Méthodes Formelles et Développement de logiciels Sûrs
Cours 2 : Why3, fondements logiques

Sandrine Blazy

istic Informatique Électronique

M1 informatique,
parcours génie logiciel et IR

16 janvier 2014

UNIVERSITÉ DE RENNES 1

Remerciements

Ce cours est inspiré du cours de Jean-Christophe Filliâtre.

Why

- Nombreux démonstrateurs pris en charge
  - solveurs SMT : Alt-Ergo, Z3, CVC3, Yices, etc.
  - prouveurs TPTP : Vampire, Eprover, SPASS, etc.
  - assistants de preuve : Coq, Isabelle, PVS, etc.
  - prouveurs dédidés : Gappa
- Extensible par l'utilisateur
- Efficace

  Le résultat des transformations est mémoïsé.

Quelle logique pour Why3 ?

La logique de Why3 permet d'utiliser une grande variété de démonstrateurs de théorèmes.

Compromis !

Logique polymorphe du premier ordre avec :

- types de données algébriques (mutuellement) récursifs,
- symboles de prédicats et de fonctions (mutuellement) récursifs,
- prédicats (mutuellement) inductifs,
- constructions let-in, match-with et if-then-else.

Déclarations

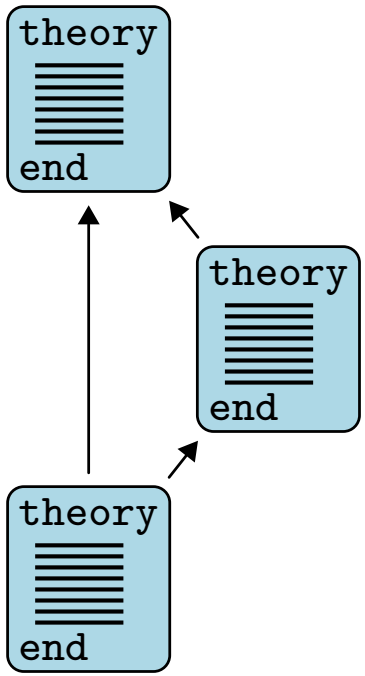
- déclaration de type
  - abstrait : `type t`
  - alias : `type t = list int`
  - algébrique : `type list 'a = Nil | Cons 'a (list 'a)`
- déclaration de fonction ou prédicat
  - non interprété : `function f int : int`
  - défini : `predicate mem (x: 'a) (l: list 'a) = ...`
- déclaration de prédicat inductif
  - `inductive sorted (l: list t) = ...`
- axiome, lemme, but
  - `goal G: forall x: int . x <= 0 -> x*x >= 0`

Théories

Les déclarations logiques sont organisées en théories.

Une théorie T_1 peut être :

- utilisée (`use`) dans une théorie T_2,
- clonée (`clone`) par une autre théorie.

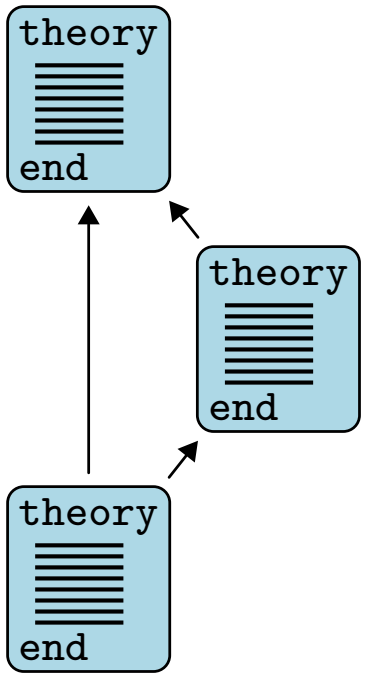


Théories

Les déclarations logiques sont organisées en théories.

Une théorie T_1 peut être :

- utilisée (`use`) dans une théorie T_2,
  - les symboles de T_1 sont partagés
  - les axiomes de T_1 restent des axiomes
  - les lemmes de T_1 deviennent des axiomes
  - les buts de T_1 sont ignorés
- clonée (`clone`) par une autre théorie.

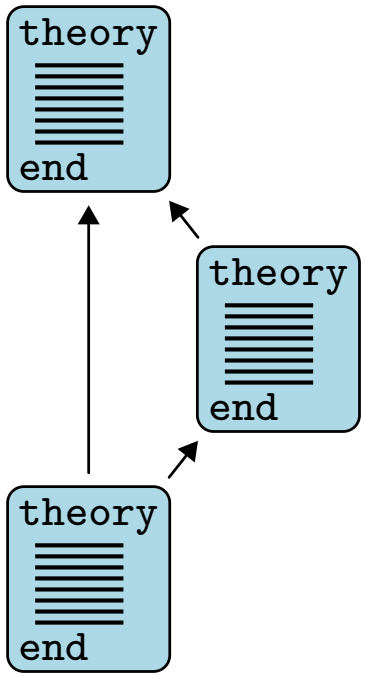


Théories

Les déclarations logiques sont organisées en théories.

Une théorie T_1 peut être :

- utilisée (`use`) dans une théorie T_2,
- clonée (`clone`) par une autre théorie.
  - les déclarations de T_1 sont copiées ou remplacées
  - les axiomes de T_1 restent des axiomes ou deviennent des buts
  - les lemmes de T_1 deviennent des axiomes
  - les buts de T_1 sont ignorés



Un premier exemple

```
theory HelloProof "My very first Why3 theory"

  goal G1 : true

  goal G2 : (true -> false) /\ (true \/ false)

  use import int.Int

  goal G3: forall x:int. x*x >= 0

end
```

Démo : `hello_proof.why`

Extrait de la théorie des entiers

```
theory Int
  constant zero : int = 0
  constant one  : int = 1

  predicate (< ) int int
  predicate (> ) (x y : int) = y < x
  predicate (<=) (x y : int) = x < y \/ x = y

  clone export algebra.OrderedUnitaryCommutRing with type t = int,
    constant zero = zero, constant one = one, predicate (<=) = (<=)
end

theory Abs
  use import Int
  function abs (x:int) : int = if x >= 0 then x else -x
  lemma Abs_le: forall x y:int. abs x <= y <-> -y <= x <= y
  lemma Abs_pos: forall x:int. abs x >= 0
end
```

Exemple : formules en logique propositionnelle

```
theory Prop
  predicate a
  predicate b
  predicate p
  predicate q
  goal G1 : ((a -> b) -> a) -> a
  goal G2 : a \/ not a
  goal G3 : not  (a \/ b) -> not a /\ not  b
  goal G4 : (not p -> not  q) -> (q -> p)
  goal G5 : p -> (not  p -> q)
  goal G6 : not  (p \/ q) <-> (not  p /\ not  q)
  goal G7 : (p -> q) <-> (not  p \/ q)
  goal G8 : (p -> q) -> (not  q -> not  p)
  goal G9 : (p /\ q) -> (q /\ p)
  goal G10 : not  p -> (p -> q)
  goal G11 : (p \/ q) -> (q \/ p)
end
```

Exemple : formules en logique du premier ordre

```
theory First
  type t
  constant c : t
  predicate a
  predicate p t
  predicate q t
  function f (t) : t

  goal P0 : (forall x:t. p(x)) -> p(c)
  goal P1 : (forall x:t. p(x)) -> (exists x:t. p(x))
  goal P2 : (a -> (forall x:t. q(x))) -> (forall x:t. a -> q(x))
  goal P3 : (forall x:t. p(x)) /\ (forall x:t. q(x)) ->
              (forall y:t. p(y) /\ q(y))
end
```

Démo : formules-logiques.why

Un premier exemple de formalisation : le club écossais

Il existe en Écosse un club très fermé qui obéit aux règles suivantes :

- Tout membre non écossais porte des chaussettes rouges.
- Tout membre porte un kilt ou ne porte pas de chaussettes rouges.
- Les membres mariés ne sortent pas le dimanche.
- Un membre sort le dimanche si et seulement s'il est écossais.
- Tout membre qui porte un kilt est écossais et marié.
- Tout membre écossais porte un kilt.

Montrer que ce club est si fermé qu'il ne peut accepter personne.

Un premier exemple de formalisation : le club écossais

Il existe en Écosse un club très fermé qui obéit aux règles suivantes :

- Tout membre non écossais porte des chaussettes rouges.
- Tout membre porte un kilt ou ne porte pas de chaussettes rouges.
- Les membres mariés ne sortent pas le dimanche.
- Un membre sort le dimanche si et seulement s'il est écossais.
- Tout membre qui porte un kilt est écossais et marié.
- Tout membre écossais porte un kilt.

Montrer que ce club est si fermé qu'il ne peut accepter personne.

Le club écossais en Why

```
theory ScottishClubProblem "the Scottish private club puzzle"
  predicate is_scottish
  predicate wears_red_socks
  predicate wears_kilt
  predicate is_married
  predicate goes_out_on_sunday

  axiom R1: not is_scottish -> wears_red_socks
  axiom R2: wears_kilt \/ not wears_red_socks
  axiom R3: is_married -> not goes_out_on_sunday
  axiom R4: goes_out_on_sunday <-> is_scottish
  axiom R5: wears_kilt -> is_scottish /\ is_married
  axiom R6: is_scottish -> wears_kilt

  goal ThereIsNobodyInTheClub: false

end
```

Démo : scottish-private-club.why

L'énigme d'Einstein (http://fr.wikipedia.org/wiki/Intégramme)

5 hommes de nationalités et de professions $\neq$ habitent avec leur animal préféré 5 maisons de couleurs $\neq$ où ils boivent leur boisson préférée.

- L'anglais habite la maison rouge.
- L'espagnol adore son chien.
- Le slovène boit du thé.
- La maison verte est située immédiatement à gauche de la blanche.
- L'islandais est ingénieur.
- La maison verte sent bon le café.
- Le sculpteur possède un âne.
- Le diplomate habite la maison jaune.
- Le norvégien habite la première maison à gauche.
- Le médecin habite la maison voisine de celle où demeure le propriétaire du renard.
- La maison du diplomate est voisine de celle où il y a un cheval.
- On boit du lait dans la maison du milieu.
- Le violoniste boit du jus d'orange.
- Le norvégien demeure à côté de la maison bleue.

L'énigme d'Einstein (http://fr.wikipedia.org/wiki/Intégramme)

5 hommes de nationalités et de professions $\neq$ habitent avec leur animal préféré 5 maisons de couleurs $\neq$ où ils boivent leur boisson préférée.

- L'anglais habite la maison rouge.
- L'espagnol adore son chien.
- Le slovène boit du thé.
- La maison verte est située immédiatement à gauche de la blanche.
- L'islandais est ingénieur.
- La maison verte sent bon le café.
- Le sculpteur possède un âne.
- Le diplomate habite la maison jaune.
- Le norvégien habite la première maison à gauche.
- Le médecin habite la maison voisine de celle où demeure le propriétaire du renard.
- La maison du diplomate est voisine de celle où il y a un cheval.
- On boit du lait dans la maison du milieu.
- Le violoniste boit du jus d'orange.
- Le norvégien demeure à côté de la maison bleue.

L'énigme en Why

```
theory Einstein "Einstein's problem"
(* Types *)
type maison = M1 | M2 | M3 | M4 | M5
type couleur = Bleu | Vert | Rouge | Blanc | Jaune
type personne= Espagnol | Anglais | Islandais | Norvegien | Slovene
type boisson = Orange | Cafe | Lait | The | Eau
type job= Ingenieur | Diplomate | Medecin | Violoniste | Sculpteur
type animal = Ane | Renard | Cheval | Chien | Zebre
```

Il reste à :

- représenter les positions des maisons,
- représenter la notion de voisin,
- associer à une personne sa nationalité, sa maison, sa boisson, son emploi ou son animal,
- associer à une maison sa couleur.

L'énigme en Why (suite)

Positions des maisons, notion de voisin

```
predicate agauchede (m1 m2 : maison) =
  match m1, m2 with
    | M1, M2
    | M2, M3
    | M3, M4
    | M4, M5 -> true
    | _      -> false
  end

predicate adroitede (m1 m2 : maison) = agauchede m2 m1

predicate voisin (m1 m2 : maison) =
        agauchede m1 m2 \/ adroitede m1 m2
```

L'énigme en Why

Associer à une personne ou une maison un attribut

Utilisation de bijections

```
theory Bijection
  type t
  type u

  function of t : u
  function to_ u : t

  axiom To_of : forall x : t. to_ (of x) = x
  axiom Of_to : forall y : u. of (to_ y) = y
end
```

L'énigme en Why

Définissons ensuite des bijections entre :

- maison et couleur
- maison et personne
- boisson et personne
- emploi et personne
- animal et personne

```
theory Einstein "Einstein's problem"
...
clone Bijection as Couleur with type t = maison, type u = couleur
clone Bijection as Owner with type t = maison, type u = personne

clone Bijection as Boisson with type t = personne, type u = boisson
clone Bijection as Job with type t = personne, type u = job
clone Bijection as Animal with type t = personne, type u = animal
...
```

L'énigme en Why

Formalisation de l'énigme

```
theory EinsteinClues "Clues"
  use import Einstein
  (* L'anglais habite la maison rouge. *)
  axiom Clue1: Couleur.of (Owner.to_ Anglais) = Rouge

  (* L'espagnol adore son chien. *)
  axiom Clue2: Animal.of Espagnol = Chien

  (* Le slovène boit du thé. *)
  axiom Clue3: Boisson.of Slovene = The

(* La maison verte est située immediatement à gauche de la blanche.
axiom Clue4: leftof (Couleur.to_ Vert) (Couleur.to_ Blanc)

  (* La maison verte sent bon le café. *)
  axiom Clue5: Boisson.of (Owner.of (Couleur.to_ Vert)) = Cafe

...
```

L'énigme en Why

Résolution de l'énigme

Qui boit de l'eau ? Qui élève le zèbre ?

```
theory Goals "Goals about Einstein's problem"
  use import Einstein
  use import EinsteinClues

  goal G1: Animal.to_ Zebre = Islandais
  goal Wrong: Animal.to_ Zebre = Norvegien
  goal G2: Boisson.to_ Eau = Norvegien

end
```

Démo : einstein.why

L'énigme en Why

D'autres propriétés intermédiaires peuvent être vérifiées.

```
theory Goals "Goals about Einstein's problem"
  lemma First_House_Not_White: Couleur.of M1 <> Blanc
  lemma Last_House_Not_Green: Couleur.of M5 <> Vert

  lemma Blue_not_Red: Bleu <> Rouge
  lemma Anglais_not_M2: Owner.to_ Anglais <> M2
  lemma Anglais_not_M1: Owner.to_ Anglais <> M1

  lemma Second_House_Blue: Couleur.of M2 = Bleu
  lemma Green_M4 : Couleur.of M4 = Vert
  lemma White_M5 : Couleur.of M5 = Blanc
  lemma Red_M3 : Couleur.of M3 = Rouge
  lemma Yellow_M1 : Couleur.of M1 = Jaune
    ...
end
```

Aident-elles à vérifier les buts ?

La suite avec Why3 ...

Polymorphisme

La logique du premier ordre n'est pas adaptée pour représenter des programmes impératifs annotés par des pré et post-conditions.

Ex. : comment raisonnner sur un tableau trié polymorphe ?

La logique de Why3 est une logique du premier ordre polymorphe

Logique du premier ordre polymorphe

Ajout de la notion de polymorphisme

Une sorte polymorphe est une paire $s : n$, avec s un symbole de sorte et $n \in \mathbb{N}$ son arité.
Une sorte d'arité 0 est dite monomorphe.

Soit S un ensemble de sortes polymorphes.

Types polymorphes :

$$\begin{array}{lll} \tau ::= \alpha & & \text{variable de type} \\ \quad | \; s(\tau, ...\tau) & & s \in S \end{array}$$

Un type τ sans variable de type est dit monomorphe.

Logique du premier ordre polymorphe : signature

$\Sigma = (S, P, F)$

- S ensemble de sortes polymorphes
- P ensemble de symboles de prédicats
  $\pi[\alpha_1, ...\alpha_k] : \tau_1 \times ... \times \tau_n$ avec $k, n \in \mathbb{N}$
- F ensemble de symboles de fonctions
  $\phi[\alpha_1, ...\alpha_k] : \tau_1 \times ... \times \tau_n \rightarrow \tau$ avec $k, n \in \mathbb{N}$

Un symbole est dit polymorphe (resp. monomorphe) si $k > 0$ (resp. $k = 0$).

Logique du premier ordre polymorphe : termes et formules

Les termes et les formules sont construits à partir de variables et d'instances de symboles.

Termes :

$$t ::= x_\tau \mid \phi[\tau,...\tau](t,...t) \qquad \phi \in F$$

Formules :

$$f ::= \pi[\tau,...\tau](t,...t) \qquad \pi \in P$$
$$\mid \texttt{true} \mid t = t \mid f \vee f \mid \neg f$$
$$\mid \exists x_\tau . f$$

Terme bien typé, formule bien formée

Sucre syntaxique : $\texttt{false}, f_1 \wedge f_2, f_1 \Rightarrow f_2, f_1 \Leftrightarrow f_2, \forall x_s . f$

Monomorphisation

Soit $\Sigma = (S, P, F)$ une signature polymorphe. On définit la signature $\texttt{mono}(\Sigma) = (S_1, P_1, F_1)$ ainsi :

- S_1 est l'ensemble des types monomorphes construits à partir de Σ
- P_1 est l'ensemble des instances monomorphes de P, c'est-à-dire tous les $\pi[\tau_1, ...\tau_k]$ avec τ_i monomorphe
- F_1 est l'ensemble des instances monomorphes de F, c'est-à-dire tous les $\phi[\tau_1, ...\tau_k]$ avec τ_i monomorphe

Monomorphisation

Soit f une formule. On note $\texttt{mono}(f)$ l'ensemble de toutes les instances monomorphes de f, c'est-à-dire de toutes les formules

$$f[\alpha_1 \leftarrow \tau_1, \ldots \alpha_n \leftarrow \tau_n]$$

où les α_i sont les variables de types de f, et les τ_i sont des types monomorphes.

Validité

Soit Γ un ensemble de formules closes (possiblement polymorphes) et f une formule close et monomorphe. On dit que f est un théorème de Γ si et seulement si

$$\mathtt{mono}(\Gamma) \models f$$

dans la logique simplement typée, pour la signature $\mathtt{mono}(\Sigma)$

Difficulté
L'ensemble $\mathtt{mono}(\Gamma)$ est rapidement infini.
On ne peut pas décider quel sous-ensemble de $\mathtt{mono}(\Gamma)$ sera nécessaire à la preuve de f.
Il faut donc traduire Γ dans la logique simplement typée par un nombre fini d'axiomes.

La théorie des listes polymorphes

Démo : demo_list.why

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

theory Length
  use import int.Int
  use import List

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

  lemma Length_nonnegative: forall l: list 'a. length l >= 0
  lemma Length_nil: forall l: list 'a. length l = 0 <-> l = Nil
end
```

La théorie des listes polymorphes (suite)

```
 theory Mem
  use export List
  predicate mem (x: 'a) (l: list 'a) = match l with
    | Nil      -> false
    | Cons y r -> x = y  mem x r
    end
end

theory Nth
  use export List
  use export option.Option    (* type option 'a = None | Some 'a *)
  use import int.Int

  function nth (n: int) (l: list 'a) : option 'a = match l with
    | Nil      -> None
    | Cons x r -> if n = 0 then Some x else nth (n - 1) r
  end
end
```

La théorie des listes polymorphes (suite)

```
theory Append
 function (++) (l1 l2: list 'a) : list 'a = match l1 with
   | Nil -> l2
   | Cons x1 r1 -> Cons x1 (r1 ++ l2)
 end
 lemma Append_assoc:
   forall l1 l2 l3: list 'a.
   l1 ++ (l2 ++ l3) = (l1 ++ l2) ++ l3
 lemma Append_l_nil:    forall l: list 'a. l ++ Nil = l
 lemma Append_length:
   forall l1 l2: list 'a. length (l1 ++ l2) = length l1 + length l2

 lemma mem_append:
   forall x: 'a, l1 l2: list 'a.
   mem x (l1 ++ l2) <-> mem x l1 \/ mem x l2
 lemma mem_decomp:
   forall x: 'a, l: list 'a.
   mem x l -> exists l1 l2: list 'a. l = l1 ++ Cons x l2
end
```

La théorie des listes polymorphes (suite)

```
theory Sorted
 type t
 predicate le t t

 inductive sorted (l: list t) =
   | Sorted_Nil:
       sorted Nil
   | Sorted_One:
       forall x: t. sorted (Cons x Nil)
   | Sorted_Two:
       forall x y: t, l: list t.
       le x y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))

 lemma sorted_mem:
   forall x: t, l: list t.
 (forall y: t. mem y l -> le x y) /\ sorted l <-> sorted (Cons x l)
end
```

Exemple d'utilisation de la théorie des listes

```
theory Meslistes "Mes premieres preuves sur des listes"
  use import int.Int
  use import list.List
  use import list.Length
  use import list.Mem
  use import list.Append

  goal G1 : length (Cons 1 (Cons 2 Nil)) > 0

  goal G2 :  length (Cons 1 (Cons 2 Nil)) >= 0

  goal G3 : not (length (Cons 1 (Cons 2 Nil))) = 0

  goal G4 : mem 2 (Cons 1 (Cons 2 Nil))

  goal G5 : forall x:'a, l: list 'a. length (Cons x l) > 0
end
```

Démo : preuves-listes.why

Bilan

- déclaration de type
  - abstrait : `type t`
  - alias : `type t = list int`
  - algébrique : `type list 'a = Nil | Cons 'a (list 'a)`
- déclaration de fonction ou prédicat
  - non interprété : `function f int : int`
  - défini : `predicate mem (x: 'a) (l: list 'a) = ...`
- déclaration de prédicat inductif
  - `inductive sorted (l: list t) = ...`
- axiome, lemme, but
  - `goal G: forall x: int . x <= 0 -> x*x >= 0`