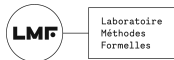


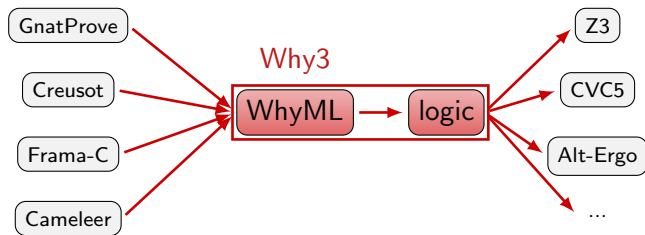
Faire chauffer Why3 avec de l'induction

Jean-Christophe Filliâtre, Andrei Paskevich, Henri Soudubray

Journées Francophones des Langages Applicatifs
30 janvier 2025



Un langage intermédiaire pour la vérification



- ① La logique de Why3
- ② Preuves par induction
- ③ Une contribution : `match inductive`
- ④ Une application

Une logique de **compromis** :

- expressivité pour les utilisateurs
- simplicité pour les solveurs SMT (et aussi pour les utilisateurs)

On obtient une extension de la logique du premier ordre

One logic to use them all [Filliâtre, CADE 13]

A Formalization of Core Why3 in Coq [Cohen, Johnson-Freyd, POPL 24]

- Types polymorphes • Types algébriques

```
type list 'a = Nil | Cons 'a (list 'a)
```

```
constant alist: list int =  
  Cons 34 (Cons 55 (Cons 89 Nil))
```

- Types polymorphes • Types algébriques • Filtrage
- Fonctions récursives et prédicats (avec vérification de terminaison)

```
function length (l: list 'a) : int
= match l with
| Nil          -> 0
| Cons _ t    -> 1 + length t
end
```

```
lemma L:
  forall l: list 'a. length l >= 0
```

- Types polymorphes • Types algébriques • Filtrage
- Fonctions récursives et prédicats (avec vérification de terminaison)
- Prédicats inductifs ($\neq$ types algébriques!)

```
inductive nim_step int int =  
| RemoveOne: forall n. n >= 0 -> nim_step (n+1) n  
| RemoveTwo: forall n. n >= 0 -> nim_step (n+2) n  
  
inductive nim int int =  
| Step: forall x y. nim_step x y -> nim x y  
| Path: forall x y z. nim x y -> nim y z -> nim x z
```

```
lemma L:  
  forall x y. nim x y -> 0 <= y < x
```

Pour appeler les solveurs SMT, la logique de Why3 est traduite vers la logique du premier ordre

Ceci implique l'**encodage**

- du polymorphisme
- des types algébriques
- du filtrage

avec une perte potentielle de la complétude

Un type algébrique comme

```
type list 'a = Nil | Cons 'a (list 'a)
```

est envoyé aux solveurs SMT comme

- un type non interprété list
- une constante Nil et une fonction Cons
- des principes d'injectivité et d'inversion
- mais pas de principe d'induction

De manière similaire, un prédicat inductif comme

```
inductive nim int int =  
| Step: forall x y. nim_step x y -> nim x y  
| Path: forall x y z. nim x y -> nim y z -> nim x z
```

est envoyé aux solveurs SMT comme

- un prédicat non interprété `nim`
- deux axiomes `Step` et `Path`
- un principe d'inversion
- mais pas de principe d'induction

2

Preuves par induction

```
lemma L:  
  forall l: list 'a. length l >= 0
```

ne peut pas être prouvé par les solveurs SMT

On utilise des **fonctions lemmes**, c'est-à-dire des fonctions de programme fantômes dont le seul but est de construire des preuves

Folklore, bien illustré dans **Program Proofs** [Leino, 2023]

```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
end
```

On utilise des **fonctions lemmes**, c'est-à-dire des fonctions de programme fantômes dont le seul but est de construire des preuves

Folklore, bien illustré dans **Program Proofs** [Leino, 2023]

fonction de programme



```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
end
```

On utilise des **fonctions lemmes**, c'est-à-dire des fonctions de programme fantômes dont le seul but est de construire des preuves

Folklore, bien illustré dans **Program Proofs** [Leino, 2023]

fonction de programme *fantôme*



```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
end
```

On utilise des **fonctions lemmes**, c'est-à-dire des fonctions de programme fantômes dont le seul but est de construire des preuves

Folklore, bien illustré dans **Program Proofs** [Leino, 2023]

fonction de programme *fantôme*

ne renvoie rien



```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
end
```

On utilise des **fonctions lemmes**, c'est-à-dire des fonctions de programme fantômes dont le seul but est de construire des preuves

Folklore, bien illustré dans **Program Proofs** [Leino, 2023]

fonction de programme *fantôme*

ne renvoie rien

```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 } ← postcondition
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
end
```

On utilise des **fonctions lemmes**, c'est-à-dire des fonctions de programme fantômes dont le seul but est de construire des preuves

Folklore, bien illustré dans **Program Proofs** [Leino, 2023]

fonction de programme *fantôme*

ne renvoie rien

```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
end
```

On utilise des **fonctions lemmes**, c'est-à-dire des fonctions de programme fantômes dont le seul but est de construire des preuves
Folklore, bien illustré dans **Program Proofs** [Leino, 2023]

fonction de programme *fantôme*

ne renvoie rien

```
let rec lemma length_nonneg (l: list 'a) : unit
  ensures { length l >= 0 }
  variant { l }
= match l with
  | Nil      -> ()
  | Cons _ t -> length_nonneg t
end
```

- ❶ La fonction lemme est d'abord vérifiée, comme toute fonction de programme
- ❷ Elle génère ensuite une hypothèse dans le contexte

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

- ❸ Elle peut être appelée comme toute fonction fantôme

```
...  
let n = length mylist in  
length_nonneg mylist;  apporte n >= 0  
...
```

Lorsqu'un lemme requiert une induction sur un **prédicat inductif**, e.g.,

```
lemma L:  
  forall x y. nim x y -> 0 <= y < x
```

on souhaiterait le prouver avec une fonction lemme

3

Une contribution : match inductive

- Rester proche des techniques habituelles dans Why3
 - Fonctions lemmes
 - Induction guidée par l'utilisateur via des appels récursifs explicites
- Ne pas polluer le contexte logique
 - Si possible, n'introduire aucun type/fonction/axiome supplémentaire

- Une nouvelle construction `match inductive` permettant un filtrage sur une preuve inductive
- Un nouveau type de `variant` pour vérifier la validité des appels récursifs

```

let rec lemma nim_bounds (x z: int)
  requires H { nim x z }
  ensures { 0 <= z < x }
= match inductive H with
  | Step _ _ _      -> ()
  | Path _ y _ _ _ -> nim_bounds x y; nim_bounds y z
end

```

```

inductive nim int int =
| Step: forall x y. nim_step x y -> nim x y
| Path: forall x y z. nim x y -> nim y z -> nim x z

```

hypothèse nommée

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z }
  ensures { 0 <= z < x }
= match inductive H with
  | Step _ _ _ -> ()
  | Path _ y _ _ -> nim_bounds x y; nim_bounds y z
end
```

filtrage sur H

```
inductive nim int int =
| Step: forall x y. nim_step x y -> nim x y
| Path: forall x y z. nim x y -> nim y z -> nim x z
```

```
let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures { 0 <= z < x }
= let ghost Step (x1 y1: int)
  requires { nim_step x1 y1 }
  requires { x = x1 /\ z = y1 }
  ensures { 0 <= z < x } = ()
in
let ghost Path (x1 y z1)
  requires { nim x1 y }
  requires { nim y z1 }
  requires { x = x1 /\ z = z1 }
  ensures { 0 <= z < x } =
  nim_bounds x y; nim_bounds y z
in assume { false }; absurd
```

```

let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures { 0 <= z < x }
= let ghost Step (x1 y1: int) ← premier cas
  requires { nim_step x1 y1 }
  requires { x = x1 /\ z = y1 }
  ensures { 0 <= z < x } = ()
in
let ghost Path (x1 y z1) ← second cas
  requires { nim x1 y }
  requires { nim y z1 }
  requires { x = x1 /\ z = z1 }
  ensures { 0 <= z < x } =
    nim_bounds x y; nim_bounds y z
in assume { false }; absurd

```

```

let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures { 0 <= z < x }
= let ghost Step (x1 y1: int) ← premier cas
  requires { nim_step x1 y1 }
  requires { x = x1 /\ z = y1 } ← unification
  ensures { 0 <= z < x } = ()
in
let ghost Path (x1 y z1) ← second cas
  requires { nim x1 y }
  requires { nim y z1 }
  requires { x = x1 /\ z = z1 } ← unification
  ensures { 0 <= z < x } =
    nim_bounds x y; nim_bounds y z
in assume { false }; absurd

```

```

let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures { 0 <= z < x }
= let ghost Step (x1 y1: int) ← premier cas
  requires { nim_step x1 y1 }
  requires { x = x1 /\ z = y1 } ← unification
  ensures { 0 <= z < x } = () code utilisateur
in
let ghost Path (x1 y z1) ← second cas
  requires { nim x1 y }
  requires { nim y z1 }
  requires { x = x1 /\ z = z1 } ← unification
  ensures { 0 <= z < x } =
nim_bounds x y; nim_bounds y z code utilisateur
in assume { false }; absurd

```

```

let rec lemma nim_bounds (x z: int)
  requires { nim x z }
  ensures { 0 <= z < x }
= let ghost Step (x1 y1: int) ← premier cas
  requires { nim_step x1 y1 }
  requires { x = x1 /\ z = y1 } ← unification
  ensures { 0 <= z < x } = () code utilisateur
in
let ghost Path (x1 y z1) ← second cas
  requires { nim x1 y }
  requires { nim y z1 }
  requires { x = x1 /\ z = z1 } ← unification
  ensures { 0 <= z < x } =
nim_bounds x y; nim_bounds y z code utilisateur
in assume { false }; absurd ← plus rien à prouver

```

Comme les postconditions des cas sont reprises de la fonction parente, `match inductive` doit être en `position terminale`

```
match inductive H with ... end; ...
```

```
File "test.mlw", line 42, ...
```

```
match inductive should be in tail position
```

Le code fantôme doit terminer, et les fonctions lemmes ne font pas exception

On étend Why3 avec une nouvelle sorte de variant :

```
variant { inductive H }
```

Tout appel récursif doit reposer sur une instance du prédicat **plus petite** que H

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z }
  ensures { 0 <= z < x }
  variant { inductive H }
= match inductive H with
  | Step _ _ _          -> ()
  | Path _ y _ Hxy Hyz -> nim_bounds x y; nim_bounds y z
end
```

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z }
  ensures { 0 <= z < x }
  variant { inductive H }  variant fourni par l'utilisateur
= match inductive H with
  | Step _ _ _          -> ()
  | Path _ y _ Hxy Hyz -> nim_bounds x y; nim_bounds y z
end
```

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z }
  ensures { 0 <= z < x }
  variant { inductive H } ← variant fourni par l'utilisateur
= match inductive H with
  | Step _ _ _          -> ()
  | Path _ y _ Hxy Hyz -> nim_bounds x y; nim_bounds y z
end
```

instances récoltées

```
let rec lemma nim_bounds (x z: int)
  requires H { nim x z }
  ensures { 0 <= z < x }
  variant { inductive H } ← variant fourni par l'utilisateur
= match inductive H with
  | Step _ _ _          -> ()
  | Path _ y _ Hxy Hyz -> nim_bounds x y; nim_bounds y z
end
```

instances récoltées

$(x = x \wedge y = y) \vee (x = y \wedge y = z)$

```

let rec lemma nim_bounds (x z: int)
  requires H { nim x z }
  ensures { 0 <= z < x }
  variant { inductive H } ← variant fourni par l'utilisateur
= match inductive H with
  | Step _ _ _          -> ()
  | Path _ y _ Hxy Hyz -> nim_bounds x y; nim_bounds y z
end

```

instances récoltées

$(x = x \wedge y = y) \vee (x = y \wedge y = z)$
 $(y = x \wedge z = y) \vee (y = y \wedge z = z)$

```

let rec lemma nim_bounds (x z: int)
  requires H { nim x z }
  ensures { 0 <= z < x }
  variant { inductive H } ← variant fourni par l'utilisateur
= match inductive H with
  | Step _ _ _          -> ()
  | Path _ y _ Hxy Hyz -> nim_bounds x y; nim_bounds y z
end

```

instances récoltées

$(x = x \wedge y = y) \vee (x = y \wedge y = z)$ ← H_{yz}
 H_{xy} → $(y = x \wedge z = y) \vee (y = y \wedge z = z)$

4

Une application

Residual theory in λ -calculus : a formal development

- Formalisation du λ -calcul en Coq par Gérard Huet (2 kloc)
- 2 ADTs, 11 prédicats inductifs, 78 preuves par induction

Constructive Computation Theory [Huet, 1988]

Residual theory in λ -calculus : a formal development [Huet, 1994]

- Entièrement traduite vers Why3 avec `match inductive`
- Environ deux fois plus courte que la preuve Coq

Le code fantôme de Why3 est étendu avec :

- une nouvelle construction `match inductive` permettant le filtrage sur les instances de prédicats inductifs
- une nouvelle sorte de `variant` pour justifier les appels récursifs sur ces instances

Extensions implémentées : variants lexicographiques, prédicats mutuellement récursifs
À venir : support des branchements infinis (e.g. accessibilité)

Comme pour tout variant, un variant inductif peut être élément d'un variant tuple lexicographique

e.g.,

```
variant { n, inductive H, t }
```

Les prédicats mutuellement inductifs

```
inductive even int = ...  
with      odd  int = ...
```

induisent des fonctions lemmes mutuellement récursives, avec des variants adaptés

```
let rec lemma evenP (n: int)  
  requires E { even n      }  
  variant    { inductive E }  
  ...  
with lemma oddP (n: int)  
  requires O { odd n       }  
  variant    { inductive O }  
  ...
```