Vérification de programmes avec lieurs

Martin Clochard
Équipe Toccata, LRI & Inria Saclay, Orsay, France

Journée annuelle du groupe LTP
LABRI, Université de Bordeaux
18 Novembre 2013

Objectifs

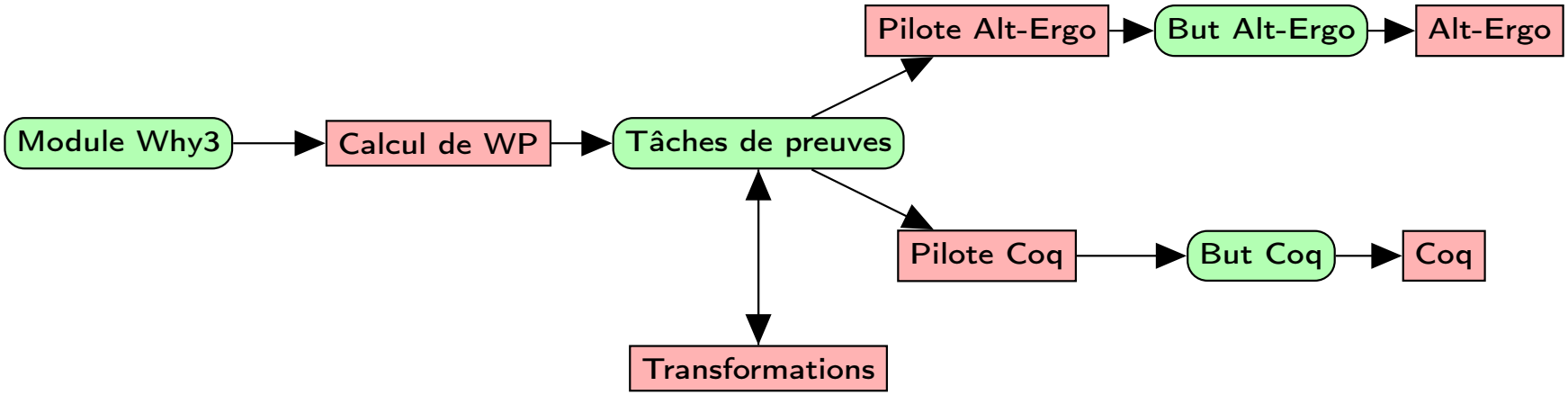
- Objectifs :
  - Trouver un moyen générique de spécifier et prouver des programmes manipulant des lieurs
  - Preuves : les plus automatiques possibles
- Solution proposée : générer automatiquement
  - Un modèle logique pour les types de données avec lieurs
  - L'implémentation des opérations de base
- Cadre de travail : la plateforme Why3

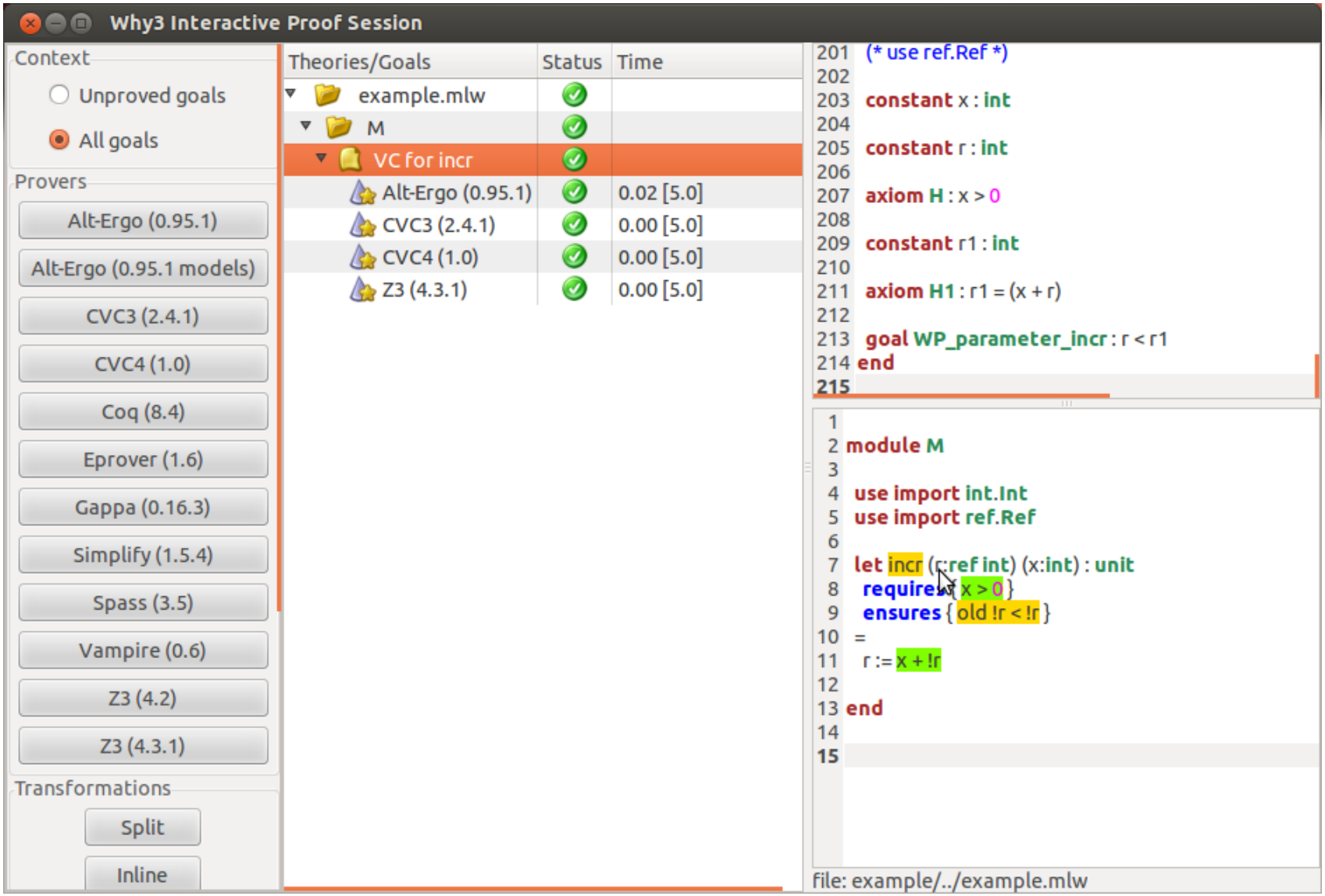
Why3

```
module M

  let incr (r:ref int) (x:int) : unit
    requires { x > 0 }
    ensures { old !r < !r }
  =
    r := x + !r

end
```




Why3 Interactive Proof Session
Context
Unproved goals
All goals
Provers
Alt-Ergo (0.95.1)
Alt-Ergo (0.95.1 models)
CVC3 (2.4.1)
CVC4 (1.0)
Coq (8.4)
Eprover (1.6)
Gappa (0.16.3)
Simplify (1.5.4)
Spass (3.5)
Vampire (0.6)
Z3 (4.2)
Z3 (4.3.1)
Transformations
Split
Inline
Theories/Goals
Status
Time
example.mlw
M
VC for incr
Alt-Ergo (0.95.1) 0.02 [5.0]
CVC3 (2.4.1) 0.00 [5.0]
CVC4 (1.0) 0.00 [5.0]
Z3 (4.3.1) 0.00 [5.0]
201 (* use ref.Ref *)
202
203 constant x : int
204
205 constant r : int
206
207 axiom H : x > 0
208
209 constant r1 : int
210
211 axiom H1 : r1 = (x + r)
212
213 goal WP_parameter_incr : r < r1
214 end
215
1
2 module M
3
4 use import int.Int
5 use import ref.Ref
6
7 let incr (r:ref int) (x:int) : unit
8 requires { x > 0 }
9 ensures { old !r < !r }
10 =
11 r := x + !r
12
13 end
14
15
file: example/../example.mlw

Notions de base sur les types de données avec lieurs

Type de donnée avec lieurs = type algébrique étendu par la liaison de variable

Exemple : $t \quad ::= \quad x \mid t\ t \mid \lambda x.t$

Notions de base sur les types de données avec lieurs

Type de donnée avec lieurs = type algébrique étendu par la liaison de variable
Exemple : $t \quad ::= \quad x \mid t\ t \mid \lambda x.t$

Notions caractéristiques :

- Construction, dont la fermeture d'un lieur (bind)
- Décomposition, dont l'ouverture d'un lieur (unbind)
- Substitution

Notions de base sur les types de données avec lieurs

Type de donnée avec lieurs = type algébrique étendu par la liaison de variable
Exemple : $t \quad ::= \quad x \mid t\ t \mid \lambda x.t$

Notions caractéristiques :

- Construction, dont la fermeture d'un lieur (bind)
- Décomposition, dont l'ouverture d'un lieur (unbind)
- Substitution
- Variables libres
- Test d'égalité (modulo α-équivalence)

État de l'art

Nombreux travaux :

- Programmation avec lieurs :
  - Système de type spécialisé (FreshML)
  - Génération de code (Cαml)
  - Interface de programmation adaptée (NaPa en Agda)
- Preuves formelles :
  - POPLmark challenge (2005), solutions dans divers assistants de preuves
  - Logique nominale

État de l'art

Nombreux travaux :

- Programmation avec lieurs :
  - Système de type spécialisé (FreshML)
  - Génération de code (Cαml)
  - Interface de programmation adaptée (NaPa en Agda)
- Preuves formelles :
  - POPLmark challenge (2005), solutions dans divers assistants de preuves
  - Logique nominale

Dans ce travail

On se préoccupe des deux aspects

Représentation Nommée

```
(* id : type des variables (string) *)

type term = | Var id
            | App term term
            | Lam id term
```

- ⊕ Représentation naturelle
- ⊖ Pose des problèmes de capture
- ⊖ α-équivalence complexe

Indices de de Bruijn

Les variables sont remplacées par leur distance au lieur correspondant
$\lambda x.(\lambda y.xya)xa$ devient $\lambda.(\lambda.1\ 0\ 44)\ 0\ 43$ (où $a = 42$)

```
type term = | Var int
            | App term term
            | Lam term
```

- ⊕ Pas de captures
- ⊕ α-équivalence triviale (représentation canonique)
- ⊖ Substitution : on doit décaler les indices

"Locally nameless"

- Variables liées : indices de de Bruijn
- Variables libres : nommées

```
type term = | BruijnVar int
            | FreeVar id
            | App term term
            | Lam term
```

- ⊕ Pas de captures
- ⊕ Substitution triviale (pas de décalage)
- ⊕ α-équivalence triviale (représentation canonique)
- ⊖ On doit fournir un nom frais à l'ouverture des lieurs

"Nested datatype"

```
type option 'a = | None
                 | Some 'a

type term 'a = | Var 'a
               | App (term 'a) (term 'a)
               | Lam (term (option 'a))
```

- Variables liées : indices de de Bruijn en notation unaire
- Variables libres : nommées sous un certain nombre d'applications du constructeur Some

"Nested datatype"

```
type option 'a = | None
                 | Some 'a

type term 'a = | Var 'a
               | App (term 'a) (term 'a)
               | Lam (term (option 'a))
```

- Variables liées : indices de de Bruijn en notation unaire
- Variables libres : nommées sous un certain nombre d'applications du constructeur `Some`

Lien avec les indices de de Bruijn :

- `None = 0`
- `Some x = x + 1`

"Nested datatype"

```
type option 'a = | None
                 | Some 'a

type term 'a = | Var 'a
               | App (term 'a) (term 'a)
               | Lam (term (option 'a))
```

- ⊕ Pas de captures
- ⊕ α-équivalence triviale (représentation canonique)
- ⊕ Les lieurs fournissent un nom frais canonique
- ⊕ Le typage offre certaines garanties
- ⊖ Inefficace en temps et en mémoire

Plan

Représentations choisies

- Modèle logique : une représentation sur laquelle il est facile de poser les définitions logiques et de raisonner dessus
  ➜ "nested datatype"
- Implémentation : une représentation sur laquelle les opérations sont efficaces
  ➜ "locally nameless"

Notions logiques

```
type option 'a = | None
                 | Some 'a

type term 'a = | Var 'a
               | App (term 'a) (term 'a)
               | Lam (term (option 'a))
```

- ouverture/fermeture : cas spéciaux de substitution
- α-équivalence : égalité structurelle
- Variables libres et substitution : voir slides suivants

Caractérisation logique des variables libres

```
predicate is_free_var (x:'a) (t:term 'a) = match t with
  | Var y -> x = y
  | App u v -> is_free_var x u \/ is_free_var x v
  | Lambda u -> is_free_var (Some x) u
end
```

Remarque

Le système de typage nous empêche d'oublier les décalages

Définition logique de la substitution

```
function subst (t:term 'a) (sigma:'a -> term 'b) : term 'b
  = match t with
    | Var x -> sigma x
    | App u v -> App (subst u sigma) (subst v sigma)
    | Lambda u -> let sigma' = (fun x:option 'a -> match x with
          | None -> Var None
          | Some x -> rename (sigma x) (fun x:'a -> Some x)
        end) in
      Lambda (subst u sigma')
  end
```

Principales propriétés prouvées pour le modèle logique

- Substitutions consécutives $\leftrightarrow$ substitution par composition

$$(t\sigma_1)\sigma_2 = t(\sigma_1 \circ \sigma_2)$$

- Substitution : ne dépend que des instances pour les variables libres

$$t\sigma_1 = t\sigma_2 \leftrightarrow (\forall x \in fv(t).\sigma_1(x) = \sigma_2(x))$$

- Caractérisation des variables libres d'une instanciation

$$x \in fv(t\sigma) \leftrightarrow (\exists y \in fv(t).x \in fv(\sigma(y)))$$

Principe de l'implémentation

Opérations structurelles du lambda-calcul :

- Implémentées en utilisant la représentation "locally nameless"
- Spécifiées via le modèle logique

Principe de l'implémentation

Opérations structurelles du lambda-calcul :

- Implémentées en utilisant la représentation "locally nameless"
- Spécifiées via le modèle logique

Il faut une interprétation d'une représentation dans l'autre

```
function model (t:nameless)
  (fr:id -> term 'a) (b:int -> term 'a) : term 'a =
  match t with
    | BruijnVar n -> b n
    | FreeVar x -> fr x
    | App u v -> Nested.App (model u fr b) (model v fr b)
    | Lambda u -> Nested.Lambda
      (model u (fun x:id -> rename (fr x) (fun x:'a -> Some x))
               (fun n:int -> if n = 0 then Var None
                             else rename (b (n-1)) (fun x:'a -> Some x))
end
```

Opérations d'ouverture/fermeture

Opérations d'ouverture/fermeture : par conversion entre indices de de Bruijn et variables libres

Opérations d'ouverture/fermeture

Opérations d'ouverture/fermeture : par conversion entre indices de de Bruijn et variables libres

```
function bind (t:nameless) (x:id) (i:int) : nameless =
  match t with
    | BruijnVar n -> t
    | FreeVar y -> if x = y then BruijnVar i else t
    | App u v -> App (bind u x i) (bind v x i)
    | Lambda u -> Lambda (bind u x (i+1))
  end

lemma model_bind : forall t:nameless,fr:id -> term 'a,b:int -> term 'a,
  x:id,i:int
  model (bind t x i) fr b = model t (fr[x <- b i]) b
```

Application : interpréteur

Cette approche a permis d'écrire un interpréteur certifié du lambda-calcul

```
val reduce (t:nameless) : nameless
  requires { nameless_ok t }
  ensures { nameless_ok result }
  ensures { beta_reduction (model t) (model result) }
  raises { Irreducible -> forall v:term id.
    not(beta_reduction (model t) v) }
```

Performance

L'efficacité de l'interpréteur a été testée par extraction vers OCaml

strategy	4^6	5^7	Fact 7	Fact 8	Ack 2 3	Ack 2 5	Ack 3 1
inner right	29 < 0.1s	40 1.2s	2173 4.1s	- >5m	2722 0.8s	10480 4.1s	13793 9.1s
inner left	29 < 0.1s	40 1.4s	2173 4.8s	- >5m	2722 0.6s	10480 2.7s	13793 8.0s
outer right	2732 1.9s	- >5m	1618 0.2s	3116 3.3s	2400 0.2s	25688 2.9s	27480 4.2s
outer left	2732 2.4s	- >5m	30901 48.7s	89681 2m44s	- >5m	- >5m	- 5m

Plan

Généralisation par outil externe

La généralisation ne peut pas être faite directement en Why3

Généralisation par outil externe

La généralisation ne peut pas être faite directement en Why3

Travail générique : effectué via un outil externe
Génère à partir d'une liste de types de données avec lieurs le code Why3 pour :

- les types correspondant aux deux représentations
- les notions logiques correspondants à chaque type avec lieurs
- l'implémentation spécifiée des opérations de base

Classe acceptée par l'outil

Classe acceptée : types algébriques mutuellement récursifs avec des lieurs dans les constructeurs
Les variables sont sortées, avec une sorte de variable par type de données

Classe acceptée par l'outil

Classe acceptée : types algébriques mutuellement récursifs avec des lieurs dans les constructeurs
Les variables sont sortées, avec une sorte de variable par type de données
Examples :

- Lambda-calcul
  $t ::= x \mid t\ t \mid \lambda x.t$
- Système $F_{<:}$ avec typage explicite
  $\tau ::= \alpha \mid \top \mid \forall \alpha <: \tau.\tau \mid \tau \to \tau$
  $t ::= \mid x$
  $\mid t\ t$
  $\mid \lambda(x : \tau).t$
  $\mid \Lambda(\alpha <: \tau).t$

Classe acceptée par l'outil

Classe acceptée : types algébriques mutuellement récursifs avec des lieurs dans les constructeurs
Les variables sont sortées, avec une sorte de variable par type de données
Examples :

- Lambda-calcul

  $t \quad ::= \quad Var \mid App\ t\ t \mid Lambda\ \sharp t\ t$

- Système $F_{<:}$ avec typage explicite

  $\tau \quad ::= \quad Var \mid Top \mid Forall\ \tau\ \sharp\tau\ \tau \mid Arrow\ \tau\ \tau$

  $t \quad ::= \quad \mid Var$
  $\mid App\ t\ t$
  $\mid Lambda\ \tau\ \sharp t\ t$
  $\mid TLambda\ \tau\ \sharp\tau\ t$

Schéma général

Les éléments générés par l'outil pour chaque type sont des généralisations de ceux présentés pour le lambda-calcul

Schéma général

Les éléments générés par l'outil pour chaque type sont des généralisations de ceux présentés pour le lambda-calcul
Example : encodage pour le système $F_{<:}$

```
type ftype 'a = | Var_type 'a
  | Top
  | Forall (ftype 'a) (ftype (option 'a))
  | Arrow (ftype 'a) (ftype 'a)

type fterm 'a 'b = | Var_term 'b
  | App (fterm 'a 'b) (fterm 'a 'b)
  | Lambda (ftype 'a 'b) (fterm 'a (option 'b))
  | TLambda (ftype 'a 'b) (fterm (option 'a) 'b)
```

Preuves automatiques

Les preuves ne peuvent être complètement automatisées

Raisonnement par induction : hors de portée des prouveurs automatiques

Preuves automatiques

Les preuves ne peuvent être complètement automatisées
Raisonnement par induction : hors de portée des prouveurs automatiques

La structure des preuves peut être partiellement fournie via des "lemma functions"

```
let rec lemma f (arguments) : unit
  requires { p }
  ensures { q }
  variant { v }
= ...
```

devient

```
lemma f : forall arguments. p -> q
```

Application : un prouveur tableau

L'approche générale présentée a permis :

- de générer des types de donnée pour la logique du premier ordre
- d'écrire une formalisation de sa sémantique
- d'implémenter un prouveur basé sur la méthode des tableaux et de certifier sa correction

```
(* May not terminate *)
val prove_unsat (l:formula_list) : unit
  requires { formula_list_ok l }
  ensures { forall rho:interpretation fsymb psymb varsymb.
    not(formula_list_conjunction l rho) }
```

Application : un prouveur tableau

Le prouveur a été extrait vers OCaml.
Exemple :

$$(\forall x.R\ x \vee R(f\ x)) \rightarrow \exists x.R\ x \wedge R\ (f^{2n}\ x)$$

n	3	4	5	6
time (sec.)	0.02	0.55	3.36	19.67
nb de noeuds génerés	502	9,506	42,898	197,244
(par sec.)	25,134	17,316	12,779	10,028

	lignes de code	obligations de preuves
Interpréteur lambda-calcul	1000	450
Outil de génération	6000	—
Prouveur (partie générée)	16000	3050
Prouveur (partie manuelle)	6000	4300

Conclusion

L'approche présentée permet de :

- Formaliser les définitions sur les types de données avec lieurs
- Écrire et certifier des programmes spécifiés via ces notions

Le tout avec un maximum d'automatisation
Objectif à plus long terme : développer de véritables outils certifiés (exemple : un vrai prouveur automatique)