université
PARIS-SACLAY

UE14 · Vérification déductive de programmes

Sûreté · Fonctions · Terminaison · Exceptions

Master Informatique · FIIL · 2017 / 2018

1. Sûreté

Sûreté d'exécution

Certaines opérations peuvent échouer pendant l'exécution
si leurs (pré)conditions de sûreté ne sont pas respectées :

- opérations arithmétiques : division par zéro, débordements...
- accès mémoire : les pointeurs `NULL`, dépassement de bornes...
- assertions

Prenons la sûreté en compte dans la sémantique axiomatique :

$\{P\}\, e\, \{Q\}$ si on exécute l'expression e
dans un état de mémoire initial qui satisfait P,
alors il n'y a pas d'erreurs pendant l'exécution
et soit l'exécution diverge, soit elle termine
dans un état de mémoire final qui satisfait Q

Assertions

Une expression de plus :

$$
\begin{array}{rcll}
e & ::= & \ldots & \textit{voir le 1}^{er}\ \textit{cours} \\
 & | & \texttt{assert}\ R & \text{assertion } (R \text{ est une formule})
\end{array}
$$

La règle d'inférence dans la logique de Hoare :

$$\frac{}{\{R \wedge Q\}\,\texttt{assert}\ R\,\{Q\}}$$

La règle de calcul de plus faible précondition :

$$\mathrm{WP}(\texttt{assert}\ R, Q) \quad = \quad R \wedge Q \quad = \quad R \wedge (R \rightarrow Q)$$

La seconde version est utile en pratique pour la vérification déductive.

Opérations partielles

Nous pourrions ajouter d'autres opérations avec des préconditions :

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & t \texttt{ div } t & \text{division euclidienne} \\
 & | & a\texttt{[}\,t\,\texttt{]} & \text{accès à un tableau} \\
 & | & \ldots &
\end{array}
$$

et définir les règles de WP pour ces opérations :

$$
\begin{aligned}
\mathrm{WP}(t_1 \texttt{ div } t_2, Q) &\equiv t_2 \neq 0 \wedge Q[\texttt{result} \mapsto (t_1 \texttt{ div } t_2)] \\
\mathrm{WP}(a\texttt{[}t\texttt{]}, Q) &\equiv 0 \leqslant t < |a| \wedge Q[\texttt{result} \mapsto a\texttt{[}t\texttt{]}] \\
&\ldots
\end{aligned}
$$

Mais autant laisser les programmeurs s'en occuper.

2. Fonctions et contrats

Fonctions et contrats

On peut déléguer une partie de fonctionnalité à une fonction :

$\texttt{let}\ f\ (v_1 : \tau_1) \ldots (v_n : \tau_n) : \varsigma\ \mathscr{C} = e$	fonction définie
$\texttt{val}\ f\ (v_1 : \tau_1) \ldots (v_n : \tau_n) : \varsigma\ \mathscr{C}$	fonction abstraite

Le comportement de fonction est décrit par un contrat :

$\mathscr{C} ::=$	$\texttt{requires}\ P$	précondition
	$\texttt{writes}\ x_1 \ldots x_k$	références globales modifiées
	$\texttt{ensures}\ Q$	postcondition

x° dans postcondition Q dénote la valeur initial d'une référence globale :

```
let incr_r (v: int): int writes x
  ensures result = x° ∧ x = x° + v
= let u = x in x := u + v ; u
```

Fonctions et contrats

On peut déléguer une partie de fonctionnalité à une fonction :

$\texttt{let}\ f\ (v_1 : \tau_1) \ldots (v_n : \tau_n) : \varsigma\ \mathscr{C} = e$	fonction définie
$\texttt{val}\ f\ (v_1 : \tau_1) \ldots (v_n : \tau_n) : \varsigma\ \mathscr{C}$	fonction abstraite

Le comportement de fonction est décrit par un contrat :

$\mathscr{C} ::=$	$\texttt{requires}\ P$	précondition
	$\texttt{writes}\ x_1 \ldots x_k$	références globales modifiées
	$\texttt{ensures}\ Q$	postcondition

x° dans postcondition Q dénote la valeur initial d'une référence globale

Condition de vérification ($\vec{x}$ sont toutes les références globales dans f) :

$$\mathrm{VC}(\texttt{let}\ f \ldots) \quad \equiv \quad \forall \vec{x}\,\vec{v}.\, P \to \mathrm{WP}(e, Q)[\vec{x}^\circ \mapsto \vec{x}]$$

Appel de fonction

Une expression de plus :

$$\begin{array}{rcll} e & ::= & \ldots & \\ & \mid & f\ t\ \ldots\ t & \text{appel de fonction} \end{array}$$

et la règle de plus faible précondition associée :

$$\begin{aligned} \mathrm{WP}(f\ t_1\ \ldots\ t_n, Q) \quad \equiv \quad & P_f[\vec{v} \mapsto \vec{t}] \wedge \\ & (\forall \vec{\boldsymbol{x}}\, \forall \mathtt{result}.\, Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \to Q)[\vec{w} \mapsto \vec{x}] \end{aligned}$$

P_f	précondition de f	$\vec{\boldsymbol{x}}$	références modifiées dans f
Q_f	postcondition de f	$\vec{x}$	références utilisées dans f
$\vec{v}$	paramètres formels de f	$\vec{w}$	variables fraîches

Preuve modulaire : dans la vérification d'un appel de fonction, on n'utilise que le contrat de la fonction appelée, non pas son code.

Exemples en WhyML

```
let max (x y: int) : int
  ensures  { result >= x /\ result >= y }
  ensures  { result =  x \/ result =  y }
= if x >= y then x else y
```

```
val r : ref int  (* déclarer une référence globale *)

let incr_r (v: int) : int
  requires { v > 0 }
  writes   { r }
  ensures  { result = old !r /\ !r = old !r + v }
= let u = !r in
  r := u + v;
  u
```

3. Terminaison

Terminaison

Correction totale : correction partielle $+$ le programme termine
pour tout état initial qui satisfait la précondition.

Il suffit de démontrer que

- toute boucle fait un nombre fini d'itérations
- les appels récursifs ne peuvent pas s'enchaîner indéfiniment

Terminaison

Correction totale : correction partielle $+$ le programme termine pour tout état initial qui satisfait la précondition.

Il suffit de démontrer que

- toute boucle fait un nombre fini d'itérations
- les appels récursifs ne peuvent pas s'enchaîner indéfiniment

Solution : prouver que chaque itération d'une boucle et chaque appel récursif fait décroître une certaine valeur, dite variant, par rapport à un ordre bien fondé (c-à-d. sans chaînes décroissantes infinies).

Par exemple, pour les entiers signés, un ordre bien fondé bien pratique :

$$i \prec j \quad = \quad i < j \wedge 0 \leq j$$

Terminaison des boucles

Une nouvelle annotation :

$$
\begin{array}{rcl}
e & ::= & \ldots \\
 & | & \texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{variant}\ t \cdot \prec\ \texttt{do}\ e\ \texttt{done}
\end{array}
$$

La règle de calcul de plus faible précondition :

$$
\begin{array}{l}
\mathrm{WP}(\texttt{while}\ t\ \texttt{invariant}\ J\ \texttt{variant}\ s \cdot \prec\ \texttt{do}\ e\ \texttt{done}, Q) \equiv \\
\quad J \wedge \\
\quad \forall x_1 \ldots x_k. \\
\qquad (J \wedge\ \ t \rightarrow \mathrm{WP}(e, J \wedge s \prec w)[w \mapsto s]) \wedge \\
\qquad (J \wedge \neg t \rightarrow Q)
\end{array}
$$

$x_1 \ldots x_k$ références modifiées dans e

w variable fraîche (le variant au début de l'itération)

Terminaison de fonctions récursives

Une nouvelle clause de contrat :

$$\begin{aligned}&\texttt{let rec } f\ (v_1 : \tau_1) \ldots (v_n : \tau_n) : \varsigma\\ &\qquad \texttt{requires } P_f\\ &\qquad \texttt{variant } s \cdot \prec\\ &\qquad \texttt{writes } \vec{\boldsymbol{x}}\\ &\qquad \texttt{ensures } Q_f\\ &\quad = e\end{aligned}$$

Pour tout appel récursif de f dans e :

$$\begin{aligned}\mathbf{WP}(f\ t_1 \ldots t_n, Q) \equiv\ & P_f[\vec{v} \mapsto \vec{t}] \wedge s[\vec{v} \mapsto \vec{t}] \prec s[\vec{x} \mapsto \vec{x}^\circ] \wedge\\ & (\forall \vec{\boldsymbol{x}}\, \forall \texttt{result}.\, Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^\circ \mapsto \vec{w}] \to Q)[\vec{w} \mapsto \vec{x}]\end{aligned}$$

$s[\vec{v} \mapsto \vec{t}]$	le variant au moment de l'appel récursif
$s[\vec{x} \mapsto \vec{x}^\circ]$	le variant à l'entrée de l'appelant

Récursion mutuelle

Les fonctions mutuellement récursives doivent avoir

- chacune son propre terme de variant
- l'ordre bien fondé commun pour toutes

Si la fonction f appelle $g\ t_1\ \ldots\ t_n$, la précondition de décroissance sera

$$s_g[\vec{v}_g \mapsto \vec{t}] \prec s_f[\vec{x} \mapsto \vec{x}^\circ]$$

$\vec{v}_g$	les paramètres formels de g
$s_g[\vec{v}_g \mapsto \vec{t}]$	le variant de g au moment de l'appel
$s_f[\vec{x} \mapsto \vec{x}^\circ]$	le variant de f à l'entrée

Exemples

Exercice : trouver les variants convenables

```
i := 0;
while i <= 100 variant ???
do
  i := i+1
done
```

```
sum := 1; res := 0;
while sum <= n variant ???
do
  res := res + 1; sum := sum + 2 * res + 1
done
```

4. Exceptions

Exceptions en tant que destinations

L'exécution d'un programme peut mener à

- la divergence — le calcul ne s'arrête jamais
  - la correction totale nous garantit contre la non-terminaison
- la terminaison anormale — le calcul échoue
  - la correction partielle nous garantit contre les erreurs
- la terminaison normale — le calcul produit un résultat
  - la correction partielle garantit la conformité au contrat

Exceptions en tant que destinations

L'exécution d'un programme peut mener à

- la divergence — le calcul ne s'arrête jamais
  - la correction totale nous garantit contre la non-terminaison
- la terminaison anormale — le calcul échoue
  - la correction partielle nous garantit contre les erreurs
- la terminaison normale — le calcul produit un résultat
  - la correction partielle garantit la conformité au contrat
- la terminaison exceptionnelle — produit *un autre genre de résultat*

Exceptions en tant que destinations

L'exécution d'un programme peut mener à

- la divergence — le calcul ne s'arrête jamais
  - la correction totale nous garantit contre la non-terminaison
- la terminaison anormale — le calcul échoue
  - la correction partielle nous garantit contre les erreurs
- la terminaison normale — le calcul produit un résultat
  - la correction partielle garantit la conformité au contrat
- la terminaison exceptionnelle — produit *un autre genre de résultat*
  - le contrat doit couvrir la terminaison exceptionnelle
  - toute exception potentielle E reçoit une postcondition associée Q_E
  - la correction partielle : *si* E *est levée, alors l'état final satisfait* Q_E

Exceptions en tant que destinations

L'exécution d'un programme peut mener à

- la divergence — le calcul ne s'arrête jamais
  - la correction totale nous garantit contre la non-terminaison
- la terminaison anormale — le calcul échoue
  - la correction partielle nous garantit contre les erreurs
- la terminaison normale — le calcul produit un résultat
  - la correction partielle garantit la conformité au contrat
- la terminaison exceptionnelle — produit *un autre genre de résultat*

```
exception Not_found

let binary_search (a: array int) (v: int) : int
  requires { forall i j. 0 ⩽ i ⩽ j < length a → a[i] ⩽ a[j] }
  ensures  { 0 ⩽ result < length a ∧ a[result] = v }
  raises   { Not_found → forall i. 0 ⩽ i < length a → a[i] ≠ v }
```

Un autre point-virgule

Notre langage grandit encore :

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{raise E} & \text{lever une exception} \\
 & | & \texttt{try } e \texttt{ with E} \to e & \text{attraper une exception}
\end{array}
$$

WP doit gérer deux postconditions :

$$\mathrm{WP}(\texttt{skip}, Q, Q_{\mathsf{E}}) \equiv Q$$

Un autre point-virgule

Notre langage grandit encore :

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{raise E} & \text{lever une exception} \\
 & | & \texttt{try } e \texttt{ with E} \to e & \text{attraper une exception}
\end{array}
$$

WP doit gérer deux postconditions :

$$\mathrm{WP}(\texttt{skip}, Q, Q_{\mathsf{E}}) \equiv Q$$

$$\mathrm{WP}(\texttt{raise E}, Q, Q_{\mathsf{E}}) \equiv Q_{\mathsf{E}}$$

Un autre point-virgule

Notre langage grandit encore :

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{raise E} & \text{lever une exception} \\
 & | & \texttt{try } e \texttt{ with E} \to e & \text{attraper une exception}
\end{array}
$$

WP doit gérer deux postconditions :

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q, Q_\mathrm{E}) &\equiv Q \\
\mathrm{WP}(\texttt{raise E}, Q, Q_\mathrm{E}) &\equiv Q_\mathrm{E} \\
\mathrm{WP}(e_1 \texttt{ ; } e_2, Q, Q_\mathrm{E}) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_\mathrm{E}), Q_\mathrm{E})
\end{aligned}
$$

Un autre point-virgule

Notre langage grandit encore :

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{raise E} & \text{lever une exception} \\
 & | & \texttt{try } e \texttt{ with E} \to e & \text{attraper une exception}
\end{array}
$$

WP doit gérer deux postconditions :

$$
\begin{aligned}
\mathrm{WP}(\texttt{skip}, Q, Q_\mathsf{E}) &\equiv Q \\
\mathrm{WP}(\texttt{raise E}, Q, Q_\mathsf{E}) &\equiv Q_\mathsf{E} \\
\mathrm{WP}(e_1 \texttt{ ; } e_2, Q, Q_\mathsf{E}) &\equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_\mathsf{E}), Q_\mathsf{E}) \\
\mathrm{WP}(\texttt{try } e_1 \texttt{ with E} \to e_2, Q, Q_\mathsf{E}) &\equiv \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_\mathsf{E}))
\end{aligned}
$$

Un autre let-in

Les exceptions peuvent transmettre des données :

$$
\begin{array}{rcll}
e & ::= & \ldots & \\
 & | & \texttt{raise E } t & \text{lever une exception} \\
 & | & \texttt{try } e \texttt{ with E } v \to e & \text{attraper une exception}
\end{array}
$$

Les mécanismes nécessaires sont déjà dans WP :

$$\mathrm{WP}(t, Q, Q_\mathsf{E}) \equiv Q[\texttt{result} \mapsto t]$$

$$\mathrm{WP}(\texttt{raise E } t, Q, Q_\mathsf{E}) \equiv Q_\mathsf{E}[\texttt{result} \mapsto t]$$

$$\mathrm{WP}(\texttt{let } v = e_1 \texttt{ in } e_2, Q, Q_\mathsf{E}) \equiv \mathrm{WP}(e_1, \mathrm{WP}(e_2, Q, Q_\mathsf{E})[v \mapsto \texttt{result}], Q_\mathsf{E})$$

$$\mathrm{WP}(\texttt{try } e_1 \texttt{ with E } v \to e_2, Q, Q_\mathsf{E}) \equiv \mathrm{WP}(e_1, Q, \mathrm{WP}(e_2, Q, Q_\mathsf{E})[v \mapsto \texttt{result}])$$

Les fonctions avec des exceptions

Une nouvelle clause de contrat :

$$\begin{aligned}&\texttt{let } f\ (v_1 : \tau_1) \ldots (v_n : \tau_n)\ : \varsigma\\ &\qquad \texttt{requires } P_f\\ &\qquad \texttt{writes } \vec{\boldsymbol{x}}\\ &\qquad \texttt{ensures } Q_f\\ &\qquad \texttt{raises E} \to Q_{\mathrm{E}f}\\ &= e\end{aligned}$$

La condition de vérification pour la définition de fonction :

$$\mathrm{VC}(\texttt{let } f \ldots) \quad \equiv \quad \forall \vec{x}\, \vec{v}.\, P_f \to \mathrm{WP}(e, Q_f, Q_{\mathrm{E}f})[\vec{x}^{\circ} \mapsto \vec{x}]$$

La règle de calcul de plus faible précondition :

$$\begin{aligned}\mathrm{WP}(f\ t_1 \ldots t_n, Q, Q_{\mathrm{E}}) \quad \equiv \quad & P_f[\vec{v} \mapsto \vec{t}] \wedge\\ & (\forall \vec{\boldsymbol{x}}\, \forall \texttt{result}.\, Q_f[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \to Q)[\vec{w} \mapsto \vec{x}] \wedge\\ & (\forall \vec{\boldsymbol{x}}\, \forall \texttt{result}.\, Q_{\mathrm{E}f}[\vec{v} \mapsto \vec{t}, \vec{x}^{\circ} \mapsto \vec{w}] \to Q_{\mathrm{E}})[\vec{w} \mapsto \vec{x}]\end{aligned}$$

5. Exercices

Exercice 1 — Fonction 91 de McCarthy

$$f_{91}(n) = \texttt{if } n \leq 100 \texttt{ then } f_{91}(f_{91}(n+11)) \texttt{ else } n-10$$

Trouver un contrat pour cette fonction

```
let rec f91 (n: int) : int
  requires { ??? }
  variant  { ??? }
  ensures  { ??? }
=
  if n <= 100 then
    f91 (f91 (n + 11))
  else
    n - 10
```

Exercice 2 — Recherche linéaire

1. Charger l'exemple « C2 - Ex2 » sur `http://why3.lri.fr/fiil-2017/`.
2. Compléter la spécification et l'implémentation.
3. Vérifier la fonction de recherche linéaire.

Exercice 3 — Recherche par dichotomie

1. Charger l'exemple « C2 - Ex3 » sur `http://why3.lri.fr/fiil-2017/`.
2. Compléter la spécification et l'implémentation.
3. Vérifier la fonction de recherche par dichotomie.