université
PARIS-SACLAY

UE14 · Vérification déductive de programmes

Introduction · Logique de Hoare · WP

Master Informatique · FIIL · 2017 / 2018

Préliminaires

- Séances 15/11, 30/11, 7/12 : Andrei Paskevich (LRI)
- Séances 14/12, 21/12, 11/01 : Julien Signoles (CEA)
- Séance 18/01 : Andrei + Julien (TP noté)
- Examen : 1/02
- Évaluation :
  - 2 interrogations écrites
  - 1 TP noté
  - examen terminal
  - note finale : $\frac{1}{3}\mathrm{CC} + \frac{2}{3}\mathrm{ET}$
- Travaux pratiques (1re partie) : outil WHY3
  - `http://why3.lri.fr/` — site du projet
  - `http://why3.lri.fr/fiil-2017/` — les TPs pour ce cours

Bibliographie : références historiques

- R. W. Floyd. Assigning Meanings to Programs.
  Proc. of Symposia in Applied Mathematics, 1967
- C. A. R. (Tony) Hoare.
  An Axiomatic Basis for Computer Programming.
  Communications of the ACM, 1969
- E. W. Dijkstra. A Discipline of Programming. Prentice Hall, 1976

Bibliographie : références moins historiques

- J.B. Almeida, M.J. Frade, J.S. Pinto, S. Melo de Sousa.
  Rigorous Software Development. Springer, 2011

- F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich.
  Let's Verify This with Why3.
  J. on Software Tools for Technology Transfer (STTT), 17(6), 2015

- F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski.
  Frama-C : a Software Analysis Perspective.
  Proc. of Formal Aspects of Computing, 2015

1. Un peu d’histoire

Software is hard. — Donald Knuth

...

- 1996 : explosion d'*Ariane 5* — conversion *float-to-int* erronée
- 1997 : redémarrages de *Pathfinder* — inversion de priorités
- 1999 : explosion de *Mars Climate Orbiter* — erreur d'unités

...

Software is hard. — DONALD KNUTH

...

- 1996 : explosion d'*Ariane 5* — conversion *float-to-int* erronée
- 1997 : redémarrages de *Pathfinder* — inversion de priorités
- 1999 : explosion de *Mars Climate Orbiter* — erreur d'unités

...

- 2006 : *Debian SSH bug* — RNG prévisible (corrigé en 2008)
- 2012 : *Heartbleed* — dépassement de bornes (corrigé en 2014)
- 1989 : *Shellshock* — contrôle d'entrée insuffisant (corrigé en 2014)

...

Un algo tout simple : Recherche par dichotomie

Problème : trouver une valeur dans un tableau trié.

Premier algorithme publié en 1946.

Première publication sans erreur en 1962.

Un algo tout simple : Recherche par dichotomie

Problème : trouver une valeur dans un tableau trié.

Premier algorithme publié en 1946.

Première publication sans erreur en 1962.

2006 : *Nearly All Binary Searches and Mergesorts are Broken*

(Joshua Bloch, Google, poste de blog)

Le code dans JDK :

```
int mid = (low + high) / 2;
int midVal = a[mid];
```

Un algo tout simple : Recherche par dichotomie

Problème : trouver une valeur dans un tableau trié.

Premier algorithme publié en 1946.

Première publication sans erreur en 1962.

2006 : *Nearly All Binary Searches and Mergesorts are Broken*

(Joshua Bloch, Google, poste de blog)

Le code dans JDK :

```
int mid = (low + high) / 2;
int midVal = a[mid];
```

Bug : l'addition peut dépasser $2^{31} - 1$, la capacité de `int` en Java.

Une solution possible :

```
int mid = low + (high - low) / 2;
```

Assurer l'absence de bugs

Plusieurs approches existent : *model checking*, interprétation abstraite...

Dans ce cours : vérification déductive

1. Fournir une spécification du programme : un modèle mathématique.
2. Construire une preuve que le code correspond à la spécification.

Assurer l'absence de bugs

Plusieurs approches existent : *model checking*, interprétation abstraite...

Dans ce cours : vérification déductive

1. Fournir une spécification du programme : un modèle mathématique.
2. Construire une preuve que le code correspond à la spécification.

Première preuve de programme : Alan Turing, 1949

```
u := 1
for r = 0 to n - 1 do
  v := u
  for s = 1 to r do
    u := u + v
```

Assurer l'absence de bugs

Plusieurs approches existent : *model checking*, interprétation abstraite...

Dans ce cours : vérification déductive

1. Fournir une spécification du programme : un modèle mathématique.
2. Construire une preuve que le code correspond à la spécification.

Première preuve de programme : Alan Turing, 1949

Premier fondement théorique : logique de Floyd-Hoare, 1969

Assurer l'absence de bugs

Plusieurs approches existent : *model checking*, interprétation abstraite...

Dans ce cours : vérification déductive

1. Fournir une spécification du programme : un modèle mathématique.
2. Construire une preuve que le code correspond à la spécification.

Première preuve de programme : Alan Turing, 1949

Premier fondement théorique : logique de Floyd-Hoare, 1969

Premier succès en pratique : la ligne métro 14, 1998

outil : Atelier B, approche par raffinement

D’autres succès majeures

- Logiciel de contrôle de vol de A380, 2005

  preuve de sureté : absence d’erreur d’exécution
  outil : Astrée, interprétation abstraite

  preuve unitaire de propriétés fonctionnelles
  outil : Caveat, vérification déductive

- Hyper-V — hyperviseur natif, 2008

  outils : VCC + prouveur automatique Z3, vérification déductive

- CompCert — compilateur C certifié, 2009

  outil : Coq, génération du code correct par construction

- seL4 — micro-noyau d’un système d’exploitation, 2009

  outil : Isabelle/HOL, vérification déductive

2. La logique de Floyd-Hoare

Langage μML : termes

$$
\begin{array}{rcll}
t & ::= & \ldots, -1, 0, 1, \ldots, 42, \ldots & \text{constantes numériques} \\
 & | & \texttt{true} \mid \texttt{false} & \text{constantes booléennes} \\
 & | & u \mid v \mid w & \text{variables immuables} \\
 & | & x \mid y \mid z & \text{pointeurs déréférencés} \\
 & | & t \; op \; t & \text{opérations binaires} \\
 & | & op \; t & \text{opérations unaires} \\
op & ::= & + \mid - \mid * & \text{opérations arithmétiques} \\
 & | & = \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant & \text{comparaisons arithmétiques} \\
 & | & \wedge \mid \vee \mid \neg & \text{connecteurs logiques}
\end{array}
$$

- deux types de données : entiers non-bornés et booléens
- un terme bien typé évalue sans erreur (pas de division)
- l'évaluation d'un terme ne modifie pas la mémoire du programme

Langage μML : expressions

e	$::=$	`skip`	aucun effet
	$\mid$	t	terme
	$\mid$	x `:=` t	affectation
	$\mid$	e `;` e	séquence
	$\mid$	`let` $v = e$ `in` e	liaison
	$\mid$	`let` $x =$ `ref` e `in` e	allocation
	$\mid$	`if` t `then` e `else` e	conditionnel
	$\mid$	`while` t `do` e `done`	boucle

- trois types : entiers, booléens, and `unit`
- les références (pointeurs) ne sont pas des valeurs de première classe
- les expressions peuvent allouer et modifier la mémoire
- les expressions bien typées s'exécutent sans erreur

Langage μML : expressions bien typées

$$
\begin{array}{lcl}
\texttt{skip} & : & \texttt{unit} \\
t_\tau & : & \tau \\
x_\tau \texttt{ := } t_\tau & : & \texttt{unit} \\
e_{\texttt{unit}} \texttt{ ; } e_\varsigma & : & \varsigma \\
\texttt{let } v_\tau = e_\tau \texttt{ in } e_\varsigma & : & \varsigma \\
\texttt{let } x_\tau = \texttt{ref } e_\tau \texttt{ in } e_\varsigma & : & \varsigma \\
\texttt{if } t_{\texttt{bool}} \texttt{ then } e_\varsigma \texttt{ else } e_\varsigma & : & \varsigma \\
\texttt{while } t_{\texttt{bool}} \texttt{ do } e_{\texttt{unit}} \texttt{ done} & : & \texttt{unit}
\end{array}
$$

- $\tau ::= \texttt{int} \mid \texttt{bool}$ and $\varsigma ::= \tau \mid \texttt{unit}$
- les références (pointeurs) ne sont pas des valeurs de première classe
- les expressions peuvent allouer et modifier la mémoire
- les expressions bien typées s'exécutent sans erreur

Langage μML : sucre syntaxique

$$
\begin{aligned}
x \mathtt{\ :=\ } e &\equiv \mathtt{let}\ v = e\ \mathtt{in}\ x \mathtt{\ :=\ } v \\
\mathtt{if}\ e\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 &\equiv \mathtt{let}\ v = e\ \mathtt{in}\ \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 \\
\mathtt{if}\ e_1\ \mathtt{then}\ e_2 &\equiv \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ \mathtt{skip} \\
e_1\ \&\&\ e_2 &\equiv \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ \mathtt{false} \\
e_1 \mathtt{\ ||\ } e_2 &\equiv \mathtt{if}\ e_1\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ e_2
\end{aligned}
$$

```
let sum = ref 1 in
let count = ref 0 in
while sum ⩽ n do
  count := count + 1;
  sum := sum + 2 * count + 1
done;
count
```

Que renvoie cette expression pour un n donné ?

Exemple — *ISQRT*

```
let sum = ref 1 in
let count = ref 0 in
while sum ⩽ n do
  count := count + 1;
  sum := sum + 2 * count + 1
done;
count
```

Que renvoie cette expression pour un `n` donné ?

Spécification informelle :

- à la fin, `count` contient la racine carrée de `n`, arrondie vers le bas
- par exemple, pour `n = 42`, la valeur finale de `count` est 6

Triplets de Hoare

Une proposition sur la correction d'un programme :

$$\{P\}\; e\; \{Q\}$$

- P formule logique de précondition
- e expression
- Q formule logique de postcondition

Que signifie un triplet de Hoare ?

$\{P\}\; e\; \{Q\}$ si on exécute l'expression e
dans un état de mémoire initial qui satisfait P,
alors soit l'exécution diverge, soit elle termine
dans un état de mémoire final qui satisfait Q

C'est la correction partielle : nous ne prouvons pas la terminaison.

Exemples de triplets valides pour la correction partielle :

- $\{x = 1\}$ x := $x + 2$ $\{x = 3\}$
- $\{x = y\}$ $x + y$ $\{\texttt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}$ $x + 42$ $\{\exists w.\ \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}$ while true do skip done $\{\boxed{\boxed{\texttt{false}}}\}$

Exemples

Exemples de triplets valides pour la correction partielle :

- $\{x = 1\}$ x := $x + 2$ $\{x = 3\}$
- $\{x = y\}$ $x + y$ $\{\texttt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}$ $x + 42$ $\{\exists w.\ \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}$ `while true do skip done` $\{\boxed{\boxed{\texttt{false}}}\}$
  - après cette boucle, *toute propriété* est prouvable
  - *ergo* : ne pas prouver la terminaison peut être fatal

Exemples

Exemples de triplets valides pour la correction partielle :

- $\{x = 1\}$ x := $x + 2$ $\{x = 3\}$
- $\{x = y\}$ $x + y$ $\{\texttt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}$ $x + 42$ $\{\exists w.\ \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}$ `while true do skip done` $\{\boxed{\boxed{\texttt{false}}}\}$
  - après cette boucle, *toute propriété* est prouvable
  - *ergo* : ne pas prouver la terminaison peut être fatal

Dans notre exemple de racine carrée :

$$\{?\}\ \mathit{ISQRT}\ \{?\}$$

Exemples de triplets valides pour la correction partielle :

- $\{x = 1\}$ x := $x + 2$ $\{x = 3\}$
- $\{x = y\}$ $x + y$ $\{\texttt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}$ $x + 42$ $\{\exists w.\ \texttt{result} = 2 * w\}$
- $\{\texttt{true}\}$ `while true do skip done` $\{\boxed{\boxed{\texttt{false}}}\}$
  - après cette boucle, *toute propriété* est prouvable
  - *ergo* : ne pas prouver la terminaison peut être fatal

Dans notre exemple de racine carrée :

$$\{n \geqslant 0\}\ \mathit{ISQRT}\ \{?\}$$

Exemples de triplets valides pour la correction partielle :

- $\{x = 1\}$ $x := x + 2$ $\{x = 3\}$
- $\{x = y\}$ $x + y$ $\{\mathtt{result} = 2 * y\}$
- $\{\exists v.\ x = 4 * v\}$ $x + 42$ $\{\exists w.\ \mathtt{result} = 2 * w\}$
- $\{\mathtt{true}\}$ `while true do skip done` $\{\boxed{\boxed{\mathtt{false}}}\}$
  - après cette boucle, *toute propriété* est prouvable
  - *ergo* : ne pas prouver la terminaison peut être fatal

Dans notre exemple de racine carrée :

$$\{n \geqslant 0\}\ \mathit{ISQRT}\ \{\mathtt{result} * \mathtt{result} \leqslant n < (\mathtt{result} + 1) * (\mathtt{result} + 1)\}$$

Initialement (1970) : la sémantique axiomatique de programmes

Ensemble de règles d'inférence pour construire les triplets valides :

$$\frac{}{\{P\}\ \mathtt{skip}\ \{P\}}$$

$$\frac{}{\{P[x \mapsto t]\}\ x\ \mathtt{:=}\ t\ \{P\}}$$

$$\frac{\{P\}\ e_1\ \{Q\} \qquad \{Q\}\ e_2\ \{R\}}{\{P\}\ e_1\ \mathtt{;}\ e_2\ \{R\}}$$

Notation $P[x \mapsto t]$: remplacer dans P toute occurrence de x par t

La règle de conséquence :

$$\frac{\models P \to P' \qquad \{P'\}\, e\, \{Q'\} \qquad \models Q' \to Q}{\{P\}\, e\, \{Q\}}$$

Exemple : preuve de $\{x = 1\}\; x \texttt{ := } x + 2\; \{x = 3\}$

$$\dfrac{\models x = 1 \to x + 2 = 3 \qquad \dfrac{\{(x = 3)[x \mapsto x + 2]\}\; x \texttt{ := } x + 2\; \{x = 3\}}{\{x + 2 = 3\}\; x \texttt{ := } x + 2\; \{x = 3\}}}{\{x = 1\}\; x \texttt{ := } x + 2\; \{x = 3\}}$$

Les règles pour `if` et `while` :

$$\frac{\{P \wedge t\}\; e_1\; \{Q\} \qquad \{P \wedge \neg t\}\; e_2\; \{Q\}}{\{P\}\; \texttt{if}\; t\; \texttt{then}\; e_1\; \texttt{else}\; e_2\; \{Q\}}$$

$$\frac{\{J \wedge t\}\; e\; \{J\}}{\{J\}\; \texttt{while}\; t\; \texttt{do}\; e\; \texttt{done}\; \{J \wedge \neg t\}}$$

La formule J est un invariant de boucle.

Trouver un bon invariant est une difficulté majeure.

3. Le calcul de plus faible précondition

La plus faible précondition

Comment établir la correction d'un programme ?

Une solution : Edsger Dijkstra, 1975

Transformateur de prédicats $\mathrm{WP}(e, Q)$

e expression

Q postcondition

calcule la précondition minimale P telle que $\{P\}\ e\ \{Q\}$

$$
\begin{array}{rcl}
\mathrm{WP}(\mathtt{skip}, Q) & \equiv & Q \\
\mathrm{WP}(t, Q) & \equiv & Q[\mathtt{result} \mapsto t] \\
\mathrm{WP}(x := t, Q) & \equiv & Q[x \mapsto t] \\
\mathrm{WP}(e_1 \,;\, e_2, Q) & \equiv & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)) \\
\mathrm{WP}(\mathtt{let}\ v = e_1\ \mathtt{in}\ e_2, Q) & \equiv & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[v \mapsto \mathtt{result}]) \\
\mathrm{WP}(\mathtt{let}\ x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2, Q) & \equiv & \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \mapsto \mathtt{result}]) \\
\mathrm{WP}(\mathtt{if}\ t\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2, Q) & \equiv & \quad (t \rightarrow \mathrm{WP}(e_1, Q)) \land \\
 & & (\neg t \rightarrow \mathrm{WP}(e_2, Q))
\end{array}
$$

```
if impair q then r := r + p;
p := p + p;
q := demi q
```

```
if impair q then
    r := r + p
else
    skip;
p := p + p;
q := demi q
```

```
if impair q then
    r := r + p
else
    skip;
p := p + p;
q := demi q
```

$Q[p, q, r]$

```
if impair q then
    r := r + p
else
    skip;
p := p + p;
```
$Q[p, \texttt{demi}\ q, r]$

```
q := demi q
```
$Q[p, q, r]$

```
if impair q then
    r := r + p
else
    skip;
```
$Q[p + p, \texttt{demi}\ q, r]$

```
p := p + p;
```
$Q[p, \texttt{demi}\ q, r]$

```
q := demi q
```
$Q[p, q, r]$

```
if impair q then
    r := r + p
  Q[p + p, demi q, r]
else
    skip;
  Q[p + p, demi q, r]
p := p + p;
Q[p, demi q, r]
q := demi q
Q[p, q, r]
```

```
if impair q then
    Q[p + p, demi q, r + p]
      r := r + p
    Q[p + p, demi q, r]
else
    Q[p + p, demi q, r]
      skip;
    Q[p + p, demi q, r]
  p := p + p;
Q[p, demi q, r]
  q := demi q
Q[p, q, r]
```

$$(\texttt{impair}\ q \rightarrow Q[p+p, \texttt{demi}\ q, r+p]) \wedge$$
$$(\neg\, \texttt{impair}\ q \rightarrow Q[p+p, \texttt{demi}\ q, r])$$

```
if impair q then
  Q[p + p, demi q, r + p]
    r := r + p
  Q[p + p, demi q, r]
else
  Q[p + p, demi q, r]
    skip;
  Q[p + p, demi q, r]
p := p + p;
Q[p, demi q, r]
q := demi q
Q[p, q, r]
```

$$\mathrm{WP}(\texttt{while}\ t\ \texttt{do}\ e\ \texttt{done}, Q) \equiv$$

$\exists J : \texttt{Prop}.$	un invariant J
$J \wedge$	qui est vrai au début
$\forall x_1 \ldots x_k.$	et qui reste vrai
$(J \wedge \quad t \rightarrow \mathrm{WP}(e, J)) \wedge$	après une itération,
$(J \wedge \neg t \rightarrow Q)$	suffit pour prouver Q

$x_1, \ldots, x_k$ références modifiées dans e

On ne connaît pas les valeurs des références modifiées après n itérations

- il faut prouver Q et la préservation de J pour des valeurs arbitraires
- J doit fournir toute l'information nécessaire sur l'état de mémoire

Définition de WP : boucle annotée

Trouver un invariant est difficile dans le cas général

- c'est équivalent à la preuve de Q par induction

Nous pouvons faciliter le travail des outils avec des annotations :

$\mathrm{WP}(\texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{do}\ e\ \texttt{done}, Q) \equiv$	l'invariant indiqué J
$\quad J \wedge$	est vrai au début,
$\quad \forall x_1 \ldots x_k.$	reste vrai
$\qquad (J \wedge t \rightarrow \mathrm{WP}(e, J)) \wedge$	après une itération
$\qquad (J \wedge \neg t \rightarrow Q)$	et suffit pour prouver Q

$x_1, \ldots, x_k$ références modifiées dans e

```
let p = ref a in
let q = ref b in
let r = ref 0 in
while q > 0 invariant J[p,q,r] do
    if impair q then r := r + p;
    p := p + p;
    q := demi q
done;
r
```

$\texttt{result} = a * b$

```
let p = ref a in
let q = ref b in
let r = ref 0 in
while q > 0 invariant J[p,q,r] do
    if impair q then r := r + p;
    p := p + p;
    q := demi q
done;
```

$r = a * b$

```
r
```

```
let p = ref a in
let q = ref b in
let r = ref 0 in
while q > 0 invariant J[p,q,r] do
    if impair q then r := r + p;
    p := p + p;
    q := demi q
  J[p, q, r]
done;
```

$r = a * b$

```
r
```

```
let p = ref a in
let q = ref b in
let r = ref 0 in
while q > 0 invariant J[p,q,r] do
    (impair q → J[p + p, demi q, r + p]) ∧
  (¬ impair q → J[p + p, demi q, r])
    if impair q then r := r + p;
    p := p + p;
    q := demi q
  J[p, q, r]
done;
r = a ∗ b
r
```

```
let p = ref a in
let q = ref b in
let r = ref 0 in
J[p,q,r] ∧
∀p q r. J[p,q,r] →
  (q > 0 →
      (impair q → J[p+p, demi q, r+p]) ∧
    (¬ impair q → J[p+p, demi q, r])) ∧
  (q ⩽ 0 →
    r = a ∗ b)
while q > 0 invariant J[p,q,r] do
    if impair q then r := r + p;
    p := p + p;
    q := demi q
done;
r
```

La multiplication du paysan russe

$$
\begin{array}{l}
J[a,b,0] \land \\
\forall p\,q\,r.\ J[p,q,r] \to \\
\quad (q > 0 \to \\
\qquad (\texttt{impair}\ q \to J[p+p, \texttt{demi}\ q, r+p]) \land \\
\quad\ \ (\neg\, \texttt{impair}\ q \to J[p+p, \texttt{demi}\ q, r])) \land \\
\quad (q \leqslant 0 \to \\
\quad\ \ r \ = \ a * b)
\end{array}
$$

```
let p = ref a in
let q = ref b in
let r = ref 0 in
while q > 0 invariant J[p,q,r] do
    if impair q then r := r + p;
    p := p + p;
    q := demi q
done;
r
```

Théorème (Cohérence)

Pour toute expression e et postcondition Q,
le triplet $\{\mathrm{WP}(e, Q)\}\ e\ \{Q\}$ est valide.

Preuve par induction sur la structure de l'expression e.

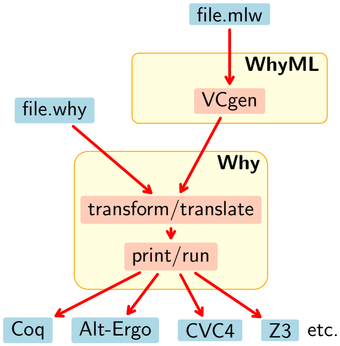
Corollaire

Pour prouver que le triplet $\{P\}\ e\ \{Q\}$ est valide,
il suffit de prouver la formule $P \rightarrow \mathrm{WP}(e, Q)$.

C'est ce que fait Why3

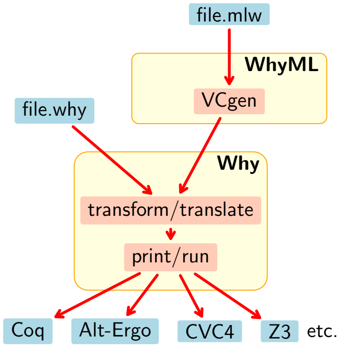
4. Prise en main de WHY3

WHY3 in a nutshell



WHYML, un langage de programmation

- polymorphisme de types • variants
- notion d'ordre supérieur
- *pattern matching* • exceptions
- code et données fantômes
- état mutable avec contrôle des *alias*
- contrats • invariants de type

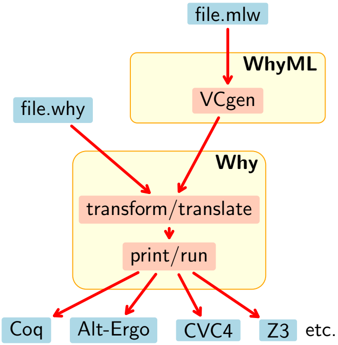


WHYML, un langage de programmation

- polymorphisme de types • variants
- notion d'ordre supérieur
- *pattern matching* • exceptions
- code et données fantômes
- état mutable avec contrôle des *alias*
- contrats • invariants de type

...et aussi de spécification

- types algébriques polymorphes
- notion d'ordre supérieur
- prédicats inductifs



WHY3 in a nutshell

WHYML, un langage de programmation

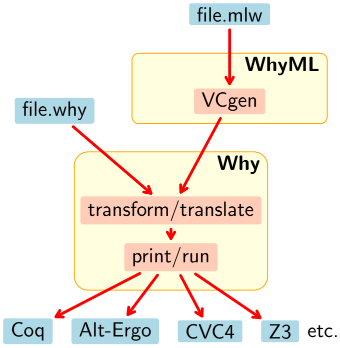
- polymorphisme de types • variants
- notion d'ordre supérieur
- *pattern matching* • exceptions
- code et données fantômes
- état mutable avec contrôle des *alias*
- contrats • invariants de type

WHY3, un outil de vérification

- génère des VC via WP ou *fast WP*
- fournit 70+ transformations de VC
- sait parler à 25+ outils ATP et ITP

...et aussi de spécification

- types algébriques polymorphes
- notion d'ordre supérieur
- prédicats inductifs



WHY3 out of the nutshell

Trois façons différentes de se servir de WHY3

- un langage logique confortable
  - une interface commune pour plusieurs prouveurs
- un langage de programmation destiné à la preuve
  - voir des exemples dans notre galerie `http://toccata.lri.fr/gallery/why3.en.html`
- un outil intermédiaire de vérification
  - programmes C — Frama-C
  - programmes Java — Krakatoa
  - programmes Ada — SPARK 2014
  - programmes probabilistes — EasyCrypt

Premiers pas

1. Se rendre à `http://why3.lri.fr/fiil-2017/`
2. Noter la barre d'outils en haut de la page.
3. Charger l'exemple « ISQRT » dans la liste déroulante sur la barre.
4. L'accès aux valeurs des références se fait à la OCaml : `!x`
5. Exécuter le programme avec le bouton « *Execute* » sur la barre.
6. Vérifier le programme avec le bouton « *Verify* » sur la barre.
7. Appliquer « *Split and prove* » à l'obligation de preuve « VC for isqrt ».
8. Étudier les tâches de preuve générées dans l'onglet « *Task view* ».
9. Passer aux exercices.

5. Exercices

Exercice 1 — Addition inefficace

Ce programme calcule (lentement) la somme $a+b$:

```
let x = ref a in
let y = ref b in
while !y > 0 do
  x := !x + 1;
  y := !y - 1
done;
!x
```

1. Proposer une postcondition qui exprime que le résultat calculé est la somme $a+b$.
2. Trouver un invariant de boucle approprié.
3. Prouver le programme.

Exercice 2 — Division Euclidienne

Ce programme est un des exemples originaux de Floyd :

```
let q = ref 0 in
let r = ref a in
while !r >= b do
  r := !r - b;
  q := !q + 1
done;
(!q, !r)
```

1. Proposer une précondition qui exprime que
   a est présumé positif ou nul, et *b* est présumé strictement positif.
2. Proposer une postcondition qui exprime que
   les valeurs finales de *q* et *r* sont, respectivement,
   le quotient et le reste de la division Euclidienne de *a* par *b*.
3. Trouver un invariant de boucle approprié.
4. Prouver le programme.

Exercice 3 — Multiplication du paysan russe

Supposons que l'on dispose de deux opérations binaires `div` et `mod` qui renvoient, respectivement, le quotient et le reste de la division Euclidienne.

Ce programme calcule dans r le produit $a * b$:

```
let p = ref a in
let q = ref b in
let r = ref 0 in
while !q > 0 do
  if mod !q 2 = 1 then r := !r + !p;
  p := !p + !p;
  q := div !q 2
done;
!r
```

1. Donner les pré- et les postconditions appropriées.
2. Trouver un invariant de boucle convenable.
3. Prouver le programme.

Exercice 4 — Affectation alternative

1. Prouver que le triplet $\{P\}\, x := t\, \{\exists v.\, t[x \mapsto v] = x \wedge P[x \mapsto v]\}$ est prouvable à partir des règles standards de la logique de Hoare.

Supposons que l'on remplace la règle pour l'affectation par une autre :

$$\frac{}{\{P\}\, x := t\, \{\exists v.\, t[x \mapsto v] = x \wedge P[x \mapsto v]\}}$$

2. Montrer que le triplet $\{P[x \mapsto t]\}\, x := t\, \{P\}$ est prouvable à partir de la nouvelle règle.