 0
...
```

Un but polymorphe, en revanche, peut toujours être vu comme un but monomorphe, en remplaçant chacune de ses variables de types par un symbole de type frais et non interprété. Une définition précise de la logique du premier ordre polymorphe de Why3 est donnée par F. Bobot et A.Paskevich [5].

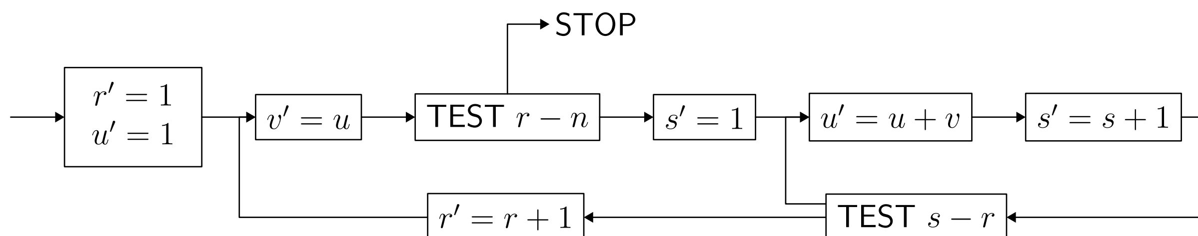
Chapitre 3

Preuve de programmes

Dans ce chapitre, on s'intéresse maintenant à la preuve de programmes avec Why3. Il faut cependant noter que le système Why3 peut parfaitement être utilisé pour sa logique uniquement, en particulier comme une interface unique vers de nombreux démonstrateurs, ainsi que nous l'avons expliqué dans le chapitre précédent.

3.1 Introduction

Comme premier exemple, on va considérer ce qui est sans doute la toute première preuve de programme. Il s'agit d'une preuve de 1949 due à Alan Turing, décrite dans l'article *Checking a Large Routine* [15]. Le programme en question calcule $n!$ en utilisant uniquement des additions. Il est donné sous la forme du graphe de flot de contrôle suivant :



La notation $u' = u+v$ désigne l'affectation de $u+v$ à la variable u, c'est-à-dire $u \leftarrow u+v$ dans une notation plus moderne. La variable n contient le paramètre d'entrée et n'est pas modifiée. Quand le programme termine, la variable u contient $n!$, la factorielle de n. Il y a deux boucles imbriquées. Dans la boucle extérieure, la variable u contient $r!$ et son contenu est copié dans la variable v. La boucle intérieure calcule $(r + 1)!$ dans u, en utilisant des additions répétées de v. Les variables r et s n'étant utilisées que comme compteurs de boucles, on peut facilement réécrire le programme avec deux boucles for. Avec une très légère modification de son flot de contrôle [1], le programme ci-dessus s'écrit donc de la manière suivante :

1. Déterminer laquelle est laissé en exercice au lecteur.

$$
\begin{array}{l}
u \leftarrow 1 \\
\textsf{for } r = 0 \textsf{ to } n-1 \textsf{ do} \\
\quad v \leftarrow u \\
\quad \textsf{for } s = 1 \textsf{ to } r \textsf{ do} \\
\quad\quad u \leftarrow u + v
\end{array}
$$

Considérons l'idée de prouver la correction de ce programme avec Why3. Dans un fichier `demo_turing.mlw` écrivons le programme ci-dessus dans le syntaxe de WhyML, le langage de programmation de Why3. Tout comme les déclarations logiques sont regroupées en théories, les déclarations de programmes sont regroupées en *modules*.

```
module CheckingALargeRoutine
  use import int.Fact
  use import module ref.Ref
```

La première commande importe la théorie logique `int.Fact`, c'est-à-dire la théorie `Fact` de la bibliothèque standard de Why3. Cette théorie fournit un symbole de fonction `fact` dénotant la factorielle. La seconde commande importe un *module* de la bibliothèque standard de WhyML, à savoir le module `Ref` introduisant les références. Comme en ML, les références sont des variables dont le contenu peut être modifié : on crée une référence avec la fonction `ref`, on accède au contenu de la référence `x` avec la notation `!x` et on le modifie avec la notation `x := e`. On peut maintenant écrire le code d'une fonction `routine` prenant un argument entier `n` et calculant sa factorielle à l'aide de l'algorithme ci-dessus.

```
  let routine (n: int) =
    { n >= 0 }
    let u = ref 1 in
    for r = 0 to n-1 do
      let v = !u in
      for s = 1 to r do
        u := !u + v
      done
    done;
    !u
    { result = fact n }
```

Le code en lui-même est ici identique à du code OCaml. Les accolades introduisent la spécification de la fonction `routine`, avec la notation usuelle de la logique de Hoare, à savoir la précondition `n >= 0` et la postcondition `result = fact n`. Dans la postcondition, la variable `result` dénote la valeur renvoyée par la fonction. On exprime donc ici que la fonction `routine` renvoie bien la factorielle de `n`.

Exactement comme nous l'avons fait dans le chapitre précédent avec un fichier contenant des déclarations logiques, on peut analyser un fichier contenant des programmes avec Why3 et même transmettre les buts correspondants à des démonstrateurs. Le programme à invoquer s'appelle `why3ml`.

```
why3ml -P alt-ergo demo_turing.mlw
WP CheckingALargeRoutine WP_parameter routine : Unknown: Unknown (0.08s)
```

On voit ici que le démonstrateur Alt-Ergo n'a pu montrer la correction de la fonction `routine`. Si on préfère l'interface graphique (`why3ide`) à la ligne de commande — ce qui est très rapidement préférable lorsque l'on cherche à prouver des programmes — on peut alors examiner le but qui exprime la correction de la fonction `routine`. Si on omet le contexte des déclarations logiques, celui-ci est réduit à la formule suivante :

$$\forall n,\ n \geq 0 \Rightarrow (0 > n - 1 \Rightarrow 1 = \texttt{fact}\ n) \wedge (0 \leq n - 1 \Rightarrow \forall u,\ u = \texttt{fact}\ n)$$

La première partie de la conjonction correspond au cas où on n'entre même pas dans la première boucle `for`, car $n = 0$. Il faut alors montrer que $1 = \texttt{fact}\ n$, ce qui est prouvable. L'autre partie de la conjonction correspond au cas où on est entré dans la boucle `for`. Il faut alors monter que la valeur finale de la variable `u`, ici notée u, est égale à la factorielle de n. Mais on ne dispose d'aucune information sur u, ce qui rend la formule ci-dessus non prouvable. Pour y remédier, il faut munir la boucle extérieure d'un *invariant* si on veut récupérer plus d'information sur la valeur finale de `u`. En l'occurrence, cet invariant exprime que `u` contient la factorielle de `r`. On l'écrit ainsi :

```
for r = 0 to n-1 do invariant { !u = fact r }
```

Avec cet invariant, il devient possible de montrer la postcondition à la sortie de la fonction `routine`. Pour le voir, on peut utiliser la commande *Split* de l'interface graphique, qui va faire apparaître quatre sous-buts. La validité de la postcondition correspond au dernier des quatre et est facilement déchargée par un démonstrateur automatique. En revanche, on ne parvient pas à décharger le troisième but, qui exprime la préservation de l'invariant de boucle que l'on vient juste d'ajouter. Sans surprise, il nous faut également munir la boucle interne d'un invariant, en l'occurrence celui-ci :

```
for s = 1 to r do invariant { !u = s * fact r }
```

Il est maintenant facile pour un démonstrateur automatique de montrer la correction du programme, c'est-à-dire le respect de sa postcondition et de ses deux invariants de boucle. On note qu'il n'est pas nécessaire de prouver ici la terminaison, car elle est automatiquement garantie pour des boucles `for`. L'intégralité du code de cet exemple est donnée figure 3.1.

Considérons maintenant un autre exemple, à savoir la fonction 91 de McCarthy déjà mentionnée dans l'introduction, page 1. On redonne sa définition :

$$f(n) = \begin{cases} n - 10 & \text{si } n > 100, \\ f(f(n+11)) & \text{sinon.} \end{cases}$$

Dans la syntaxe de WhyML, on définit une telle fonction récursive avec la construction `let rec`.

```
let rec f (n:int) : int =
  if n <= 100 then f (f (n + 11)) else n - 10
```

On cherche ici à prouver la terminaison de cette fonction. Pour cela, il faut la munir d'un *variant*, c'est-à-dire d'une quantité qui décroît à chaque appel récursif pour un ordre bien fondé. Le plus usuel est d'utiliser une expression entière pour l'ordre bien fondé défini par

$$x \prec y \stackrel{\text{def}}{=} x < y \wedge 0 \leq y.$$

```
module CheckingALargeRoutine
  use import int.Fact
  use import module ref.Ref

  let routine (n: int) =
    { n >= 0 }
    let u = ref 1 in
    for r = 0 to n-1 do invariant { !u = fact r }
      let v = !u in
      for s = 1 to r do invariant { !u = s * fact r }
        u := !u + v
      done
    done;
    !u
    { result = fact n }
end
```

FIGURE 3.1 – Calcul de la factorielle à l'aide d'additions (Turing, 1949).

Dans le cas de la fonction 91, le variant est relativement facile à trouver. Il s'agit de la quantité `100 - n`. Dans la syntaxe de WhyML, on l'indique ainsi :

```
  let rec f (n:int) : int variant { 100 - n } =
    if n <= 100 then f (f (n + 11)) else n - 10
```

On obtient alors une obligation de preuve visant à montrer la décroissance du variant pour chacun des deux appels récursifs. Dans le premier cas, le sous-but est facilement déchargé. On a en effet $100-(n+11) \prec 100-n$ lorsque $n \leq 100$. En revanche, le second sous-but ne peut être déchargé, car on ne possède pas d'information sur la valeur renvoyée par le premier appel récursif. Il faut donc munir la fonction `f` d'une spécification. On le fait comme précédemment en insérant une précondition (ici vide) et une postcondition autour du corps de la fonction `f`.

```
  let rec f (n: int) : int variant { 100-n } =
    {}
    if n <= 100 then
      f (f (n + 11))
    else
      n - 10
    { result = if n <= 100 then 91 else n-10 }
```

On obtient alors une obligation de preuve facilement déchargée par les démonstrateurs automatiques. Elle exprime à la fois le respect de la spécification et la terminaison. C'est un exemple où la preuve de terminaison exige la preuve d'une propriété fonctionnelle. (Un autre exemple est celui de l'algorithme de détection de cycle de Floyd, connu aussi sous le nom d'algorithme du lièvre et de la tortue.)

Considérons maintenant une version non récursive de la fonction 91 de McCarthy, à savoir le programme itératif suivant [2] :

```
let iter (n0: int) =
  let n = ref n0 in
  let e = ref 1 in
  while !e > 0 do
    if !n > 100 then begin
      n := !n - 10;
      e := !e - 1
    end else begin
      n := !n + 11;
      e := !e + 1
    end
  done;
  !n
```

L'idée de ce programme est la suivante. Il n'est pas nécessaire de maintenir la pile de tous les appels récursifs à `f`, mais seulement leur nombre. Il est maintenu dans la variable `e`, initialisée à 1. Si $\mathtt{n} > 100$ on soustrait 10 à `n` et on se contente de décrémenter `e`, ce qui signifie qu'on vient de terminer un appel à `f`. Sinon, on ajoute 11 à l'argument `n`, et on incrémente `e` pour signifier qu'on a terminé un appel à `f` et que deux autres doivent être faits.

Prouver la terminaison de cette boucle est à la fois plus simple et plus compliqué que prouver la terminaison de la fonction récursive `f`. C'est plus simple car il n'est pas nécessaire de prouver quoique ce soit à propos de ce que calcule le code. Mais c'est également plus difficile car le variant est maintenant une paire, à savoir

$$(101 - n + 10e,\ e),$$

qui décroît pour l'ordre lexicographique, chaque composante étant ordonnée par la relation $\prec$ ci-dessus. Cette relation lexicographique est définie dans la bibliothèque standard de Why3 et il suffit de l'importer :

```
use import int.Lex2
```

On peut alors munir la boucle `while` de ce variant, avec la syntaxe suivante

```
  while !e > 0 do
    variant { (100 - !n + 10 * !e, !e) } with lex
    ...
```

où le mot clé `with` est utilisé pour spécifier la relation d'ordre. L'obligation de preuve exprimant la décroissance de ce variant est déchargée automatiquement, sans qu'il soit nécessaire d'ajouter d'autre annotation dans le programme.

On pourrait s'arrêter là. Mais de la même façon que la version récursive posait un défi pour la preuve de terminaison, cette version non récursive pose un défi pour la preuve de correction. Il n'est en effet pas si évident que cela que ce programme calcule bien $f(n)$. L'invariant de boucle est

$$f^e(n) = f(n_0)$$

2. La preuve de ce programme m'a été suggérée par Judicaël Courant.

```
use import module ref.Ref
use import int.Lex2

function f (x: int) : int = if x >= 101 then x-10 else 91

clone import int.Iter with type t = int, function f = f

let f91_nonrec (n0: int) =
  { }
  let e = ref 1 in
  let n = ref n0 in
  while !e > 0 do
    invariant { e >= 0 and iter e n = f n0 }
    variant   { (101 - n + 10 * e, e) } with lex
    if !n > 100 then begin
      n := !n - 10; e := !e - 1
    end else begin
      n := !n + 11; e := !e + 1
    end
  done;
  !n
  { result = f n0 }
```

FIGURE 3.2 – Version non récursive de la fonction 91 de McCarthy.

où n_0 désigne la valeur initiale de n. Par conséquent, quand on sort de la boucle, avec $e = 0$, on a $n = f(n_0)$ et on renvoie cette valeur. Pour exprimer cet invariant, on commence par introduire la fonction logique f.

```
function f (x: int) : int = if x >= 101 then x-10 else 91
```

Il ne s'agit pas là de la version récursive mais plutôt de la spécification de la fonction 91 de McCarthy. On définit ensuite $f^k(x)$. Une fonction logique ne peut être définie récursivement que par une récurrence structurelle ; on ne peut donc pas définir $f^k(x)$ par récurrence sur k. On va donc l'axiomatiser.

```
function iter int int : int
axiom iter_0: forall x: int. iter 0 x = x
axiom iter_s: forall k x: int. 0 < k -> iter k x = iter (k-1) (f x)
```

De manière plus générale, la bibliothèque standard de Why3 contient une théorie `int.Iter` qui axiomatise l'itération d'une fonction. On peut donc remplacer les trois déclarations ci-dessus par le clonage approprié de cette théorie.

```
clone import int.Iter with type t = int, function f = f
```

Le code annoté final est donné figure 3.2. Comme pour la version récursive, l'obligation de preuve est aisément déchargée par les démonstrateurs SMT.

3.2 Structures de données

Jusqu'à présent, on s'est limité à des programmes manipulant des entiers. Considérons maintenant des structures de données plus complexes, notamment des tableaux et des types algébriques.

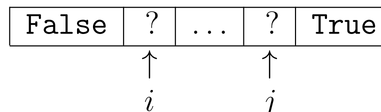
3.2.1 Tableaux

La bibliothèque standard de Why3 fournit des tableaux, dans un module que l'on importe de la façon suivante.

```
use import module array.Array
```

Ce module fournit un type polymorphe `array 'a`, une opération d'accès notée `a[e]`, une opération d'affectation notée `a[e1] <- e2`, et diverses opérations comme `create`, `length`, `append`, `sub` ou `copy`. Comme illustration, considérons la preuve de l'algorithme suivant qui trie un tableau a de n booléens avec la convention `False` $<$ `True`.

$i, j \leftarrow 0, n-1$
while $i < j$
  if $\neg a[i]$ then $i \leftarrow i+1$
  else if $a[j]$ then $j \leftarrow j-1$
  else échanger $a[i]$ et $a[j]$; $i \leftarrow i+1$; $j \leftarrow j-1$



On commence par exprimer la spécification du programme. Il faut notamment exprimer que le tableau a est trié à la fin de l'exécution. On définit pour cela un prédicat `sorted`.

```
predicate le (x y: bool) = x = False \/ y = True
predicate sorted (a: array bool) =
  forall i1 i2: int. 0 <= i1 <= i2 < a.length -> le a[i1] a[i2]
```

Il faut également spécifier que le contenu final du tableau a est une *permutation* de son contenu initial (sans quoi un programme affectant `False` à toutes les cases du tableau serait accepté comme un tri valide). Il y a de multiples façons de définir cette propriété de permutation. On peut par exemple décompter le nombre de valeurs `False` (ou `True`) et exprimer qu'il est inchangé ou, de manière équivalente, exprimer que les multi-ensembles d'éléments sont les mêmes. Une autre solution, parfaitement adaptée au programme de tri ci-dessus, consiste à définir la notion de permutation comme la plus petite relation d'équivalence contenant les transpositions. On définit donc le prédicat `permut` comme un prédicat inductif

```
inductive permut (a1 a2: array bool) =
```

avec d'une part deux clauses exprimant le caractère réflexif et transitif de `permut`

```
| permut_refl:
    forall a: array bool. permut a a
| permut_trans:
    forall a1 a2 a3: array bool.
    permut a1 a2 -> permut a2 a3 -> permut a1 a3
```

et d'autre part une troisième clause indiquant que la relation `permut` contient les transpositions, c'est-à-dire les échanges de deux éléments.

```
  | permut_swap:
      forall a1 a2: array bool. forall i j : int.
      length a1 = length a2 -> 0 <= i < length a1 -> 0 <= j < length a2 ->
      a1[i] = a2[j] -> a1[j] = a2[i] ->
      (forall k: int. 0 <= k < length a1 -> k <> i -> k <> j -> a1[k]=a1[k]) ->
      permut a1 a2
```

On note qu'on n'a pas donné de clause indiquant le caractère symétrique de `permut` (ce qui, en toute généralité, est nécessaire pour définir une relation d'équivalence). Il y a plusieurs raisons à cela : d'une part, on peut montrer que la relation `permut` ainsi définie est symétrique ; d'autre part, cette propriété n'est pas nécessaire pour la preuve de programme qui nous intéresse ici. La définition générale d'un tel prédicat `permut` pour des tableaux contenant des éléments d'un type quelconque fait partie de la bibliothèque standard de Why3, dans le module `array.ArrayPermut`.

On peut maintenant écrire la spécification du programme de tri. Elle prend la forme suivante.

```
  let two_way_sort (a: array bool) =
    { } ... { sorted a /\ permut (old a) a }
```

Ici la notation `old a` dans la postcondition désigne la valeur initiale de `a`, c'est-à-dire sa valeur au point d'entrée de la fonction. Il nous reste à munir le code d'un invariant de boucle. Celui-ci se décompose en quatre parties. On énonce tout d'abord deux relations arithmétiques sur i et j.

```
0 <= !i /\ !j < length a
```

On exprime ensuite que les éléments situés à gauche de i valent tous `False`.

```
(forall k: int. 0 <= k < !i -> a[k] = False)
```

De même, on exprime que les éléments situés à droite de j valent tous `True`.

```
(forall k: int. !j < k < length a -> a[k] = True)
```

Enfin, il faut exprimer la propriété de permutation, c'est-à-dire que le contenu du tableau `a` reste une permutation de son contenu initial. Pour cela, on introduit une étiquette `Init` au début du corps de la fonction, avec la construction `'Init:`, puis on l'utilise dans l'invariant pour faire référence à la valeur de `a` en ce point de programme, avec la notation `at a 'Init`.

```
permut (at a 'Init) a
```

Ceci complète l'invariant de boucle. Par ailleurs, on prouve la terminaison de ce programme de manière immédiate, en utilisant la quantité $j - i$ comme variant. L'intégralité du code annoté est donnée figure 3.3.

Exercice : le drapeau hollandais de Dijkstra. Prouver de même la correction d'un programme triant un tableau contenant trois valeurs différentes, d'un type représentant les trois couleurs du drapeau hollandais défini par `type color = Blue | White | Red`. Le code de ce programme est donné sur la page du cours (voir page 2).

```
use import int.Int
use import bool.Bool
use import module ref.Refint
use import module array.Array
use import module array.ArrayPermut

predicate le (x y: bool) = x = False \/ y = True

predicate sorted (a: array bool) =
  forall i1 i2: int. 0 <= i1 <= i2 < a.length -> le a[i1] a[i2]

let two_way_sort (a: array bool) =
  { }
  'Init:
  let i = ref 0 in
  let j = ref (length a - 1) in
  while !i < !j do
    invariant { 0 <= !i /\ !j < length a /\
      permut (at a 'Init) a /\
      (forall k: int. 0 <= k < !i -> a[k] = False) /\
      (forall k: int. !j < k < length a -> a[k] = True) }
    variant { !j - !i }
    if not a[!i] then
      incr i
    else if a[!j] then
      decr j
    else begin
      let tmp = a[!i] in
      a[!i] <- a[!j];
      a[!j] <- tmp;
      incr i;
      decr j
    end
  done
  { sorted a /\ permut (old a) a }
```

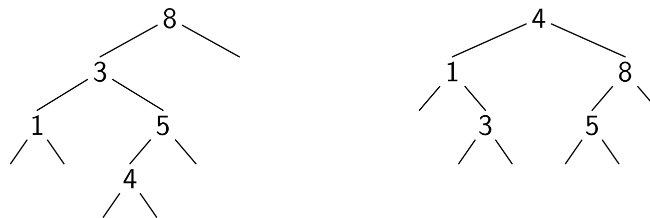
FIGURE 3.3 – Tri d'un tableau de booléens.

3.2.2 Types algébriques

Une *idée centrale* de la logique de Hoare est que tous les types et symboles logiques peuvent être utilisés dans les programmes. En particulier, on l'a déjà fait avec le type `int`. Il en va de même des types algébriques. Dans la bibliothèque de Why3, on trouve notamment les types algébriques suivants :

```
type bool = True | False                          (dans bool.Bool)
type option 'a = None | Some 'a                   (dans option.Option)
type list 'a = Nil | Cons 'a (list 'a)            (dans list.List)
```

Mais l'utilisateur est libre d'en définir de nouveaux et de les utiliser dans des programmes. Illustrons-le sur un exemple. Il s'agit du *problème de la même frange* : étant donnés deux arbres binaires contenant des valeurs aux nœuds, on cherche à déterminer s'ils présentent les mêmes éléments pour un parcours infixe. Le problème est subtil car les deux arbres peuvent avoir des structures différentes. Ainsi les deux arbres



présentent la même suite d'éléments dans un parcours infixe, à savoir $1, 3, 4, 5, 8$. Ce problème a une application pratique : il permet de définir une relation d'ordre sur les arbres binaires de recherche (ABR), ce qui permet de construire des ABR dont les éléments sont des ABR, par exemple pour représenter des ensembles d'ensembles.

Écrivons un programme WhyML pour résoudre le problème de la même frange et prouvons sa correction. On se donne un type non interprété `elt` pour les éléments (leur nature n'a pas d'importance ici) et un type `tree` pour les arbres binaires.

```
type elt
type tree = Empty | Node tree elt tree
```

Pour les besoins de la spécification, on se donne une fonction `elements` qui calcule la liste des éléments d'un arbre pour un parcours infixe.

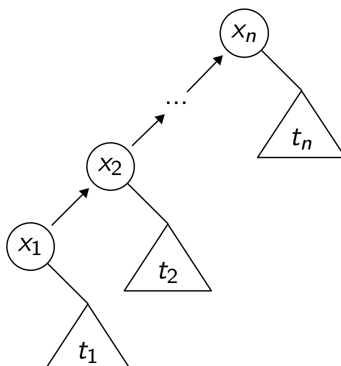
```
function elements (t: tree) : list elt = match t with
| Empty -> Nil
| Node l x r -> elements l ++ Cons x (elements r)
end
```

La fonction que l'on cherche à construire à donc la spécification suivante :

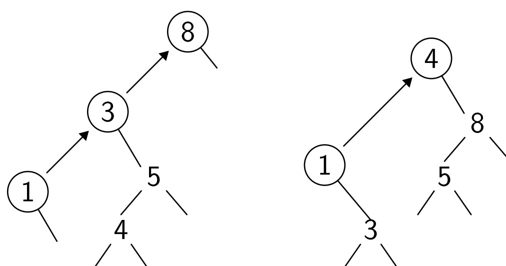
```
let same_fringe t1 t2 =
  { } ... { result=True <-> elements t1 = elements t2 }
```

Bien entendu, on pourrait utiliser directement la fonction `elements` pour résoudre le problème. Ce serait cependant une bien mauvaise solution : d'une part les multiples concaténations de listes coûtent cher (elles peuvent atteindre un coût total quadratique en le nombre d'éléments) ; d'autre part, il est inutilement coûteux de construire l'intégralité des deux listes s'il s'avère qu'elles diffèrent dès les premiers éléments.

On considère ici une bien meilleure solution, qui consiste à voir la branche gauche de chaque arbre comme une liste, de bas en haut. Chaque élément y est associé à son sous-arbre droit. On a donc une liste de paires (x_i, t_i), que l'on peut visualiser ainsi :



Sur l'exemple ci-dessus, les deux listes sont donc les suivantes :



La comparaison procède alors de la manière suivante. On commence par comparer les premiers éléments des deux listes : s'ils diffèrent, la comparaison est terminée. Sinon, on remplace le premier élément de chaque liste par la branche gauche de son sous-arbre droit, puis on recommence la comparaison. Si, à un certain moment, une des deux listes s'épuise avant l'autre, la comparaison échoue. Enfin, la comparaison termine avec succès si les deux listes s'épuisent en même temps.

Pour écrire ce programme en WhyML, on commence par définir un type `enum` pour représenter une liste de paires (par la suite, on appelle une telle liste un énumérateur).

```
type enum = Done | Next elt tree enum
```

Puis on écrit une fonction `enum_elements` qui donne ses éléments en ordre infixe, analogue à la fonction `elements`, toujours dans un but de spécification uniquement.

```
function enum_elements (e: enum) : list elt = match e with
  | Done -> Nil
  | Next x r e -> Cons x (elements r ++ enum_elements e)
end
```

On peut enfin donner le code de la solution au problème de la même frange. On procède en trois temps. On commence par une fonction `enum` qui construit un énumérateur de type `enum` à partir d'un arbre `t` de type `tree`. Le code procède par une descente le long de la branche gauche de `t`. Pour pouvoir procéder récursivement, on passe un second argument `e` de type `enum` à concaténer à la fin.

```
let rec enum t e =
  { }
  match t with
  | Empty -> e
  | Node l x r -> enum l (Next x r e)
  end
  { enum_elements result = elements t ++ enum_elements e }
```

La spécification indique que la liste des éléments du résultat est la concaténation des éléments de `t` et de ceux de `e`. On écrit ensuite une fonction `eq_enum` qui compare les listes données par deux énumérateurs `e1` et `e2` de type `enum`. On procède en examinant les têtes de deux listes, jusqu'à épuisement, comme expliqué ci-dessus.

```
let rec eq_enum e1 e2 =
  { }
  match e1, e2 with
  | Done, Done ->
      True
  | Next x1 r1 e1, Next x2 r2 e2 ->
      x1 = x2 && eq_enum (enum r1 e1) (enum r2 e2)
  | _ ->
      False
  end
  { result=True <-> enum_elements e1 = enum_elements e2 }
```

La spécification de cette fonction indique qu'elle détermine l'égalité des listes représentées par `e1` et `e2`. Enfin, la solution au problème de la même frange est obtenue en appelant `eq_enum` sur les énumérateurs construits à partir des deux arbres.

```
let same_fringe t1 t2 =
  { }
  eq_enum (enum t1 Done) (enum t2 Done)
  { result=True <-> elements t1 = elements t2 }
```

L'intégralité de la solution est donnée figure 3.4.

Exercice : parcours infixe. Avec le même type d'arbres binaires, on considère la fonction récursive suivante qui remplit un tableau a avec les éléments d'un arbre t, en suivant un parcours infixe.

$\texttt{fill}\ t\ start \stackrel{\text{def}}{=}$
  if $t = \texttt{Empty}$ then return $start$
  if $t = \texttt{Node}\ l\ x\ r$ then
    $res \leftarrow \texttt{fill}\ l\ start$
    if $res = \texttt{length}\ a$ then return res
    $a[res] \leftarrow x$
    $\texttt{fill}\ r\ (res + 1)$

Le code WhyML de ce programme, ainsi que les propriétés que l'on souhaite vérifier, sont donnés sur la page du cours (voir page 2).

```
use import int.Int
use import list.List
use import list.Append

type elt
type tree = Empty | Node tree elt tree

function elements (t: tree) : list elt = match t with
  | Empty -> Nil
  | Node l x r -> elements l ++ Cons x (elements r)
end

type enum = Done | Next elt tree enum

function enum_elements (e: enum) : list elt = match e with
  | Done -> Nil
  | Next x r e -> Cons x (elements r ++ enum_elements e)
end

let rec enum t e =
  { }
  match t with
  | Empty -> e
  | Node l x r -> enum l (Next x r e)
  end
  { enum_elements result = elements t ++ enum_elements e }

let rec eq_enum e1 e2 =
  { }
  match e1, e2 with
  | Done, Done ->
      True
  | Next x1 r1 e1, Next x2 r2 e2 ->
      x1 = x2 && eq_enum (enum r1 e1) (enum r2 e2)
  | _ ->
      False
  end
  { result=True <-> enum_elements e1 = enum_elements e2 }

let same_fringe t1 t2 =
  { }
  eq_enum (enum t1 Done) (enum t2 Done)
  { result=True <-> elements t1 = elements t2 }
```

FIGURE 3.4 – Solution au problème de la même frange.

3.2.3 Modélisation

Si on regarde dans la bibliothèque de Why3 le module `array.Array` qui fournit les tableaux, on trouve la déclaration suivante :

```
type array 'a model {| length: int; mutable elts: map int 'a |}
```

Une telle déclaration a une double signification. Dans les *programmes*, elle est analogue à la déclaration d'un type abstrait, qui serait

```
type array 'a
```

Dans la *logique*, en revanche, elle est équivalente à la déclaration d'un type d'enregistrement immuable, qui serait

```
type array 'a = {| length: int; elts: map int 'a |}
```

Autrement dit, un tableau est *modélisé* dans la logique comme un enregistrement contenant d'une part sa longueur, de type `int`, et d'autre part ses éléments sous la forme d'un tableau purement applicatif, de type `map int 'a`.

Le type étant abstrait dans les programmes, on ne peut pas coder d'opérations sur le type `array 'a`. Mais on peut en revanche les *déclarer*. Ainsi l'opération d'accès à un élément d'un tableau est déclarée de la façon suivante :

```
val ([]) (a: array 'a) (i: int) :
  { 0 <= i < a.length } 'a reads a { result = Map.get a.elts i }
```

Il s'agit donc d'une fonction prenant `a` et `i` en arguments, avec une précondition pour garantir l'accès dans les bornes du tableau, une valeur de retour de type `'a`, et une postcondition indiquant que la valeur renvoyée est celle contenue dans le tableau applicatif modélisant `a`, c'est-à-dire `a.elts`. On note en particulier qu'il est possible, à l'intérieur des annotations logiques, d'accéder aux champs `length` et `elts` de l'enregistrement. C'est le but de la modélisation d'être accessible dans la logique (et uniquement dans la logique). L'annotation `reads a` indique que la fonction accède au contenu du tableau `a` en lecture. La fonction d'affectation dans un tableau est déclarée de manière analogue.

```
val ([]<-) (a: array 'a) (i: int) (v: 'a) :
  { 0 <= i < a.length }
  unit writes a
  { a.elts = Map.set (old a.elts) i v }
```

La principale différence est l'annotation `writes a`, qui indique que, cette fois, le contenu de `a` est modifié par la fonction. Ceci est en effet autorisé par le caractère `mutable` du champ `elts` (voir sa déclaration ci-dessus).

À titre d'exemple, montrons comment utiliser le mécanisme ci-dessus pour modéliser des tables de hachage. Plus précisément, considérons des tables de hachage polymorphes, avec l'interface minimale suivante (dans la syntaxe du langage OCaml) :

```
type t 'a 'b
val create: int -> t 'a 'b
val clear: t 'a 'b -> unit
val add:  t 'a 'b -> 'a -> 'b -> unit
exception Not_found
val find: t 'a 'b -> 'a -> 'b
```

La fonction `create` renvoie une nouvelle table, ne contenant aucune entrée. La fonction `clear` vide une table de toutes ses entrées. La fonction `add` ajoute une nouvelle entrée. Enfin la fonction `find` renvoie la valeur associée à une clé dans une table, si elle existe, et lève l'exception `Not_found` sinon.

Comme pour les tableaux ci-dessus, on déclare un type `t 'a 'b` pour une table de hachage en modélisant son contenu par un tableau purement applicatif.

```
type t 'a 'b model {| mutable contents: map 'a (option 'b) |}
```

Ici, ce tableau associe chaque clé k de type `'a` à une valeur de type `option 'b` : `None` désigne l'absence d'entrée pour la clé k, et `Some` v désigne une entrée $k \mapsto v$ dans la table. Pour faciliter les spécifications à venir, on se donne la notation $h[k]$ pour désigner l'entrée de la clé k dans la table h.

```
function ([]) (h: t 'a 'b) (k: 'a) : option 'b = get h.contents k
```

La fonction `create` prend en argument un entier n (la taille initiale du tableau réalisant la table de hachage) et renvoie une nouvelle table de hachage, vide de toute entrée.

```
val create (n:int) :
  { 0 < n } t 'a 'b { forall k: 'a. result[k] = None }
```

Déclarons maintenant la fonction `clear` qui vide une table de toutes ses entrées. Elle a la même postcondition que la fonction `create`. En revanche son type exprime qu'elle modifie la table `h` passée en argument (`writes h`).

```
val clear (h: t 'a 'b) :
  {} unit writes h { forall k: 'a. h[k] = None }
```

La fonction `add` qui ajoute une entrée dans une table a le même effet en écriture.

```
val add (h: t 'a 'b) (k: 'a) (v: 'b) :
  {}
  unit writes h
  { h[k] = Some v /\ forall k': 'a. k' <> k -> h[k'] = (old h)[k'] }
```

Sa postcondition indique d'une part que la clé `k` est associée à la valeur `v` dans le nouveau contenu de `h` et d'autre part que toute autre clé `k'` reste associée à la même valeur qu'auparavant, c'est-à-dire à `(old h)[k']`. L'annotation `writes h` désigne en effet une modification potentielle de *tout* le contenu de la table `h` et c'est donc ici le rôle de la postcondition que d'exprimer ce qui a été réellement modifié. On déclare enfin la fonction `find` qui renvoie la valeur associée à une clé `k` dans une table `h`, si elle existe, et lève l'exception `Not_found` sinon.

```
exception Not_found

val find (h: t 'a 'b) (k: 'a) :
  {}
  'b reads h raises Not_found
  { h[k] = Some result } | Not_found -> { h[k] = None }
```

Ici l'annotation `raises Not_found` signifie que cette fonction est susceptible de lever l'exception `Not_found`. On a donc une double postcondition : une postcondition normale

```
use import option.Option
use import int.Int
use import map.Map

type t 'a 'b model {| mutable contents: map 'a (option 'b) |}

function ([]) (h: t 'a 'b) (k: 'a) : option 'b = get h.contents k

val create (n:int) :
  { 0 < n } t 'a 'b { forall k: 'a. result[k] = None }

val clear (h: t 'a 'b) :
  {} unit writes h { forall k: 'a. h[k] = None }

val add (h: t 'a 'b) (k: 'a) (v: 'b) :
  {}
  unit writes h
  { h[k] = Some v /\ forall k': 'a. k' <> k -> h[k'] = (old h)[k'] }

exception Not_found

val find (h: t 'a 'b) (k: 'a) :
  {}
  'b reads h raises Not_found
  { h[k] = Some result } | Not_found -> { h[k] = None }
```

FIGURE 3.5 – Modélisation de tables de hachage.

`h[k] = Some result`, correspondant au cas où `find` renvoie une valeur, et une post-condition exceptionnelle `h[k] = None`, correspondant au cas où l'exception est levée. La figure 3.5 donne l'intégralité de la modélisation des tables de hachage.

Il est important de signaler que, s'il est possible de coder aussi les tables de hachage, par exemple sous la forme d'un tableau de listes de paires, de type `array (list ('a, 'b))`, il n'est pour l'instant pas possible de faire vérifier par Why3 la conformité de ce code au modèle précédent.

La modélisation n'est pas limitée aux structures de données impératives. On peut aussi l'appliquer à des structures purement applicatives. Par exemple, si on réalise une file purement applicative par une paire de listes, c'est-à-dire

```
type queue 'a = {| front: list 'a; rear : list 'a; |}
```

on peut introduire une fonction pour en modéliser le contenu dans la logique, de la manière suivante :

```
function sequence (q: queue 'a) : list 'a = q.front ++ reverse q.rear
```

Cette fonction ne sera utilisée que dans les annotations de programme, c'est-à-dire dans un but de spécification uniquement.

Un autre exemple est la modélisation de l'arithmétique de la machine. Supposons que l'on veuille modéliser l'arithmétique 32 bits signée, soit pour prouver l'absence de débordement arithmétique dans un programme, soit pour modéliser fidèlement l'arithmétique de la machine. La difficulté est ici de ne pas perdre les capacités arithmétiques des démonstrateurs automatiques. Une solution consiste à introduire un type non interprété `int32` pour les entiers machine

```
type int32
```

et immédiatement une fonction donnant sa valeur, comme un élément de type `int`

```
function toint int32 : int
```

L'idée consiste à n'utiliser que le type `int` dans les annotations de programme, c'est-à-dire en appliquant systématiquement la fonction `toint` aux expressions de type `int32`. On définit la plage des entiers 32 bits avec deux constantes :

```
constant min_int: int = -2147483648
constant max_int: int =  2147483647
```

Un axiome exprime que les valeurs de type `int32` sont comprises entre ces deux bornes.

```
axiom int32_domain:
  forall x: int32. min_int <= toint x <= max_int
```

Si on cherche à définir un modèle pour garantir l'absence de débordement dans les programmes, il suffit d'introduire une fonction de construction d'une valeur de type `int32` avec la précondition adéquate.

```
val ofint (x:int) :
  { min_int <= x <= max_int } int32 { toint result = x }
```

On peut alors traduire un programme utilisant des entiers machine en un programme WhyML où chaque calcul arithmétique est exprimé à l'aide de la fonction `ofint`. Ainsi une expression telle que $x + y$ sera traduite par `ofint` (`toint` x) (`toint` y). De manière équivalente, mais plus agréable, on peut déclarer (ou définir) une fonction effectuant cette opération, c'est-à-dire

```
val (+) (x: int32) (y: int32) :
  { min_int <= toint x + toint y <= max_int }
  int32
  { toint result = toint x + toint y }
```

La page du cours contient une application de cette modélisation à un programme effectuant une recherche dichotomique dans un tableau trié (*binary search*). Ceci permet notamment de trouver l'erreur usuelle consistant à calculer la moyenne de deux indices de tableau `l` et `u` sous la forme `(l+u)/2`, ce qui peut provoquer un débordement arithmétique, puis de montrer qu'en revanche le calcul de cette moyenne sous la forme `l+(u-l)/2` ne pose plus de problème.

3.3 Syntaxe et sémantique

Dans cette dernière section, on décrit formellement la syntaxe et la sémantique de WhyML. La sémantique est ici donnée sous la forme du calcul de plus faibles préconditions.

3.3.1 Syntaxe

La syntaxe abstraite de WhyML est donnée figure 3.6. Sur cette syntaxe, plusieurs constructions sont définies comme du sucre syntaxique. La séquence est la plus simple :

$$e_1;\ e_2 \stackrel{\text{def}}{=} \mathsf{let}\ _ = e_1\ \mathsf{in}\ e_2$$

L'accès à un champ d'enregistrement f peut être ramené à un simple terme de la logique grâce à un let :

$$e_1.f \stackrel{\text{def}}{=} \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ x_1.f$$

En effet, tout terme de la logique peut être utilisé dans les programmes, comme nous l'avons déjà expliqué. De même, tout symbole de prédicat s peut être utilisé dans un programme, avec le sens suivant :

$$\begin{array}{lll} s\ e_1\ \dots\ e_n & \stackrel{\text{def}}{=} & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \dots\ \mathsf{let}\ x_1 = e_1\ \mathsf{in} \\ & & \mathsf{if}\ s\ x_1\ \dots\ x_n\ \mathsf{then}\ \mathit{True}\ \mathsf{else}\ \mathit{False} \end{array}$$

Cela inclut notamment l'égalité et les comparaisons entières.

Un appel de fonction $e_1\ e_2$, avec e_1 de type $x_2 : \tau_2 \rightarrow \kappa_1$, est traduit à l'aide de la construction non-déterministe any :

$$e_1\ e_2 \stackrel{\text{def}}{=} \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ \mathsf{any}\ \kappa_1$$

L'expression any κ_1 représente un calcul dont le type, l'effet et la spécification sont données par κ_1 ; dans un contexte de vérification modulaire, c'est exactement ce qu'un appel de fonction représente. De même, l'affectation d'un champ d'enregistrement f peut être interprétée par la construction any, comme s'il s'agissait d'un appel de fonction :

$$e_1.f \leftarrow e_2 \stackrel{\text{def}}{=} \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ \mathsf{any}\ \{\}\mathit{unit}\ \mathsf{writes}\ x_1.f\{x_1.f = x_2\}$$

Une boucle while est du sucre syntaxique pour une boucle infinie où l'exception prédéfinie Exit est utilisée pour sortir :

$$\begin{array}{l} \mathsf{while}\ e_1\ \mathsf{do}\ e_2\ \mathsf{invariant}\ I\ \mathsf{variant}\ t \stackrel{\text{def}}{=} \\ \quad \mathsf{try}\ \mathsf{loop}\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ \mathsf{raise}\ \mathit{Exit}\ \mathsf{invariant}\ I\ \mathsf{variant}\ t \\ \quad \mathsf{with}\ \mathit{Exit}_ \rightarrow () \end{array}$$

Dans une boucle infinie, le variant est du sucre syntaxique pour une assertion placée à la fin du corps de la boucle :

$$\mathsf{loop}\ e\ \mathsf{invariant}\ f\ \mathsf{variant}\ t \stackrel{\text{def}}{=} \mathsf{loop}\ L : e;\ \mathsf{assert}\ t \prec \mathsf{at}\ t\ L\ \mathsf{invariant}\ f$$

e	$::=$	x	variable
	$\mid$	t	terme
	$\mid$	$\mathsf{let}\ x\ =\ e\ \mathsf{in}\ e$	liaison locale
	$\mid$	$\mathsf{let\ rec}\ d, \dots, d\ \mathsf{in}\ e$	liaison locale
	$\mid$	$\mathsf{fun}\ d$	fonction anonyme
	$\mid$	$\mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e$	conditionnelle
	$\mid$	$\mathsf{loop}\ e\ \mathsf{invariant}\ f$	boucle infinie
	$\mid$	$\mathsf{for}\ x = x\ \mathsf{to}\ x\ \mathsf{do}\ e\ \mathsf{invariant}\ f$	boucle for
	$\mid$	$\mathsf{match}\ x\ \mathsf{with}\ p \to e, \dots, p \to e\ \mathsf{end}$	filtrage
	$\mid$	$\mathsf{raise}\ (E\ e)$	levée d'exception
	$\mid$	$\mathsf{try}\ e\ \mathsf{with}\ E\ x \to e, \dots, E\ x \to e$	gestionnaire d'exception
	$\mid$	$L : e$	étiquette
	$\mid$	$\mathsf{assert}\ f$	assertion (coupure)
	$\mid$	$\mathsf{check}\ f$	assertion
	$\mid$	$\mathsf{assume}\ f$	hypothèse
	$\mid$	absurd	absurdité
	$\mid$	$\mathsf{any}\ \kappa$	non-déterminisme
d	$::=$	$x : \tau, \dots, x : \tau = \{f\}\ e\ \{q\}$	corps de fonction
τ	$::=$	α	variable de type
	$\mid$	$s\ r, \dots, r\ \tau, \dots, \tau$	application de type
	$\mid$	$x : \tau \to \kappa$	type de fonction
κ	$::=$	$\{f\}\ \tau\ \epsilon\ \{q\}$	spécification
ϵ	$::=$	$\mathsf{reads}\ r, \dots, r\ \mathsf{writes}\ r, \dots, r$	
		$\mathsf{raises}\ E, \dots, E$	effect
q	$::=$	$f, E \to f, \dots, E \to f$	postcondition

FIGURE 3.6 – Syntaxe abstraite de WhyML.

Une étiquette fraîche L est utilisée pour dénoter le début du corps de la boucle, de manière à pouvoir exprimer la décroissance du variant (ici $\prec$ dénote la relation d'ordre associée au variant t).

Les variants des fonctions récursives sont également du sucre syntaxique. On utilise les préconditions des fonctions pour insérer les assertions correspondantes. On procède en deux temps. En premier lieu, on sauvegarde la valeur des variants aux points d'entrée des fonctions, dans des variables locales v_i :

$$\begin{array}{l} \mathsf{let\ rec}\ f_1\ \vec{x_1}\ \mathsf{variant}\ t_1 = e_1 \\ \mathsf{and}\ f_2\ \vec{x_2}\ \mathsf{variant}\ t_2 = e_2 \\ \dots \\ \mathsf{and}\ f_n\ \vec{x_n}\ \mathsf{variant}\ t_n = e_n \\ \mathsf{in}\ e \end{array} \stackrel{\text{def}}{=} \begin{array}{l} \mathsf{let\ rec}\ f_1\ \vec{x_1} = \mathsf{let}\ v_1 = t_1\ \mathsf{in}\ e_1 \\ \mathsf{and}\ f_2\ \vec{x_2} = \mathsf{let}\ v_2 = t_2\ \mathsf{in}\ e_2 \\ \dots \\ \mathsf{and}\ f_n\ \vec{x_n} = \mathsf{let}\ v_n = t_n\ \mathsf{in}\ e_n \\ \mathsf{in}\ e \end{array}$$

On type ensuite chaque corps de fonction e_i dans un environnement où chaque fonction f_j a le type

$$f_j : \vec{x_j} \to \{t_j(\vec{x_j}) \prec v_i \wedge p_j\}\tau_j\epsilon_j\{q_j\}$$

où $\prec$ est la relation d'ordre associée aux variants t_i (elle est unique).

Enfin, les opérateurs `&&` et `||` sont du sucre syntaxique pour des expressions conditionnelles.

$$e_1\ \texttt{\&\&}\ e_2 \stackrel{\text{def}}{=} \begin{cases} \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathit{andb}\ x_1\ e_2 & \text{if } e_2 \text{ is pure,} \\ \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ \mathit{false} & \text{otherwise.} \end{cases}$$

On réalise cependant un cas particulier lorsque e_2 est pure (c'est-à-dire sans autre effets que des lectures) pour limiter le nombre d'expressions conditionnelles. On obtient ainsi des obligations de preuve moins nombreuses et plus intuitives. L'opération `||` est interprétée de manière similaire.

3.3.2 Typage

Le but du typage de WhyML est triple :

1. un typage ML traditionnel ;
2. une inférence des effets ;
3. une exclusion des alias.

Concernant le premier point, on utilise un algorithm W standard [11]. On évite le problème des références polymorphes de la manière suivante : les types ne sont pas généralisés au niveau des liaisons locales et les liaisons globales ne lient que des valeurs. Ainsi la règle de la restriction aux valeurs [16] s'applique aisément.

Le deuxième point est également très classique : il revient à calculer les ensembles de régions possiblement lues et modifiées pour chaque sous-expression de programme, ainsi que l'ensemble des exceptions possiblement levées. On applique l'analyse de Talpin et Jouvelot *the type and effect discipline* [13, 14] de manière immédiate. Les effets résultants sont utilisés pour calculer les plus faibles préconditions.

Le dernier point est la détection, et le rejet subséquent, des alias de régions. C'est une nécessité pour garantir la correction du calcul de plus faibles préconditions, car ce dernier suppose que des régions distinctes dénotent des emplacements mémoires distincts. Il y

a deux idées différentes. D'une part, les fonctions sont polymorphes vis-à-vis des effets, exactement comme elles sont polymorphes vis-à-vis des types. Ainsi une fonction qui échange le contenu de deux références a pour type

$$\texttt{swap} : \forall r_1, r_2.\ \forall \alpha.\ \texttt{ref}\ r_1\ \alpha \to \texttt{ref}\ r_2\ \alpha \to\ \texttt{unit}$$

(on omet ici la spécification pour plus de clarté). D'autre part, quand on définit et que l'on prouve une telle fonction, on fait l'hypothèse que les régions r_1 et r_2 sont distinctes. Par conséquent, quand on applique cette fonction à deux références, on vérifie que les régions correspondantes sont effectivement distinctes. Une expression telle que

$$\mathsf{let}\ r = \texttt{ref}\ 0\ \mathsf{in}\ \texttt{swap}\ r\ r$$

est donc rejetée. Une fonction prenant deux fois la même référence en argument devrait donc avoir un type différent, de la forme

$$\texttt{swap'} : \forall r.\ \forall \alpha.\ \texttt{ref}\ r\ \alpha \to \texttt{ref}\ r\ \alpha \to\ \texttt{unit}.$$

Cependant, pour maintenir un système aussi simple que possible (sans être trop simple), les régions ne sont pas exposées à l'utilisateur de Why3. Pour que cela soit possible, un compromis est fait. D'une part, les régions apportées par les arguments d'une fonction doivent être toutes distinctes. La fonction `swap'` ci-dessus n'est donc pas définissable (mais de telles fonctions sont peu utiles). D'autres part, les régions renvoyées par une fonction doivent être fraîches, c'est-à-dire allouées durant l'appel de fonction. Ceci exclut une fonction comme l'identité polymorphes sur les références :

$$\texttt{id} : \forall r.\ \forall \alpha.\ \texttt{ref}\ r\ \alpha \to \texttt{ref}\ r\ \alpha.$$

Mais ce sont là encore des fonctions de peu d'utilité. Ces deux conditions sont vérifiées par le typage de WhyML.

3.3.3 Plus faibles préconditions

Étant donnée une expression de programme e, une postcondition f, et un ensemble $\vec{q}$ de postconditions exceptionnelles $E_1 \to f_1, \dots, E_n \to f_n$, on définit la plus faible précondition $\mathsf{wp}(e, f, \vec{q})$ par récurrence sur la structure de e.

Constructions purement applicatives. Le cas d'une variable est immédiat :

$$\mathsf{wp}(x, f, _) \stackrel{\text{def}}{=} f[\mathit{result} \leftarrow \lceil x \rceil],$$

ainsi que celui d'un terme logique :

$$\mathsf{wp}(t, f, _) \stackrel{\text{def}}{=} f[\mathit{result} \leftarrow t].$$

La plus faible précondition d'un filtrage n'est autre que le filtrage des plus faibles préconditions :

$$\begin{array}{l}\mathsf{wp}(\mathsf{match}\ x\ \mathsf{with}\ p_1 \to e_1, \dots, p_n \to e_n\ \mathsf{end}, f, \vec{q}) \stackrel{\text{def}}{=} \\ \quad \mathsf{match}\ x\ \mathsf{with}\ p_1 \to \mathsf{wp}(e_1, f, \vec{q}), \dots, p_n \to \mathsf{wp}(e_n, f, \vec{q})\ \mathsf{end}.\end{array}$$

Bien entendu, cela n'est possible que parce la logique de Why3 fournit également la construction de filtrage.

Séquence et conditionnelle. Séquence et conditionnelle suivent les règles usuelles. La séquence est généralisée au cas d'une liaison `let` :

$$\mathsf{wp}(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, \mathsf{wp}(e_2, f, \vec{q})[\lceil x \rceil \leftarrow \mathit{result}], \vec{q}).$$

Dans le cas particulier d'une vraie séquence $e_1; e_2$, c'est-à-dire lorsque x n'apparaît pas dans e_2, l'expression se simplifie comme attendu :

$$\mathsf{wp}(e_1;\ e_2, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, \mathsf{wp}(e_2, f, \vec{q}), \vec{q}).$$

Lorsque e_1 et e_2 ne peuvent lever d'exception, elle se simplifie encore plus, jusqu'à la règle traditionnelle :

$$\mathsf{wp}(e_1;\ e_2, f) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, \mathsf{wp}(e_2, f)).$$

Le cas de la conditionnelle est également standard :

$$\mathsf{wp}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, \mathsf{if}\ \mathit{result}\ \mathsf{then}\ \mathsf{wp}(e_2, f, \vec{q})\ \mathsf{else}\ \mathsf{wp}(e_3, f, \vec{q}), \vec{q}).$$

Cependant, dans le cas particulier où e_2 et e_3 sont deux termes logiques (ou deux variables), on évite la duplication de f en utilisant un terme conditionnel :

$$\mathsf{wp}(\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e_1, f[\mathit{result} \leftarrow \mathsf{if}\ \mathit{result}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3], \vec{q}).$$

Assertions. Les règles associées aux assertions se dérivent simplement. Une assertion introduite avec assert s'ajoute à la postcondition :

$$\mathsf{wp}(\mathsf{assert}\ f_1, f_2, _) \stackrel{\text{def}}{=} f_1 \wedge f_2.$$

En pratique, une telle conjonction est étiquetée de manière à permettre plus tard un découpage de cette formule en f_1 et $f_1 \Rightarrow f_2$, plutôt que f_1 et f_2. Ainsi, f_1 assume le rôle de coupure logique, ce qui est le but de assert. Une variante de assert, notée check, produit une conjonction symétrique. Une assertion introduite avec assume devient une hypothèse :

$$\mathsf{wp}(\mathsf{assume}\ f_1, f_2, _) \stackrel{\text{def}}{=} f_1 \Rightarrow f_2.$$

Enfin, la plus faible précondition de la construction absurd est tout simplement false :

$$\mathsf{wp}(\mathsf{absurd}, _, _) \stackrel{\text{def}}{=} \mathsf{false}.$$

Étiquettes. Quand une étiquette est rencontrée, on se contente de l'effacer dans les annotations. On définit donc

$$\mathsf{wp}(L : e, f, \vec{q}) \quad \stackrel{\text{def}}{=} \quad \mathsf{wp}(e, f, \vec{q})[\mathsf{at}\ t\ L \leftarrow t]$$

où la substitution parcourt tous les termes de la forme at t L dans la plus faible précondition.

Exceptions. Soit $\vec{q} = E_1 \to f_1, \ldots, E_n \to f_n$ l'ensemble des postconditions exceptionnelles. Quand on considère l'expression raise $(E_i\ e)$, on peut toujours supposer que E_i appartient à $\vec{q}$ — dans le cas contraire, il suffit d'étendre $\vec{q}$ avec la postcondition true pour E_i. Dès lors on a

$$\mathsf{wp}(\mathsf{raise}\ (E_i\ e), _, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e, f_i, \vec{q}).$$

Pour la construction try with, on peut supposer que les exceptions rattrapées sont $E_1, \ldots, E_m$, avec $m \leq n$, sans perte de généralité. On a alors

$$\begin{array}{l}\mathsf{wp}(\mathsf{try}\ e\ \mathsf{with}\ E_1\ x_1 \to e_1, \ldots, E_m\ x_m \to e_m, f, \vec{q}) \stackrel{\text{def}}{=} \\ \quad \mathsf{wp}(e, f, (E_1 \to \mathsf{wp}(e_1, f_1, \vec{q}), \ldots, E_m \to \mathsf{wp}(e_m, f_m, \vec{q}), \vec{q}_{m+1..n})).\end{array}$$

Une nouvelle quantification. Avant d'aller plus loin, il faut introduire une nouvelle opération. Dans la logique de Hoare traditionnelle, chaque fois qu'il faut généraliser la plus faible précondition par rapport à l'état, il suffit de quantifier universellement sur toutes les variables du programme (données par l'effet). Dans notre cas, cependant, il s'agit d'une opération plus complexe. La raison est que certaines *parties* seulement des structures de données peuvent être modifiées. En conséquent, on doit introduire de nouvelles valeurs pour ces éléments-là et reconstruire ensuite toutes les valeurs dont ils font parties.

Soit f une formule (typiquement la plus faible précondition en cours de calcul) et $\vec{r} = r_1, \ldots, r_n$ un ensemble de régions (typiquement l'effet *writes* d'une sous-expression). On définit la formule $\nabla\vec{r}.\,f$ de la manière suivante. Soit $v_1, \ldots, v_m$ l'ensemble des variables libres de f dont les types contiennent au moins une région de $\vec{r}$. On pose alors

$$\nabla\vec{r}.\,f \ \stackrel{\text{def}}{=}\ \forall\vec{r}.\ \begin{array}{l}\mathsf{let}\ v_1 = \mathsf{update}\ v_1\ \vec{r}\ \mathsf{in} \\ \ldots \\ \mathsf{let}\ v_n = \mathsf{update}\ v_n\ \vec{r}\ \mathsf{in} \\ f\end{array}$$

La valeur de chaque variable v_i est reconstruire à l'aide de la fonction update. Cette fonction décompose l'ancienne valeur et reconstruit ses champs un par un :

$$\mathsf{update}\ v\ \vec{r}\ \stackrel{\text{def}}{=}\ \begin{array}{l}\mathsf{match}\ v\ \mathsf{with} \\ C\ y_1 \ldots\ y_m \to C\ (\mathsf{update}_1\ y_1\ \vec{r}) \ldots\ (\mathsf{update}_m\ y_m\ \vec{r}).\end{array}$$

C est le constructeur du type de v, qui existe puisque v contient des régions dans son type. La fonction update_i reconstruit la valeur y_i du champ i, ainsi :

$$\mathsf{update}_i\ y_i\ \vec{r}\ \stackrel{\text{def}}{=}\ \begin{cases} r_j & \text{si } r_j \text{ est la région du champ } i, \\ y_i & \text{si } \vec{r} = \emptyset, \\ \mathsf{update}\ y_i\ (\vec{r} \cap reg(y_i)) & \text{sinon.}\end{cases}$$

Ici $reg(y_i)$ dénote l'ensemble des régions apparaissant dans le type de y_i. Voici un exemple. Supposons deux types d'enregistrement *array* et *state* ainsi déclarés :

$$\begin{array}{l}\mathsf{type}\ array\ r\ \alpha = \{\mathsf{mutable}\ elts : map\ int\ \alpha\} \\ \mathsf{type}\ state\ r_1\ r_2\ = \{a : array\ r_1\ bool;\ b : array\ r_2\ real\}\end{array}$$

Soit s une variable de type *state* r_1 r_2 et f une formule. On suppose que s est la seule variable de f dont le type contient r_1. Alors

$$\begin{array}{rl}\nabla r_1.\, f = & \forall r_1 : \mathit{map\ int\ bool}. \\ & \mathsf{let}\ s = \mathsf{match}\ s\ \mathsf{with} \\ & \quad C\ y_1\ y_2 \to C\ (\mathsf{match}\ y_1\ \mathsf{with}\ C'_ \to C'\ r_1)\ b\ \mathsf{in} \\ & f.\end{array}$$

Boucles. La règle pour la boucle infinie est la suivante :

$$\mathsf{wp}(\mathsf{loop}\ e\ \mathsf{invariant}\ I, _, \vec{q}) \stackrel{\text{def}}{=} I \wedge \nabla \mathit{writes}(e).\, I \Rightarrow \mathsf{wp}(e, I, \vec{q}).$$

On note que la postcondition normale — le deuxième argument de wp — n'est pas utilisé ici car la boucle ne peut pas terminer normalement. En revanche, on peut sortir de la boucle en levant une exception, d'où l'utilité de $\vec{q}$. Le cas d'une boucle for est plus subtil. C'est la raison pour laquelle cette construction n'est pas traitée comme du sucre syntaxique.

$$\begin{array}{l}\mathsf{wp}(\mathsf{for}\ x = x_1\ \mathsf{to}\ x_2\ \mathsf{do}\ e\ \mathsf{invariant}\ I(x), f, \vec{q}) \stackrel{\text{def}}{=} \\ \quad x_1 > x_2 \Rightarrow f \\ \wedge \quad x_1 \leq x_2 \Rightarrow \quad I(x_1) \\ \qquad\qquad\qquad \wedge \quad \nabla \mathit{writes}(e). \quad \forall i. x_1 \leq i \leq x_2 \Rightarrow I(i) \Rightarrow \mathsf{wp}(e, I(i+1), \vec{q}) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad I(x_2 + 1) \Rightarrow f\end{array}$$

On distingue le cas où on n'entre pas du tout dans la boucle for, c'est-à-dire lorsque $x_1 > x_2$. Sinon, I doit être vérifié initialement (c'est-à-dire $I(x_1)$), doit être préservé par e (c'est-à-dire, $I(i)$ doit impliquer $I(i+1)$, pour ainsi dire), et doit impliquer f à la sortie de la boucle. Ces deux dernières conditions doivent être vérifiées dans un état frais, d'où l'utilisation de l'opération ∇. On note que la dernière condition mentionne $I(x_2 + 1)$, et non $I(x_2)$. En effet, il y a $x_2 - x_1 + 1$ états différents dans la boucle for et on adopte la convention que $I(x)$ dénote l'état au début du corps de la boucle, comme dans les autres boucles.

Appel de fonction. Un appel de fonction est interprété en utilisant la construction non-déterministe any. L'expression any $\{p\}$ τ ϵ $\{q\}$ dénote un calcul arbitraire de précondition p, de type τ, d'effet ϵ, et de postcondition q. Quand on calcule la plus faible précondition, on peut supposer que les deux postconditions font référence au même ensemble d'exceptions, puisqu'il est toujours possible d'étendre l'un ou l'autre avec des postconditions true. On pose alors

$$\begin{array}{l}\mathsf{wp}(\mathsf{any}\ \{f_0\}\ \tau\ \epsilon\ \{f_1, \vec{q_1}\}, f_2, \vec{q_2}) \stackrel{\text{def}}{=} \\ \quad f_0 \wedge \nabla \mathit{writes}(\epsilon).\, (\forall r : \tau.\, f_1 \Rightarrow f_2) \wedge \bigwedge_i (\forall r : \tau_i.\, q_{1,i} \Rightarrow q_{2,i})\end{array}$$

où τ_i est le type de la valeur portée par l'exception E_i. Il y a une seule quantification $\nabla \mathit{writes}(\epsilon)$, pour les postconditions normale et exceptionnelles. En effet, on ne fait pas de distinction entre effets en écriture pour une évaluation normale et une évaluation exceptionnelle.

Définition de fonction. La plus faible précondition pour une fonction anonyme se réduit à la vérification de sa définition :

$$\mathsf{wp}(\mathsf{fun}\ d, _, _) \stackrel{\text{def}}{=} \mathsf{correct}(d)$$

En effet, le résultat de cette expression étant une fonction, il ne peut apparaître dans la postcondition. Le prédicat correct est défini ci-dessous. Les fonctions récursives, quant à elles, sont nécessairement liées à une expression. En conséquence, la plus faible précondition contient deux parties :

$$\mathsf{wp}(\mathsf{letrec}\ d_1, \dots, d_n\ \mathsf{in}\ e, f, \vec{q}) \stackrel{\text{def}}{=} \mathsf{wp}(e, f, \vec{q}) \wedge \bigwedge_i \mathsf{correct}(d_i)$$

Le prédicat correct exprime qu'une définition de fonction respecte ses annotations. Il est défini en utilisant le prédicat wp :

$$\begin{array}{l}\mathsf{correct}(x_1 : \tau_1, \dots x_n : \tau_n \ = \{p\}\ e\ \{f, \vec{q}\}) \stackrel{\text{def}}{=} \\ \quad \forall x_1, \dots, x_n. \nabla \mathit{writes}(e) \cup \mathit{reads}(e).\, p \Rightarrow \mathsf{wp}(e, f, \vec{q})\end{array}$$

Il y a une subtilité ici : contrairement à ce qui a été fait plus haut pour les constructions loop, for, et any, on quantifie ici également sur $\mathit{reads}(e)$. On exprime ainsi qu'une fonction peut être utilisée dans un contexte différent de celui de sa définition.

Déclarations. En WhyML, les déclarations sont limitées aux déclarations et aux définitions de fonctions. Une déclaration ne produit aucune obligation de preuve. Une définition de fonction est soit de la forme let d, soit de la forme letrec $d_1, \dots, d_n$ et est traduite en une obligation de preuve à l'aide du prédicat correct, exactement comme pour les définitions locales. L'obligation est transformée en une formule close en quantifiant universellement sur toutes les variables du programme.

Bibliographie

[1] Manuel Barbosa, Jean-Christophe Filliâtre, Jorge Sousa Pinto, and Bárbara Vieira. A deductive verification platform for cryptographic software. In *4th International Workshop on Foundations and Techniques for Open Source Software Certification (OpenCert 2010)*, volume 33, Pisa, Italy, September 2010. Electronic Communications of the EASST.

[2] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Zhong Shao and Benjamin C. Pierce, editors, *POPL*, pages 90–101, Savannah, GA, USA, January 2009. ACM Press.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 platform*. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, February 2011. `http://why3.org/`.

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

[5] François Bobot and Andrei Paskevich. Expressing Polymorphic Types in a Many-Sorted Language. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 87–102, Saarbrücken, Germany, October 2011.

[6] Jean-Christophe Filliâtre and Claude Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods*, pages 15–29. Springer, 2004.

[7] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580 and 583, October 1969.

[8] Zohar Manna and John McCarthy. Properties of programs and partial function logic. 5 :79–98, 1970.

[9] María Manzano. *Extensions of first order logic*. Cambridge University Press, New York, NY, USA, 1996.

[10] Claude Marché, Christine Paulin, and Xavier Urbain. The Krakatoa proof tool, 2002. `http://krakatoa.lri.fr/`.

[11] R. Milner. A theory of type polymorphismn in programming. *Journal of Computer and System Sciences*, 17, 1978.

[12] Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009.

[13] Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3) :245–271, 1992.

[14] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(4) :245–298, June 1994.

[15] Alan Mathison Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculing Machines*, pages 67–69, Cambridge, 1949. Mathematical Laboratory.

[16] Andrew K. Wright. Simple imperative polymorphism. *LISP and symbolic computation*, 8 :343–355, 1995.