TD 1 : Exprimer formellement des propriétés

Yann Régis-Gianas (yrg@pps.jussieu.fr)
GL6 - Licence 3 - Université Paris 7

3 février 2009

L'objectif de ces travaux dirigés est de s'initier à l'utilisation d'un formalisme logique pour décrire des propriétés portant sur des objets [1] utilisés en programmation. Après quelques rappels autour de la syntaxe des formules logiques, nous allons faire varier le domaine des objets sur lesquels travaille la logique pour s'initier aux problèmes d'expressivité : Est-ce que cette formule logique représente la notion intuitive que j'ai en tête ? Est-ce que le domaine des objets est suffisamment riche pour exprimer ma propriété ?

Dans la première partie de ce cours dédié à la spécification des programmes séquentiels, nous utiliserons la plateforme Why qui sert à vérifier la conformité d'un programme vis-à-vis de sa spécification. Il est disponible à l'adresse :

http://why.lri.fr

Pour commencer à nous initier à son usage, nous utiliserons désormais la syntaxe des formules logiques de Why (décrite dans la section 1).

Au terme de cette séance, vous devrez :

- ✓ avoir assimilé la syntaxe des déclarations logiques de Why ;
- ✓ avoir (re)vu le sens des connecteurs et des quantificateurs logiques ;
- ✓ être sensibilisé aux difficultés générales de la formalisation d'une spécification écrite en langage naturel ;
- ✓ avoir noté les différences d'expressivité entre différentes axiomatisations des entiers et des séquences ;
- ✓ avoir forgé une première intuition du mécanisme d'induction.

1 Syntaxe de Why

Why permet de définir un cadre logique en s'appuyant sur des **types abstraits**, des **opérations** et des **prédicats** sur ces types ainsi que des **axiomes**.

Type abstrait Un type abstrait est un nom qu'on se donne pour catégoriser les objets. Dans nos formalisations, nous supposerons qu'à tout type t est associé un **domaine d'interprétation**. Il s'agit d'un ensemble mathématique dont les éléments sont représentés par les objets de type t. La syntaxe Why pour déclarer un type t est :

type [nom-de-type]

Why permet aussi de définir des familles de type (des types paramétrés) mais nous n'utiliserons pas cette fonctionnalité pour le moment.

Symbole de fonction Un fois qu'on s'est donné un type abstrait, on ne peut pas construire de valeur de ce type tant qu'on ne se donne pas de symboles de fonction. Les symboles de fonction d'arité [2] 0 seront appelés constantes. À partir de ces constantes et de symboles de fonction d'arité positive, on peut construire une infinité de représentation d'objet (les termes du premier ordre). On supposera qu'à chaque symbole est associé une interprétation dans le domaine. Voici la syntaxe de définition des symboles de fonction en Why [3] :

logic c : [nom-de-type] *(* Constante. *)*
logic f : [nom-de-type] , . . . , [nom-de-type] → [nom-de-type] *(* Fonction. *)*

Par exemple, la spécification Why suivante permet de se donner un type pour représenter $(\mathbb{Z}/3\mathbb{Z}, +, *)$:

[1] Ici, on parle d'objet au sens "classique" c'est-à-dire comme "propos du discours logique" et non au sens de la programmation orientée objet.
[2] Arité : nombre d'arguments
[3] Encore une fois, il ne s'agit ici que d'une restriction des types de Why à nos besoins.

```
type z3z
logic zero :  z3z
logic one :  z3z
logic two :  z3z
logic plus :  z3z, z3z → z3z
logic mult :  z3z, z3z → z3z
```

Prédicat Pour caractériser les objets, on se donne des prédicats typés. Nous les interpréterons comme des fonctions totales à valeur dans les booléens. Déclarer des prédicats se fait ainsi :

```
logic f :  [nom-de-type] , ..., [nom-de-type] → prop
```

Si on veut se donner un prédicat pour caractériser un élément neutre pour l'addition, on peut écrire :

```
logic is_neutral :  z3z → prop
```

Axiomes Pour raisonner sur les objets déclarés dans Why, il faut se donner des formules logiques supposées valides (du point de vue du domaine d'interprétation) qui joueront le rôle de point de départ à la vérification.

Cet ensemble d'axiomes représente la **base de confiance**. *Il est essentiel que ces formules soient des théorèmes, c'est-à-dire des formules valides mathématiquement !* Dans le cas contraire, les résultats de la démonstration n'auraient aucun valeur. Le choix de ces axiomes doit donc se faire de manière très méticuleuse [4] La syntaxe pour les déclarer est :

```
axiom [nom] : [formule]
```

Buts Why peut s'utiliser comme un démonstrateur automatique de la validité d'une formule. Pour cela, il fait appel à des **prouveurs automatiques**. Ceux-ci sont nécessairement **incomplets**. Néanmoins, leur efficacité a énormément progressé ces dernières années. Pour faire une requête de validité d'une formule sous les hypothèses déclarées jusqu'à un certain point du fichier, on écrit :

```
goal [nom] : [formule]
```

On peut effectuer plusieurs requêtes dans un même fichier. Une fois prouvée, une formule est rajoutée à l'ensemble des hypothèses pour effectuer les preuves suivantes.

La syntaxe des formules suit la logique du premier ordre :

```
true (* Formule vraie *)
false (* Formule fausse *)
[formule] ↔ [formule] (* Équivalence *)
[formule] → [formule] (* Implication *)
[formule] and [formule] (* Conjonction *)
[formule] or [formule] (* Disjonction *)
not [formule] (* Négation *)
[nom-predicat] ([terme] , ..., [terme]) (* Application de prédicat *)
forall x:  [type] , [formule] (* Quantification universelle *)
exists x:  [type] , [formule] (* Quantification existentielle *)
```

Le mot-clé **and** a une priorité plus forte que **or**.

Voici un fichier Why permettant de montrer que le 0 est le neutre de l'addition dans $\mathbb{Z}/3\mathbb{Z}$:

[4] L'utilisation d'assistant à la preuve comme Coq peut permettre de vérifier que ces théorèmes sont valides mais son utilisation dépasse le cadre de ce cours.

```
type z3z

logic zero : z3z
logic one : z3z
logic two : z3z
logic plus : z3z, z3z → z3z
logic mult : z3z, z3z → z3z

axiom injection : one ≠ two and two ≠ zero and zero ≠ one
axiom surjection : forall z: z3z. z = zero or z = one or z = two

axiom plus_definition :
    plus (zero, one) = one
and plus (zero, two) = two
and plus (zero, zero) = zero
and plus (one, one) = two
and plus (one, two) = zero
and plus (one, zero) = one
and plus (two, one) = zero
and plus (two, two) = one
and plus (two, zero) = two

logic is_neutral : z3z → prop
axiom is_neutral_definition :
  forall x: z3z. is_neutral (x) ↔ (forall z: z3z. plus (x, z) = z and plus (z, x) = z)

goal zero_is_neutral : is_neutral (zero)
goal one_is_not_neutral : not (is_neutral (one))
goal two_is_not_neutral : not (is_neutral (two))
```

Exercice 1 *(Correction syntaxique) Voici quelques entrées de l'utilisateur :*

- *type t = None | Some of int*
- *type nat*
- *logic f : nat → prop*
- *logic f : (nat → prop) → prop*
- *logic q : prop*
- *axiom : forall x: z2z, x = zero → forall x: z2z, x = one*
- *axiom associative_plus : forall x: nat, forall y: nat, plus (x, y) = plus (y, x)*
- *goal reflexive : true ↔ forall x: z2z, x = x*

❶ *Parmi ces entrées, lesquelles sont syntaxiquement correctes ?*

❷ *Soulignez les variables libres de ces formules.*

❸ *Liez chaque occurrence des variables liées à leurs lieurs respectifs.* □

2 Tautologies

Voici une liste de formules :

```
logic p, q, r, s : prop
goal t1 : p → p
goal t2 : (p or q) → (p or q)
goal t3 : p or q
goal t4 : p or (not p)
goal t5 : (p → q) and (r → s) → ((p or r) → (q and s))
goal t6 : (p → q) → ((p or r) → (q or r))
goal t7 : (not (p → q)) ↔ (p or not q)
goal t8 : (p ↔ q) ↔ (not p ↔ not q)
```

Exercice 2

❶ *Parmi ces formules lesquelles sont des tautologies ? Comment le vérifier ?*

❷ *Rappelez les lois de distributivité entre la conjonction et la disjonction.*

□

3 Entiers de Peano

On se place dans l'environnement logique suivant :

```
type nat
logic zero : nat
logic succ : nat → nat
logic pred : nat → nat
axiom pred_succ : forall x: nat. pred (succ (x)) = x
logic is_prime : nat → prop
logic is_even : nat → prop
logic is_odd : nat → prop
logic is_greater_than : nat, nat → prop
```

Le type nat est une représentation des entiers naturels. La constante zero représente 0. La fonction succ est la fonction $x \in \mathbb{N} \mapsto x + 1$. Le prédicat is_prime représente les entiers premiers. Le prédicat is_even représente les entiers pairs. Le prédicat is_odd représente les entiers impairs. La relation logique is_greater_than représente l'ordre strict standard sur les entiers naturels.

Exercice 3 *Pour chacun des énoncés informels suivants :*

- *Indiquez si on peut l'exprimez dans cet environnement logique et, le cas échéant, écrivez la formule correspondante.*
- *Cette propriété est-elle valide dans le modèle ?*
- *À votre avis, peut-on démontrer cette propriété dans cet environnement ?*
- *Si la formule est valide dans le modèle mais non démontrable, quelles propriétés sur les prédicats et les opérations seraient nécessaires pour la démontrer ?*

❶ *"2 moins 2 est égal à 0."*

❷ *"La fonction succ est injective."*

❸ *"Pour tout entier x, il existe un entier y tel que x = succ (y)."*

❹ *"Il existe un entier y tel que pour tout entier x, x = succ (y)."*

❺ *"Pour tout entier x, il existe un entier y tel que x = y + y."*

❻ *"Si un entier est pair alors son successeur est impair."*

❼ *"Tout entier premier qui n'est pas égal à 2 est impair."*

❽ *"Soit un entier est pair, soit il est impair."*

❾ *"Pour tout entier x, il existe un entier y tel que $x < y$. "*

❿ *"x est pair et il existe y tel que x = succ (succ (y)) est équivalent à x et y sont pairs."*

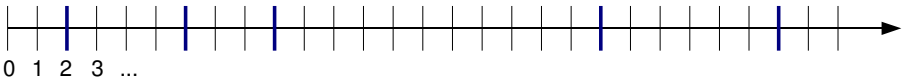
□

4 Séquence représentée par un prédicat

On se place de nouveau dans l'environnement logique suivant :

```
type nat
logic zero : nat
logic succ : nat → nat
logic pred : nat → nat
axiom pred_succ : forall x : nat. pred (succ (x)) = x
```

On se donne en outre un prédicat s de type nat → prop. Ce prédicat représente une séquence strictement croissante d'entiers naturels.



$$s = \{2, 6, 9, 20, 26, \ldots\}$$

Exercice 4

❶ *Formalisez la propriété « s est exactement la suite des entiers pairs ».*

❷ *Si on suppose la propriété précédente, peut-on prouver « $s(8)$ » ?*

❸ *On augmente l'environnement logique à l'aide des déclarations suivantes :*

```
logic plus : nat, nat → nat
axiom plus_zero_left : forall x: nat. plus (zero, x) = x
axiom plus_zero_right : forall x: nat. plus (x, zero) = x
axiom plus_succ_left : forall x: nat. forall y: nat. plus (succ (x), y) = succ (plus (x, y))
axiom plus_succ_right : forall x: nat. forall y: nat. plus (x, succ (y)) = succ (plus (x, y))
```

Si on suppose la propriété précédente, peut-on prouver « $\forall x : nat.s(x + x)$ » ?

❹ *Rappeler le principe de la démonstration par récurrence sur les entiers naturels. Peut-on exprimer ce principe dans notre logique ?*

❺ *En supposant une version spécialisée du principe d'induction, comment montrer la propriété : « $\forall x : nat.s(x+x)$ » ?*

❻ *Mêmes questions avec la propriété « les entiers pairs sont une sous-suite de s ».*

□

5 Séquence représentée syntaxiquement

Dans cette section, une séquence sur l'alphabet $\{a, b\}$ est représentée syntaxiquement à l'aide de deux symboles :
– nil représente la séquence vide ;
– si x $\in \{a, b\}$ et xs est une séquence alors cons (x, xs) représente la séquence dont la tête est x et la suite est xs.

Exercice 5

❶ *Définissez un environnement logique pour représenter les séquences.*

❷ *Déclarer un symbole negate qui représente la fonction qui remplace tous les a d'une séquence par des b. Quelles caractérisations de cette fonction pouvez-vous donner dans* Why *?*

❸ *Comment montrer que la fonction negate est idempotente ? Introduisez les théorèmes raisonnables nécessaires.*

❹ *Définissez un symbole concat représentant la fonction de concaténation entre deux séquences finies. Énoncez les théorèmes qui vous semblent raisonnables sur cette opération.*

❺ *Comment prouver « concat (negate (s1), negate (s2)) = negate (concat (s1, s2)) » ?*

□