« Calcul de plus faible précondition, revisité en Why3 » . . . revisité !

C. Marché A. Tafat . . . et beaucoup d'autres

Lab. Recherche en Informatique, Univ Paris-Sud, CNRS, INRIA Saclay.

JFLA 2014

11 janvier 2014

Plan

Plan

Un peu d'histoire

- (finale CdM+1an) Thèse Jean-Christophe Filliâtre
  - *Preuve de programmes impératifs*, en Coq

Un peu d'histoire

- (finale CdM+1an) Thèse Jean-Christophe Filliâtre
  - *Preuve de programmes impératifs*, en Coq
- 2001 : outil *Why* indépendant
  - produisant des buts en *Coq* ou en *PVS*

Un peu d'histoire

- (finale CdM+1an) Thèse Jean-Christophe Filliâtre
  - *Preuve de programmes impératifs*, en Coq
- 2001 : outil *Why* indépendant
  - produisant des buts en *Coq* ou en *PVS*
- 2002 : projet EU VerifiCard
  - JavaCard, Outil *Krakatoa* traduisant Java vers Why
  - ESC/Java, On se rend compte que l'on peut produire des buts en syntaxe *Simplify*

Un peu d'histoire

- (finale CdM+1an) Thèse Jean-Christophe Filliâtre
  - *Preuve de programmes impératifs*, en Coq
- 2001 : outil *Why* indépendant
  - produisant des buts en *Coq* ou en *PVS*
- 2002 : projet EU VerifiCard
  - JavaCard, Outil *Krakatoa* traduisant Java vers Why
  - ESC/Java, On se rend compte que l'on peut produire des buts en syntaxe *Simplify*
- 2004 : Outil Caduceus
  - Comme Krakatoa mais pour traiter le *cas du C*
  - ICFEM 2004

Un peu d'histoire

- (finale CdM+1an) Thèse Jean-Christophe Filliâtre
  - *Preuve de programmes impératifs*, en Coq
- 2001 : outil *Why* indépendant
  - produisant des buts en *Coq* ou en *PVS*
- 2002 : projet EU VerifiCard
  - JavaCard, Outil *Krakatoa* traduisant Java vers Why
  - ESC/Java, On se rend compte que l'on peut produire des buts en syntaxe *Simplify*
- 2004 : Outil Caduceus
  - Comme Krakatoa mais pour traiter le *cas du C*
  - ICFEM 2004
- 2005-2008, projet ANR CAT, l'aventure *Frama-C*
  - CEA, B. Monate
  - Caduceus remplacé par le greffon *Jessie*

Un peu d'histoire

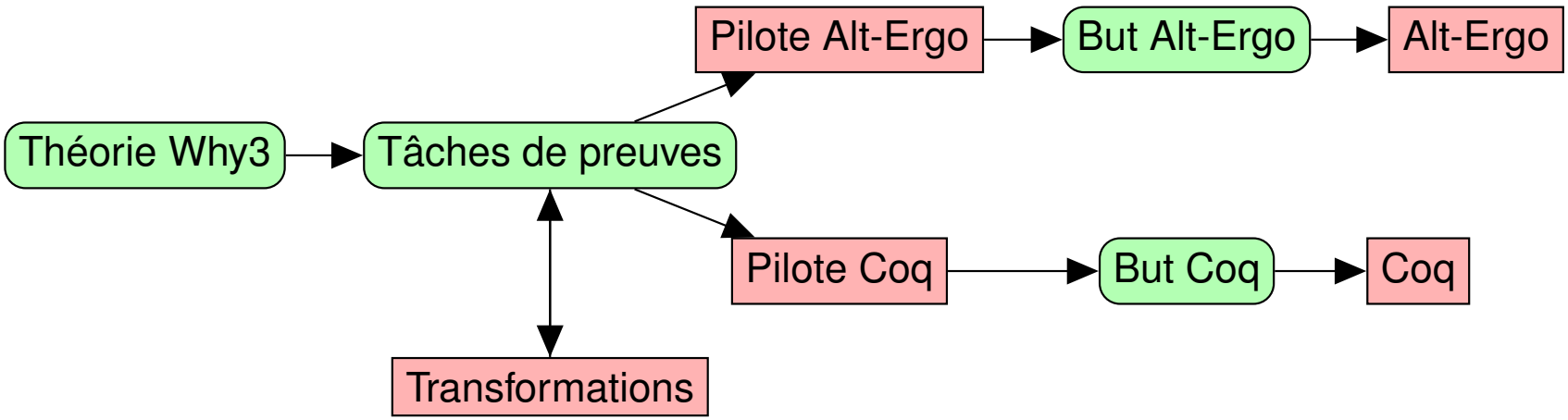
- (finale CdM+1an) Thèse Jean-Christophe Filliâtre
  - *Preuve de programmes impératifs*, en Coq
- 2001 : outil *Why* indépendant
  - produisant des buts en *Coq* ou en *PVS*
- 2002 : projet EU VerifiCard
  - JavaCard, Outil *Krakatoa* traduisant Java vers Why
  - ESC/Java, On se rend compte que l'on peut produire des buts en syntaxe *Simplify*
- 2004 : Outil Caduceus
  - Comme Krakatoa mais pour traiter le *cas du C*
  - ICFEM 2004
- 2005-2008, projet ANR CAT, l'aventure *Frama-C*
  - CEA, B. Monate
  - Caduceus remplacé par le greffon *Jessie*
- 2009-2012, projet ANR U3CAT
  - Support des *flottants* dans Jessie
  - Problématique de la certification des analyses

Naissance de Why3

- 2011, réimplantation complète de Why : Why3
- nombreuses nouveautés
  - Langage de spécification plus riche
  - Langage de programmation plus riche
  - Interface avec les prouveurs plus générique
    - tâches de preuve, transformations
    - drivers pour chaque prouveur
  - Sessions de preuve
  - API
  - etc.

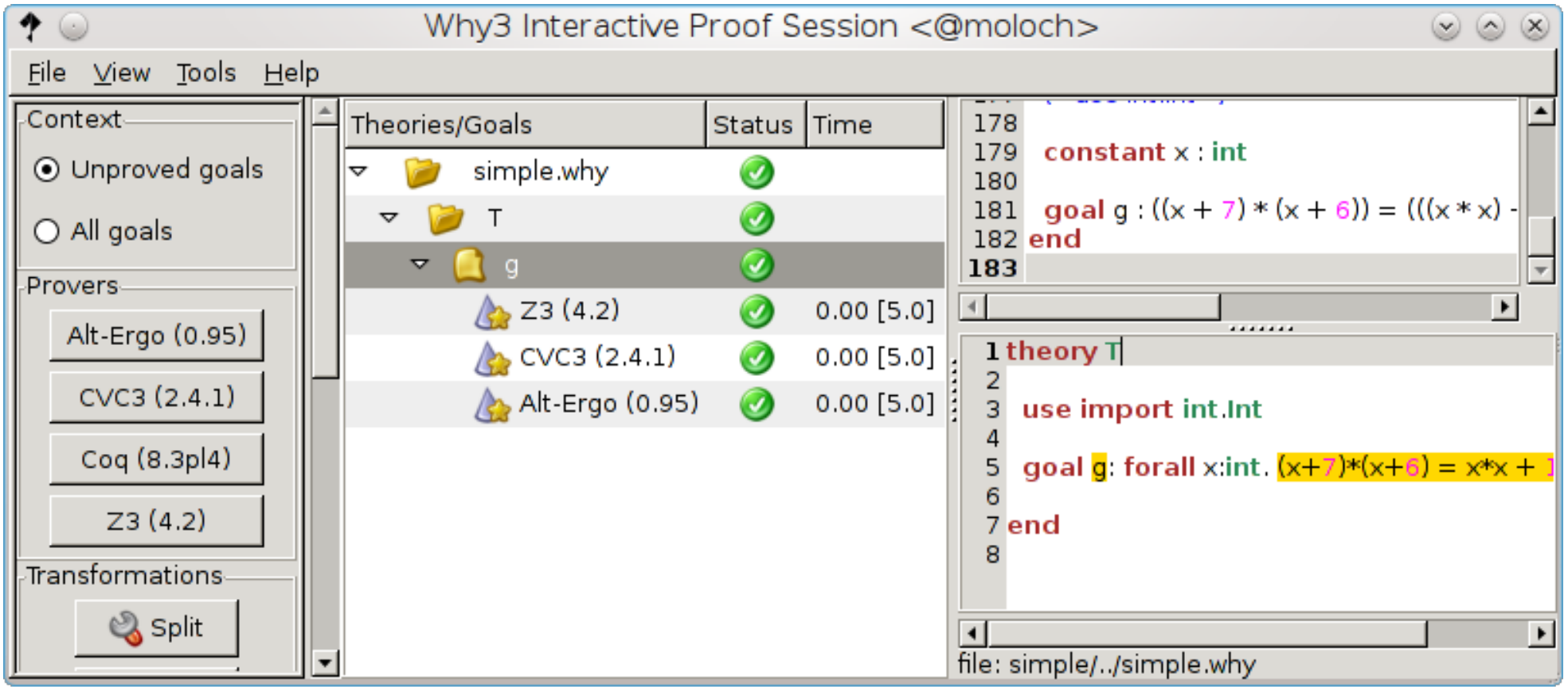
Why3 : Exemple

```
theory T

  use import int.Int

  goal g: forall x:int. (x+7)*(x+6) = x*x + 13*x + 42

end
```



Interface graphique Why3

```
> why3ide simple.why
```



Le côté obscur

- ICFEM 2004, Rustan Leino, *bug de correction* de Krakatoa
- U3CAT 2010, partenaires industriels qui tombent sur des bugs de correction de Jessie
- autres bugs de correction trouvés par nous-mêmes

Le côté obscur

- ICFEM 2004, Rustan Leino, *bug de correction* de Krakatoa
- U3CAT 2010, partenaires industriels qui tombent sur des bugs de correction de Jessie
- autres bugs de correction trouvés par nous-mêmes

Idée émergente (U3CAT)

- *prouver* la chaîne Frama-C/Jessie/Why
- thèse de Paolo Herms, co-encadrée par B. Monate

Le côté obscur

- ICFEM 2004, Rustan Leino, *bug de correction* de Krakatoa
- U3CAT 2010, partenaires industriels qui tombent sur des bugs de correction de Jessie
- autres bugs de correction trouvés par nous-mêmes

Idée émergente (U3CAT)

- *prouver* la chaîne Frama-C/Jessie/Why
- thèse de Paolo Herms, co-encadrée par B. Monate

Montagne à gravir...

plutôt

que

Thèse de Paolo Herms

- Prouver un générateur d'OP en Coq, en extraire un code OCaml pour en faire un greffon Frama-C
- Réutilisation *sémantique opérationnelle* de CompCert
- Formaliser la *sémantique de ACSL*
- Définir formellement *« un programme C respecte ses spécifications ACSL »*
- Prouver

Théorème de correction

si OP valides alors le programme respecte ses spécifications

Sémantique bloquante

Approche classique :

- Triplet de Hoare : $\{P\}\ S\ \{Q\}$
- Calcul de WP : $\mathit{statement} \rightarrow \mathit{formula} \rightarrow \mathit{formula}$
- Preuve de validité d'un triplet
  Si $P \Rightarrow WP(s, Q)$ alors $\{P\}\ S\ \{Q\}$ est valide

Sémantique bloquante

Approche classique :

- Triplet de Hoare : $\{P\}\ S\ \{Q\}$
- Calcul de WP : $\mathit{statement} \rightarrow \mathit{formula} \rightarrow \mathit{formula}$
- Preuve de validité d'un triplet
  Si $P \Rightarrow WP(s, Q)$ alors $\{P\}\ S\ \{Q\}$ est valide
- Difficulté : ACSL plus complexe que des triplets de Pre/post
  - assertions dans le code
  - invariants de boucle
  - variants
  - clauses `assigns`
  - `behavior`

Solution

Respect des annotations intégrée à la sémantique opérationnelle

Sémantique bloquante

(Ici à grands pas)

Version classique

$$\frac{}{\Sigma, \texttt{assert}\ P \rightsquigarrow \Sigma}$$

$$\frac{[\![c]\!]_\Sigma \qquad \Sigma, b \rightsquigarrow \Sigma_1 \qquad \Sigma_1, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ b \rightsquigarrow \Sigma_2}{\Sigma, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ b \rightsquigarrow \Sigma_2}$$

Sémantique bloquante

(Ici à grands pas)

Version bloquante

$$\frac{\Sigma \models P}{\Sigma, \texttt{assert}\ P \rightsquigarrow \Sigma}$$

$$\frac{\Sigma \models I \qquad [\![c]\!]_\Sigma \qquad \Sigma, b \rightsquigarrow \Sigma_1 \qquad \Sigma_1, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ b \rightsquigarrow \Sigma_2}{\Sigma, \texttt{while}\ c\ \texttt{invariant}\ I\ \texttt{do}\ b \rightsquigarrow \Sigma_2}$$

Sémantique bloquante

Par définition, un programme annoté *respecte ses spécifications* si son exécution ne bloque pas

Choix de la thèse de Paolo

- Difficulté : quelle sémantique prend-on : petits pas, grands pas ?
- Choix de Paolo : grand pas
- Plus simple au départ, mais ensuite
  - cas des comportements infinis
  - preuves par co-induction

Génèse du papier JFLA 2013

- Thèse d'Asma : preuve par raffinement dans les programmes à pointeurs
- Pour prouver la correction : sémantique bloquante
- Refaire ça en Coq : trop dur
- Mais en Why3 : pourquoi pas ?
  - Sur un langage impératif minimal pour commencer
  - Choix : sémantique à petit pas

Challenge

Le langage de spécification de Why3 est-il suffisant pour formaliser la sémantique d'un langage ?

Plan

Instructions

- Langage impératif minimal avec annotations
- Syntaxe abstraite formalisée grâce à un *type algébrique* de Why3

```
type stmt =
  | Sskip                    (* instruction no-op *)
  | Sassign mident term      (* affectation id := term *)
  | Sseq stmt stmt           (* sequence *)
  | Sif term stmt stmt       (* conditionnelle *)
  | Sassert fmla             (* assertion *)
  | Swhile term fmla stmt    (* while cond invariant body *)
```

Instructions

- Langage impératif minimal avec annotations
- Syntaxe abstraite formalisée grâce à un *type algébrique* de Why3

```
type stmt =
  | Sskip                    (* instruction no-op *)
  | Sassign mident term      (* affectation id := term *)
  | Sseq stmt stmt           (* sequence *)
  | Sif term stmt stmt       (* conditionnelle *)
  | Sassert fmla             (* assertion *)
  | Swhile term fmla stmt    (* while cond invariant body *)
```

Termes

- Why3 permet d'introduire des *types abstraits*

```
type mident (* identificateurs pour les variables mutables *)
type ident  (* identificateurs pour les variables logiques *)
```

```
type datatype = TYunit | TYint | TYbool
type value = Vvoid | Vint int | Vbool bool
type operator = Oplus | Ominus | Omult | Ole
type term =
 | Tvalue value                (* valeur *)
 | Tvar ident                  (* variable logique *)
 | Tderef mident               (* variable mutable *)
 | Tbin term operator term     (* operation binaire *)
```

Formules

```
type fmla =
  | Fterm term                        (* formule atomique *)
  | Fand fmla fmla                    (* conjonction *)
  | Fnot fmla                         (* negation *)
  | Fimplies fmla fmla                (* implication *)
  | Flet ident term fmla              (* let id = term in fmla *)
  | Fforall ident datatype fmla (* forall id: ty, fmla *)
```

Environnements

- Why3 possède une *librairie standard riche*
- Généricité grâce au *polymorphisme de type*

```
use map.Map as IdMap
type env = IdMap.map mident value
      (* associe des valeurs aux variables globales *)

use export list.List
type stack = list (ident, value)
      (* associe des valeurs aux variables logiques *)
```

Accès aux environnements

- Why3 permet de définir des fonctions *par récurrence structurelle*

```
function get_stack (i:ident) (pi:stack) : value =
  match pi with
  | Nil -> Vvoid
  | Cons (x,v) r -> if x=i then v else get_stack i r
  end
```

Accès aux environnements

- Why3 permet de définir des fonctions *par récurrence structurelle*

```
function get_stack (i:ident) (pi:stack) : value =
  match pi with
  | Nil -> Vvoid
  | Cons (x,v) r -> if x=i then v else get_stack i r
  end
```

En Why3, les fonctions logiques doivent être totales

Évaluation des termes

```
function eval_term (sigma:env)(pi:stack)(t:term): value =
  match t with
  | Tvalue v        -> v
  | Tvar id         -> get_stack id pi
  | Tderef id       -> IdMap.get sigma id
  | Tbin t1 op t2   -> eval_bin (eval_term sigma pi t1)
                         op (eval_term sigma pi t2)
  end
```

Évaluation des formules

```
predicate eval_fmla (sigma:env) (pi:stack) (f:fmla) =
  match f with
  | Fterm t          -> eval_term sigma pi t = Vbool True
  | Fand f1 f2       -> eval_fmla sigma pi f1 /\
                                  eval_fmla sigma pi f2
  | Fnot f           -> not (eval_fmla sigma pi f)
  | Flet x t f       -> eval_fmla sigma
                           (Cons (x,eval_term sigma pi t) pi) f
  | Fforall x TYint f-> forall n:int.
                           eval_fmla sigma (Cons (x,Vint n) pi) f
  ...
  end
```

Sémantique opérationnelle

- Why3 permet la *définition de prédicat par induction*

```
inductive one_step env stack stmt env stack stmt =
  | one_step_assert: forall sigma:env, pi:stack, f:fmla.
      eval_fmla sigma pi f ->        (* semantique bloquante *)
      one_step sigma pi (Sassert f) sigma pi Sskip

... (* 7 autres regles *)
```

Typage

- Nous définissons les environnements de typage

```
type type_stack = list (ident, datatype)
type type_env = IdMap.map mident datatype
```

Typage

- Nous définissons les environnements de typage

```
type type_stack = list (ident, datatype)
type type_env = IdMap.map mident datatype
```

- Règles de typage $\rightarrow$ prédicats inductifs en Why3

```
inductive type_term type_env type_stack term datatype =
...

inductive type_fmla type_env type_stack fmla =
...

inductive type_stmt type_env type_stack stmt =
...
```

Typage

- Nous définissons les environnements de typage

```
type type_stack = list (ident, datatype)
type type_env = IdMap.map mident datatype
```

- Règles de typage $\rightarrow$ prédicats inductifs en Why3

```
inductive type_term type_env type_stack term datatype =
...

inductive type_fmla type_env type_stack fmla =
...

inductive type_stmt type_env type_stack stmt =
...
```

```
predicate compatible_env (sigma:env) (sigmat:type_env)
                  (pi:stack) (pit: type_stack) = ...
```

Préservation du type par réduction

```
lemma type_preservation :
  forall s1 s2:stmt, sigma1 sigma2:env, pi1 pi2:stack,
                     sigmat:type_env, pit:type_stack.
   type_stmt sigmat pit s1 /\
   compatible_env sigma1 sigmat pi1 pit /\
   one_step sigma1 pi1 s1 sigma2 pi2 s2 ->
   type_stmt sigmat pit s2 /\
   compatible_env sigma2 sigmat pi2 pit
```

Préservation du type par réduction

Lemme

$\forall\ s\ s'\ \Sigma\ \Sigma'\ \Pi\ \Pi'\ \Sigma_t\ \Pi_t.$
$\mathit{type\_stmt}\ \Sigma_t\ \Pi_t\ s \wedge \mathit{compatible\_env}\ \Sigma\ \Sigma_t\ \Pi\ \Pi_t \wedge$
$\Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s' \Rightarrow \mathit{type\_stmt}\ \Sigma_t\ \Pi_t\ s' \wedge$
$\mathit{compatible\_env}\ \Sigma'\ \Sigma_t\ \Pi'\ \Pi_t$

Préservation du type par réduction

Lemme

$\forall\ s\ s'\ \Sigma\ \Sigma'\ \Pi\ \Pi'\ \Sigma_t\ \Pi_t.$
$\mathit{type\_stmt}\ \Sigma_t\ \Pi_t\ s \wedge \mathit{compatible\_env}\ \Sigma\ \Sigma_t\ \Pi\ \Pi_t \wedge$
$\Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s' \Rightarrow \mathit{type\_stmt}\ \Sigma_t\ \Pi_t\ s' \wedge$
$\mathit{compatible\_env}\ \Sigma'\ \Sigma_t\ \Pi'\ \Pi_t$

Preuve (Standard)

Par induction sur l'hypothèse $\Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s'$

Préservation du type par réduction

Lemme

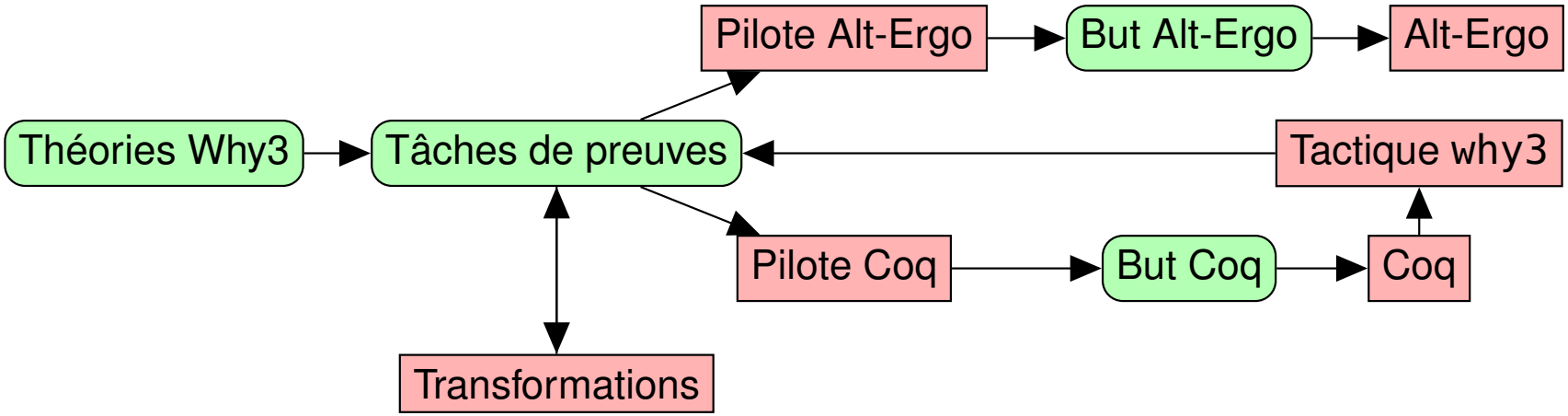
$\forall\ s\ s'\ \Sigma\ \Sigma'\ \Pi\ \Pi'\ \Sigma_t\ \Pi_t.$
$\mathit{type\_stmt}\ \Sigma_t\ \Pi_t\ s \wedge \mathit{compatible\_env}\ \Sigma\ \Sigma_t\ \Pi\ \Pi_t \wedge$
$\Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s' \Rightarrow \mathit{type\_stmt}\ \Sigma_t\ \Pi_t\ s' \wedge$
$\mathit{compatible\_env}\ \Sigma'\ \Sigma_t\ \Pi'\ \Pi_t$

Preuve (Standard)

Par induction sur l'hypothèse $\Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s'$

Impossible avec les prouveurs automatiques

Tactique why3 de Coq



Preuve

```
Require Import Why3.
Ltac ae := why3 "alt-ergo" timelimit 5

intros s1 s2 sigma1 sigma2 pi1 pi2 sigmat pit (h1,(h2,h3)).
induction h3; try ae. (* 7 sous-buts prouves sur 8 *)
inversion h1; subst; ae.
```

Substitution

- Substituer une variable mutable par une variable logique

```
function msubst_term (t:term) (x:mident) (v:ident) : term =
  match t with
  | Tvalue _ | Tvar _ -> t
  | Tderef y          -> if x = y then Tvar v else t
  | Tbin t1 op t2     -> Tbin (msubst_term t1 x v) op
                              (msubst_term t2 x v)
  end

function msubst (f:fmla) (x:mident) (v:ident) : fmla =
  match f with
  | Fterm e       -> Fterm (msubst_term e x v)
  | Fand f1 f2    -> Fand (msubst f1 x v) (msubst f2 x v)
  | Flet y t f    -> Flet y (msubst_term t x v) (msubst f x v)
  ...
```

Fraîcheur

Affirmer qu'une variable est fraîche dans un terme ou une formule

```
predicate fresh_in_term (id:ident) (t:term) =
  match t with
  | Tvalue _ | Tderef _ -> true
  | Tvar i        -> id <> i
  | Tbin t1 _ t2 -> fresh_in_term id t1 /\ fresh_in_term id t2
  end

predicate fresh_in_fmla (id:ident) (f:fmla) =
  match f with
  | Fterm e     -> fresh_in_term id e
  | Fand f1 f2  -> fresh_in_fmla id f1 /\ fresh_in_fmla id f2
  | Flet y t f  -> id <> y /\ fresh_in_term id t /\
                                fresh_in_fmla id f
  ...
```

Propriétés sur `subst` et `fresh`

Lemme 1

$[\![t[x \leftarrow v]]\!]_{\Sigma,\, \Pi} = [\![t]\!]_{\Sigma[x \leftarrow \Pi(v)],\, \Pi}$
$[\![f[x \leftarrow v]]\!]_{\Sigma,\, \Pi} \Leftrightarrow [\![f]\!]_{\Sigma[x \leftarrow \Pi(v)],\, \Pi}$

Propriétés sur `subst` et `fresh`

Lemme 1

$[\![t[x \leftarrow v]]\!]_{\Sigma,\, \Pi} = [\![t]\!]_{\Sigma[x \leftarrow \Pi(v)],\, \Pi}$
$[\![f[x \leftarrow v]]\!]_{\Sigma,\, \Pi} \Leftrightarrow [\![f]\!]_{\Sigma[x \leftarrow \Pi(v)],\, \Pi}$

Preuve

Transformation "induction" de Why3

Propriétés sur `subst` et `fresh`

Lemme 1

$[\![t[x \leftarrow v]]\!]_{\Sigma,\, \Pi} = [\![t]\!]_{\Sigma[x \leftarrow \Pi(v)],\, \Pi}$
$[\![f[x \leftarrow v]]\!]_{\Sigma,\, \Pi} \Leftrightarrow [\![f]\!]_{\Sigma[x \leftarrow \Pi(v)],\, \Pi}$

Preuves

- Termes : Les 4 sous-buts prouvés automatiquement
- Formules :
  - 10 sous-buts prouvés automatiquement
  - 2 sous-buts (Fforall et Flet) prouvés en Coq

```
destruct f; auto.
simpl; ae.
```

Propriétés sur `subst` et `fresh`

Lemme 2

$[\![t]\!]_{\Sigma,\ \Pi_1\cdot(id_1,v_1)\cdot(id_2,v_2)\cdot\Pi_2} = [\![t]\!]_{\Sigma,\ \Pi_1\cdot(id_2,v_2)\cdot(id_1,v_1)\cdot\Pi_2}$ $\quad id_1 \neq id_2$

$[\![f]\!]_{\Sigma,\ \Pi_1\cdot(id_1,v_1)\cdot(id_2,v_2)\cdot\Pi_2} \Leftrightarrow [\![f]\!]_{\Sigma,\ \Pi_1\cdot(id_2,v_2)\cdot(id_1,v_1)\cdot\Pi_2}$

Lemme 3

$[\![t]\!]_{\Sigma,\ (id,v)\cdot\Pi} = [\![t]\!]_{\Sigma,\ \Pi}$ si id est frais.

$[\![f]\!]_{\Sigma,\ (id,v)\cdot\Pi} \Leftrightarrow [\![f]\!]_{\Sigma,\ \Pi}$ si id est frais.

Preuves avec la transformation "induction" de Why3

- 32 sous-buts
  - 30 prouvés automatiquement
  - 2 restants : 4 + 3 lignes de Coq

Définition du calcul de WP

```
function wp (s:stmt) (q:fmla) : fmla =
  match s with
  | Sskip                  -> q
  | Sassert f              -> Fand f (Fimplies f q)
  | Sseq s1 s2             -> wp s1 (wp s2 q)
  | Sassign x t            -> let id = fresh_from q in
                                 Flet id t (msubst q x id)
  | Sif t s1 s2            ->
       Fand (Fimplies (Fterm t) (wp s1 q))
            (Fimplies (Fnot (Fterm t)) (wp s2 q))
  | Swhile cond inv body ->
    Fand inv (abstract_effects body
     (Fand (Fimplies (Fand (Fterm cond) inv)(wp body inv))
       (Fimplies (Fand (Fnot (Fterm cond)) inv) q)))
  end
```

Propriété de correction

Rappel : Sémantique bloquante

Par définition, un programme annoté *respecte ses spécifications* si son exécution ne bloque pas

Théorème de correction

$\forall\ \Sigma\ \Pi\ s\ Q.$
Si $[\![\mathrm{WP}(s,\ Q)]\!]_{\Sigma,\Pi}$ alors

- Soit $\Sigma,\ \Pi,\ s \rightsquigarrow^*\ \Sigma',\ \Pi',\ \mathit{Sskip}$ et $[\![Q]\!]_{\Sigma',\Pi'}$
- Soit $\Sigma,\ \Pi,\ s$ se réduit indéfiniment

Préservation par réduction

Lemme de préservation

$$\forall\ \Sigma\ \Sigma'\ \Pi\ \Pi'\ s\ s'.\ \ \Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s' \Rightarrow$$
$$\forall\ Q.\ [\![\mathrm{WP}(s,\ Q)]\!]_{\Sigma,\Pi} \Rightarrow\ [\![\mathrm{WP}(s',\ Q)]\!]_{\Sigma',\Pi'}$$

Préservation par réduction

Lemme de préservation

$$\forall\ \Sigma\ \Sigma'\ \Pi\ \Pi'\ s\ s'.\ \ \Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s' \Rightarrow$$
$$\forall\ Q.\ [\![\mathrm{WP}(s,\ Q)]\!]_{\Sigma,\Pi} \Rightarrow\ [\![\mathrm{WP}(s',\ Q)]\!]_{\Sigma',\Pi'}$$

Preuve par induction sur l'hypothèse $\Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s'$

```
intros sigma sigma' pi pi' s s' h1.
induction h1; try (simpl; intro; ae).
simpl; intros q (_ & h).
generalize h; intro h'.
apply abstract_effects_specialize in h'; simpl in h'; ae.
simpl; intros q (_ & h).
apply abstract_effects_specialize in h; simpl in h; ae.
```

Progrès

$$\mathit{reducible}\ \Sigma\ \Pi\ s\ :=\ \exists \Sigma'\ \Pi'\ s'.\ \ \Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s'$$

Lemme de progrès

$\forall\ s\ \Sigma\ \Pi\ \Sigma_t\ \Pi_t\ Q.$
$\mathit{compatible\_env}\ \Sigma\ \Sigma_t\ \Pi\ \Pi_t\ \wedge\ \mathit{type\_stmt}\ \Sigma_t\ \Pi_t\ s\quad \wedge$
$[\![\mathrm{WP}(s,\ Q)]\!]_{\Sigma,\Pi} \wedge\ s \neq \mathit{Sskip}\ \Rightarrow\ \mathit{reducible}\ \Sigma\ \Pi\ s$

Progrès

$$\mathit{reducible}\ \Sigma\ \Pi\ s\ :=\ \exists \Sigma'\ \Pi'\ s'.\ \ \Sigma,\ \Pi,\ s \rightsquigarrow \Sigma',\ \Pi',\ s'$$

Lemme de progrès

$\forall\ s\ \Sigma\ \Pi\ \Sigma_t\ \Pi_t\ Q.$
$\mathit{compatible\_env}\ \Sigma\ \Sigma_t\ \Pi\ \Pi_t\ \wedge\ \mathit{type\_stmt}\ \Sigma_t\ \Pi_t\ s\quad \wedge$
$[\![\mathrm{WP}(s,\ Q)]\!]_{\Sigma,\Pi} \wedge\ s \neq \mathit{Sskip}\ \Rightarrow\ \mathit{reducible}\ \Sigma\ \Pi\ s$

Preuve avec la transformation "induction" de Why3

- 1 sous-but prouvé automatiquement : Sskip,
- 5 sous-buts prouvés en Coq (46 lignes)

Correction

Théorème de correction (formulation équivalente)

$$\begin{array}{l}\forall\ n\ \Sigma\ \Sigma'\ \Pi\ \Pi'\ s\ s'\ \Sigma_t\ \Pi_t\ Q. \\ \quad \mathit{compatible\_env}\ \Sigma\ \Sigma_t\ \Pi\ \Pi_t \quad \wedge \quad \mathit{type\_stmt}\ \Sigma_t\ \Pi_t t\ s \quad \wedge \\ \quad \Sigma,\ \Pi,\ s \rightsquigarrow^n\ \Sigma',\ \Pi',\ s' \quad \wedge \quad \mathit{not}(\mathit{reducible}\ \Sigma'\ \Pi'\ s') \quad \wedge \\ \quad [\![\mathrm{WP}(s,\ Q)]\!]_{\Sigma,\Pi} \quad \Rightarrow \quad s' = \mathit{Sskip} \wedge [\![Q]\!]_{\Sigma',\Pi'}\end{array}$$

Preuve

En Coq, par induction sur n (22 lignes)

Conclusions JFLA 2013

- Utilisation de la richesse du langage Why3 et des outils
  - Types algébriques
  - Définition de fonctions par récursion structurelle
  - Définition de prédicats par induction
  - Transformation d'induction structurelle de Why3
  - Tactique `why3` dans Coq
- Degrés satisfaisant d'automatisation :
  - 390 lignes de code
  - 142 lignes de preuve

Perspectives JFLA 2013

- Extension de l'étude
  - Un langage avec fonctions et appels de fonction
- Développer du code correct par construction
  - Alternative à une approche tout en Coq
    - Plus d'automatisation
    - Programmes avec effets de bord
  - Mécanisme d'extraction de Why3 vers OCaml en cours de développement

Plan

Que s'est-il passé après ?

- Étendre aux fonctions et appels de fonctions : pas fait. . .
- Asma a soutenu sa thèse
  - preuve sur le papier, non formalisée, de correction de son calcul de WP
- Aller plus loin : est-ce vraiment envisageable de développer un code d'un programme complet en Why3, et d'en faire un exécutable (via extraction vers Caml)

Un prouveur développé en Why3

- Stage de M2 de Martin Clochard, co-encadré par Andrei Paskevich et C. Marché.
- Présenté à la journée LTP, future présentation à PLPV 2014

- Problème des *lieurs*
  - *outil générant du code Why3* à partir d'une déclaration d'un *type algébrique avec lieurs* (*locally nameless* pour le code, *nested type* pour la partie logique)
  - génère aussi des lemmes généraux sur les variables fraîches, les substitutions, etc.
- Études de cas
  - interpréteur du lambda-calcul, stratégies diverses
  - un *prouveur en logique du premier ordre, méthode des tableaux*

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
```

Lemma functions : exemple

```
type nat = O | S nat

function add (x y:nat) : nat =
  match y with
  | O -> x
  | S z -> S (add x z)
  end

lemma add_0_n: forall n:nat. add O n = n
```

Lemma functions : exemple

```
type nat = O | S nat

function add (x y:nat) : nat =
  match y with
  | O -> x
  | S z -> S (add x z)
  end

(* lemma add_0_n: forall n:nat. add O n = n *)
let rec lemma add_0_n (n:nat) : unit
  ensures { add O n = n }
  variant { n }
 = match n with
   | O -> ()
   | S m -> add_0_n m
   end
```

Application : un prouveur, méthode des tableaux

L'approche a permis :

- de générer des types de données adéquats 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*

```
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) }
```

Évaluation de la performance du prouveur

Le prouveur a été *extrait vers OCaml*.

Famille d'exemples :

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

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

Quelques chiffres

Outil de génération : $\simeq$ 6.000 loc OCaml

Prouveur :

	loc Why3	obligations de preuves
partie générée	$\simeq$ 16.000	3.051
partie manuelle	$\simeq$ 6.000	4.303

Tout est prouvé avec Alt-Ergo, CVC3, CVC4, Eprover, Spass, et Z3 (temps limite max : 20s)

Conclusions et Perspectives

- Why3 commence a être suffisamment au point pour réellement développer du code certifié
- Perspectives
  - Améliorer l'extraction
  - Ajout d'ordre supérieur dans la logique/dans les programmes
  - Raffinement de modules
  - Travailler avec des entiers machines
  - ...