Introduction - Logique de Hoare - WP

Amélie Ledein

D'après le matériel de cours de
Andrei Paskevich et Julien Signoles

Démontrer plus vite que son ombre

Considérons le programme suivant :

```
let isqrt (n : int) : int =
  let ref sum = 1 in
  let ref count = 0 in
  while sum <= n do
    count <- count + 1;
    sum <- sum + 2 * count + 1
  done;
  count
```

Comment le démontrer en 5 minutes ?

Démontrer plus vite que son ombre

Considérons le programme suivant :

```
let isqrt (n : int) : int =
  let ref sum = 1 in
  let ref count = 0 in
  while sum <= n do
    count <- count + 1;
    sum <- sum + 2 * count + 1
  done;
  count
```

Comment le démontrer en 5 minutes ?

- Écrire la spécification

Démontrer plus vite que son ombre

Considérons le programme suivant :

```
let isqrt (n : int) : int =
  let ref sum = 1 in
  let ref count = 0 in
  while sum <= n do
    count <- count + 1;
    sum <- sum + 2 * count + 1
  done;
  count
```

Comment le démontrer en 5 minutes ?

- Écrire la spécification
- Écrire les variants et invariants de boucle

Démontrer plus vite que son ombre

Considérons le programme suivant :

```
let isqrt (n : int) : int =
  let ref sum = 1 in
  let ref count = 0 in
  while sum <= n do
    count <- count + 1;
    sum <- sum + 2 * count + 1
  done;
  count
```

Comment le démontrer en 5 minutes ?

- Écrire la spécification
- Écrire les variants et invariants de boucle
- Utiliser un outil basé sur le calcul de plus faible pré-condition

Démontrer plus vite que son ombre

```
function sqr (n: int) : int = n * n

let isqrt (n: int): int
  requires { n >= 0 }
  ensures  { result >= 0
           /\ sqr result <= n < sqr (result + 1) }
 =
  let ref sum = 1 in
  let ref count = 0 in
  while sum <= n do
   count <- count + 1;
   sum <- sum + 2 * count + 1
  done;
  count
```

Démontrer plus vite que son ombre

```
function sqr (n : int) : int = n * n

let isqrt (n: int): int
  requires { n >= 0 }
  ensures  { result >= 0
           /\ sqr result <= n < sqr (result + 1) }
=
 let ref sum = 1 in
 let ref count = 0 in
 while sum <= n do
  invariant { count >= 0 }
  invariant { sqr count <= n }
  invariant { sqr (count + 1) = sum }
  variant   { n - count }
  count <- count + 1;
  sum <- sum + 2 * count + 1
 done;
 count
```

Langage μML : termes

t	$::=$	$\ldots, -1, 0, 1, \ldots, 42, \ldots$	constantes numériques
	$\mid$	`true` $\mid$ `false`	constantes booléennes
	$\mid$	$u \mid v \mid w$	variables immuables
	$\mid$	$x \mid y \mid z$	pointeurs déréférencés
	$\mid$	$t\ op\ t$	opérations binaires
	$\mid$	$op\ t$	opérations unaires
op	$::=$	$+ \mid - \mid *$	opérations arithmétiques
	$\mid$	$= \mid \neq \mid < \mid > \mid \leqslant \mid \geqslant$	comparaisons arithmétiques
	$\mid$	$\wedge \mid \vee \mid \neg$	connecteurs logiques

- 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
	$\mid$	$x \leftarrow t$	affectation
	$\mid$	e `;` e	séquence
	$\mid$	`let` $v = e$ `in` e	liaison
	$\mid$	`let ref` $x = e$ `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 1re classe
- les expressions peuvent allouer et modifier la mémoire
- les expressions bien typées s'exécutent sans erreur

Langage μML : expressions bien typées

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

- $\tau ::= \texttt{int} \mid \texttt{bool}$ 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

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

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

Exemples de triplets valides pour la correction partielle :

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

Dans notre exemple de racine carrée :

$$\{n \geqslant 0\}\ \mathit{ISQRT}\ \{\texttt{result}^2 \leqslant n < (\texttt{result} + 1)^2\}$$

Initialement (1970) : la sémantique axiomatique de programmes

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

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}$$

$$\frac{}{\{P[x \mapsto t]\}\ x \leftarrow t\ \{P\}}$$

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

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 $\{x = 1\}\, x \leftarrow x + 2\, \{x = 3\}$

$$\dfrac{\models x = 1 \to x + 2 = 3 \qquad \dfrac{\{(x = 3)[x \mapsto x + 2]\}\, x \leftarrow x + 2\, \{x = 3\}}{\{x + 2 = 3\}\, x \leftarrow x + 2\, \{x = 3\}}}{\{x = 1\}\, x \leftarrow x + 2\, \{x = 3\}}$$

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.

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 $\{P\}\, e\, \{Q\}$

Intuition of WP

$\{\ 3xy \text{ est pair}\ \}$ $\quad x \leftarrow 3 * x * y \quad$ $\{\ x \text{ est pair}\ \}$

$\{\ Q[s]\ \}$ $\quad x \leftarrow s \quad$ $\{\ Q[x]\ \}$

$\{\ \text{if } c \text{ then } P_1 \text{ else } P_2\ \}$ $\quad \texttt{if } c \texttt{ then } P_1\ e_1\ Q \texttt{ else } P_2\ e_2\ Q \quad$ $\{\ Q\ \}$

$\{\ \text{if } c \text{ then } P \text{ else } Q\ \}$ $\quad \texttt{if } c \texttt{ then } P\ e\ Q \quad$ $\{\ Q\ \}$

? $\quad \texttt{while } c \texttt{ do } e \texttt{ done} \quad$ $\{\ Q\ \}$

Définition de WP

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q) &\equiv Q \\
\mathrm{WP}(t, Q) &\equiv Q[\texttt{result} \mapsto t] \\
\mathrm{WP}(x \leftarrow t, Q) &\equiv Q[x \mapsto t] \\
\mathrm{WP}(e_1 \texttt{ ; } e_2, Q) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)) \\
\mathrm{WP}(\texttt{let } v = e_1 \texttt{ in } e_2, Q) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[v \mapsto \texttt{result}]) \\
\mathrm{WP}(\texttt{let ref } x = e_1 \texttt{ in } e_2, Q) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q)[x \mapsto \texttt{result}]) \\
\mathrm{WP}(\texttt{if } t \texttt{ then } e_1 \texttt{ else } e_2, Q) &\equiv (t \to \mathrm{WP}(e_1, Q)) \wedge \\
&\quad\ (\neg t \to \mathrm{WP}(e_2, Q))
\end{aligned}
$$

Remonter le courant

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

Remonter le courant

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

$Q[p, q, r]$

Remonter le courant

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

Remonter le courant

```
if impair q then
    r ← r + p
else
    skip;
```

$Q[p + p, \texttt{demi}\ q, r]$

$p \leftarrow p + p$;

$Q[p, \texttt{demi}\ q, r]$

$q \leftarrow \texttt{demi}\ q$

$Q[p, q, r]$

Remonter le courant

```
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]
```

Remonter le courant

```
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]
```

Remonter le courant

$$
\begin{array}{l}
\quad (\texttt{impair}\ q \to Q[p + p, \texttt{demi}\ q, r + p]) \wedge \\
(\neg\, \texttt{impair}\ q \to Q[p + p, \texttt{demi}\ q, r]) \\
\quad \texttt{if impair}\ q\ \texttt{then} \\
\qquad Q[p + p, \texttt{demi}\ q, r + p] \\
\qquad\quad r \leftarrow r + p \\
\qquad Q[p + p, \texttt{demi}\ q, r] \\
\quad \texttt{else} \\
\qquad Q[p + p, \texttt{demi}\ q, r] \\
\qquad\quad \texttt{skip}\,; \\
\qquad Q[p + p, \texttt{demi}\ q, r] \\
\quad p \leftarrow p + p\,; \\
Q[p, \texttt{demi}\ q, r] \\
\quad q \leftarrow \texttt{demi}\ q \\
Q[p, q, r]
\end{array}
$$

Définition de WP : boucle

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

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

$$\begin{aligned}
&\mathrm{WP}(\texttt{while } t \texttt{ invariant } J \texttt{ do } e \texttt{ done}, Q) \equiv \\
&\quad J \wedge \\
&\quad \forall x_1 \ldots x_k. \\
&\qquad (J \wedge \quad t \rightarrow \mathrm{WP}(e, J)) \wedge \\
&\qquad (J \wedge \neg t \rightarrow Q)
\end{aligned}$$

l'invariant indiqué J
est vrai au début,
reste vrai
après une itération
et suffit pour prouver Q

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

La multiplication du paysan russe

```
let ref p = a in
let ref q = b in
let ref r = 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
```

result = $a * b$

La multiplication du paysan russe

```
let ref p = a in
let ref q = b in
let ref r = 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
```

La multiplication du paysan russe

```
let ref p = a in
let ref q = b in
let ref r = 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
```

La multiplication du paysan russe

```
let ref p = a in
let ref q = b in
let ref r = 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
```

La multiplication du paysan russe

$$
\begin{array}{l}
\texttt{let ref } p = a \texttt{ in} \\
\texttt{let ref } q = b \texttt{ in} \\
\texttt{let ref } r = 0 \texttt{ in} \\
J[p,q,r] \wedge \\
\forall p\, q\, r.\ J[p,q,r] \rightarrow \\
\quad (q > 0 \rightarrow \\
\qquad (\texttt{impair } q \rightarrow J[p+p, \texttt{demi } q, r+p]) \wedge \\
\qquad (\neg\, \texttt{impair } q \rightarrow J[p+p, \texttt{demi } q, r])) \wedge \\
\quad (q \leqslant 0 \rightarrow \\
\qquad r = a * b) \\
\quad \texttt{while } q > 0 \texttt{ invariant } J[p,q,r] \texttt{ do} \\
\qquad \texttt{if impair } q \texttt{ then } r \leftarrow r + p\,; \\
\qquad p \leftarrow p + p\,; \\
\qquad q \leftarrow \texttt{demi } q \\
\quad \texttt{done}\,; \\
\quad r
\end{array}
$$

La multiplication du paysan russe

$$
\begin{array}{l}
J[a,b,0] \wedge \\
\forall p\,q\,r.\ J[p,q,r] \rightarrow \\
\quad (q > 0 \rightarrow \\
\qquad (\texttt{impair}\ q \rightarrow J[p+p, \texttt{demi}\ q, r+p]) \wedge \\
\qquad (\neg\ \texttt{impair}\ q \rightarrow J[p+p, \texttt{demi}\ q, r])) \wedge \\
\quad (q \leqslant 0 \rightarrow \\
\qquad r = a * b) \\
\quad \texttt{let ref}\ p = a\ \texttt{in} \\
\quad \texttt{let ref}\ q = b\ \texttt{in} \\
\quad \texttt{let ref}\ r = 0\ \texttt{in} \\
\quad \texttt{while}\ q > 0\ \texttt{invariant}\ J[p,q,r]\ \texttt{do} \\
\qquad \texttt{if impair}\ q\ \texttt{then}\ r \leftarrow r + p\,; \\
\qquad p \leftarrow p + p\,; \\
\qquad q \leftarrow \texttt{demi}\ q \\
\quad \texttt{done}\,; \\
\quad r
\end{array}
$$

Théorème (Cohérence)

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

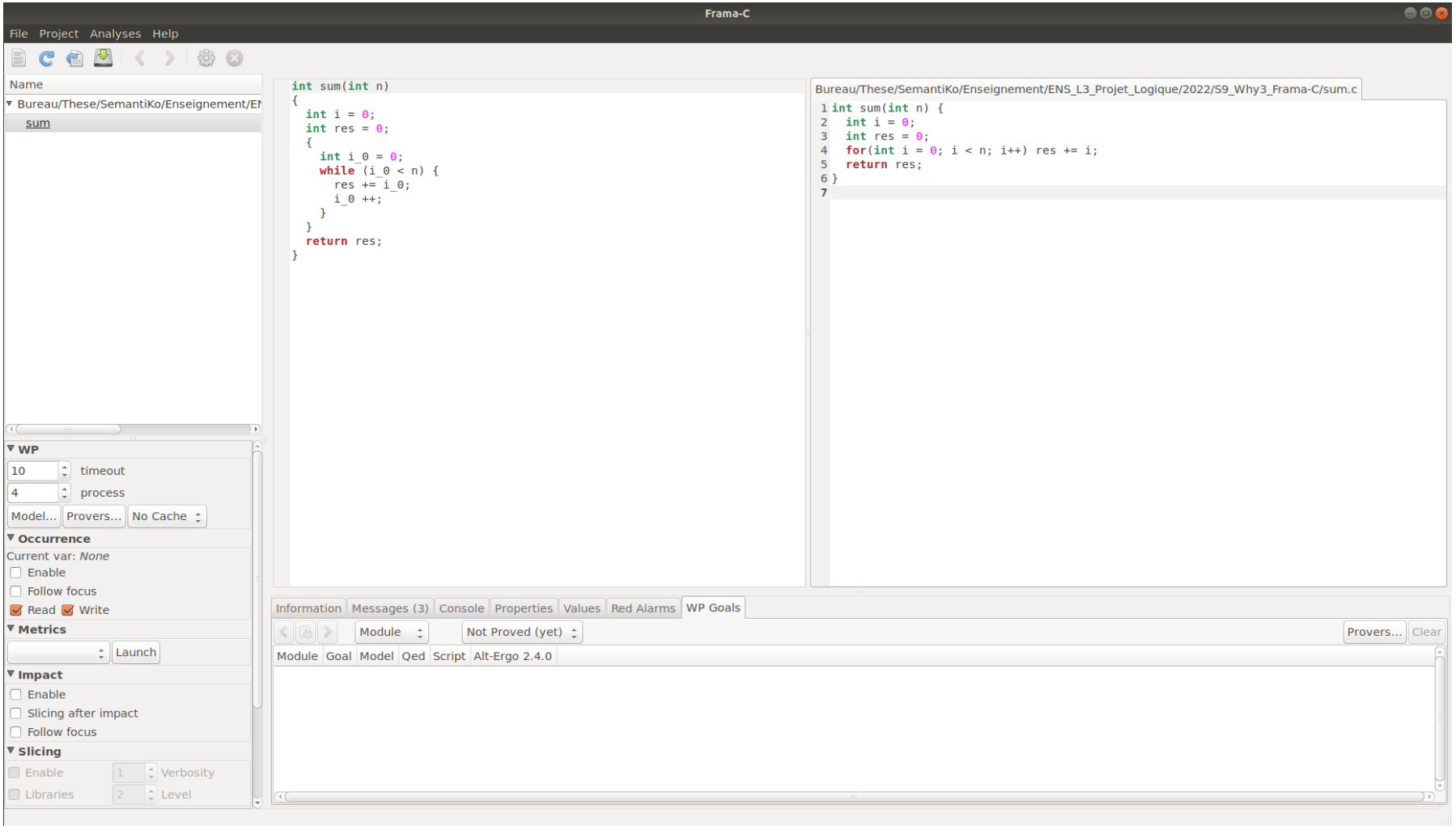
Preuve par induction sur la structure de l'expression e.

Corollaire

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

C'est ce que fait WHY3

Démonstration de Frama-C



WHY3 in a nutshell

WHYML, un langage de programmation

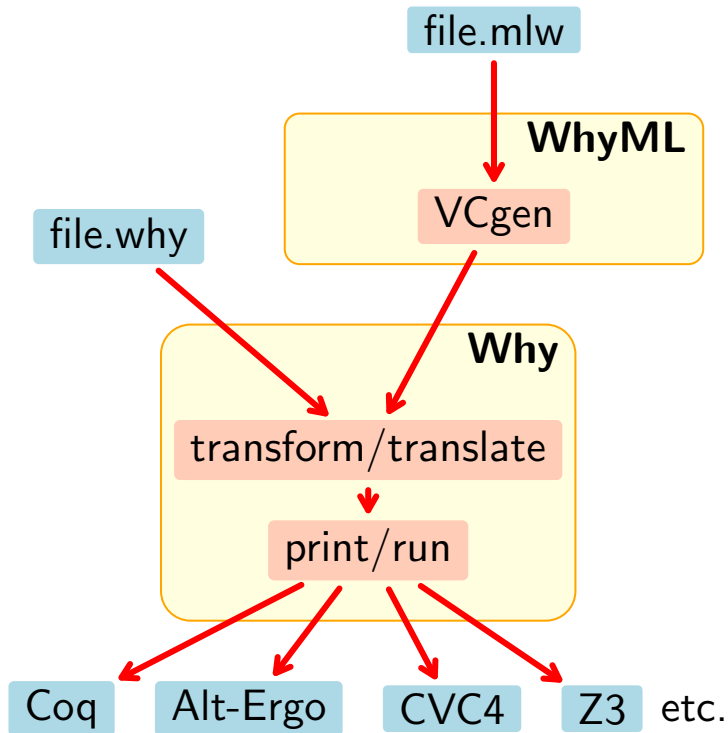
- polymorphisme de types • variants
- notion d'ordre supérieur
- *pattern matching* • exceptions
- code et données fantômes
- état mutable avec contrôle des *alias*
- contrats • invariants de type

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



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

TP avec Why3

1. Aller à l'adresse suivante : http://why3.lri.fr/try/.
2. Charger l'exemple `isqrt_solution.mlw`.
3. Exécuter le programme avec le bouton "Execute" sur la barre.
4. Vérifier le programme avec le bouton "Verify" sur la barre.
5. Appliquer "Split and prove" à l'obligation de preuve "VC for isqrt".
6. Étudier les tâches de preuve générées dans l'onglet "Task view".
7. Passer aux exercices à rendre sur eCampus avant le 12/04/2023 minuit :
   - `dutch.mlw`
   - `kadane.mlw`