Une logique de séparation pour Cminor

Sandrine Blazy

ENSIIE/CEDRIC et INRIA Rocquencourt
collaboration avec Andrew Appel

Séminaire PPS, 14 XII 2006

Contexte
Certifier du logiciel C embarqué

- Compilateur certifié
- Preuve de programmes : logique de séparation

Évolutions : considérer d'autres langages que C (C concurrent, Java)

Langage intermédiaire + sémantique commune à définir

Automatisation en Coq

La vérification formelle de compilateurs

Compilation certifiée, [Leroy, POPL2006]

Appliquer les méthodes formelles au compilateur lui-même pour établir un théorème de préservation sémantique :

Théorème

Pour tous les codes source S,
si le compilateur transforme S en le code machine C,
sans signaler d'erreur de compilation,
et si S a une sémantique bien définie,
alors C a la même sémantique que S à équivalence observationnelle près.

Logique de séparation
O'Hearn, Brookes

- Extension de la logique de Hoare à des programmes manipulant des pointeurs
- Langage d'assertions facilitant la description de la mémoire

Raisonner localement sur la mémoire, tout en garantissant des propriétés globales.

Prouver des propriétés non assurées par le compilateur (*e.g.* aliasing, partitionnement).

Outline

Le langage Cminor

- Un langage intermédiaire du compilateur certifié.
- Petit langage impératif structuré en expressions, instructions et fonctions.
- Moins de constructions et de plus bas niveau que C.
  - Allocation en pile des variables locales
  - Accès mémoire : calculs d'adresses explicites
  - Une seule sorte de boucle (infinie)
- Seulement deux types : `int` (entiers & pointeurs) ; `float`.

Exemple de code Cminor

```
"square" (x): float -> float
  return x *f x;

"integr"(f, low, high, n): int -> float -> float -> int -> float
 var h, x, s, i;
  h = (high -f low) /f floatofint n;
  x = low;
  s = 0.0;
  i = n;
  block  loop
      if (! (i > 0)) exit;
      s = s +f (f(x): float -> float);
      x = x +f h;
      i = i - 1;
  return s *f h;

"test"(n) : int -> float
 return "integr"("square", 0.0, 1.0, n):
            int -> float -> float -> int -> float;
```

Modèle mémoire

- Un bloc de mémoire est composé d'un nombre quelconque de cellules élémentaires.
- Une valeur est lue ou écrite en mémoire connaissant sa taille, son type et son signe (*memory chunk*), qui déterminent le nombre de cellules élémentaires nécessaires pour accéder à la valeur.
  - Lecture $m \vdash v_1 \overset{ch}{\mapsto} v$
  - Écriture $[e_1]_{ch} := e_2$

Mémoire :

adresse d'un bloc $\rightarrow$ bornes $\times$ (décalage $\rightarrow$ chunk $\rightarrow$ valeur)

Traduction de C vers Cminor

Exemple 1

Résolution de la surcharge des opérateurs arithmétiques, insertion de coercions explicites.

```
i += f   --->  i = intoffloat (floatofint(i) +. f)
```

Accès mémoires $\rightarrow$ calculs d'adresses explicites + `load` ou `store` d'un *memory chunk*.

```
a[i]    --->      load(int32, a + i*4)
```

Traduction de C vers Cminor

Exemple 2

Transformation des boucles `while`, `dowhile`, `for` en boucles infinies + blocs + sorties prématurées.

```
while (e) {
    ...
    continue;
    ...
    break;
   ...
}
```

```
block {
  loop {
    if (! trad(e)) exit 0;
    block {
      ...
      exit 0;
      ...
      exit 1;
     ...
    }
  }
}
```

Traduction de C vers Cminor

Exemple 3

Les variables locales Cminor ne sont pas allouées en mémoire.
$\Rightarrow$ allocation explicite en pile des variables tableaux et scalaires dont on prend l'adresse.

```
{
  int i;
  int j;
  f (&i);
  i = ...;
  ... = ... i ...;
  j = ...;
  ... = ... j ...;
}
```

```
{
  stack 4;
  var j;
  f (stackaddr(0));
  store (int32, stack(0), ...);
  ... = ... load (int32, stack(0)) ...;
  j = ...;
  ... = ... j ...;
  }
```

Jugements d'évaluation
Expressions

Sémantique à grands pas

- $\sigma \vdash e \Downarrow v$ (expressions)
- $\sigma \vdash \vec{e} \Downarrow \vec{v}$ (liste d'expressions)

Jugements d'évaluation
Expressions

Sémantique à grands pas

- $\sigma \vdash e \Downarrow v$ (expressions)
- $\sigma \vdash \vec{e} \Downarrow \vec{v}$ (liste d'expressions)

$$\textit{Valeurs} \begin{cases} \texttt{undef} \\ \text{entier 32 bits} \\ \text{flottant 64 bits} \\ \text{pointeur} = (\text{adresse de bloc, décalage en octets}) \end{cases}$$

Représenté par une fonction Coq.

Jugements d'évaluation
Expressions

$\sigma \vdash e \Downarrow v \qquad \sigma : state = (\Psi; sp; \rho; \phi; m)$

Ψ	environnement global	identificateur $\longrightarrow$ adresse bloc $\times$ adresse fonction $\longrightarrow$ corps fonction
sp	pointeur de pile	
ρ	environnement local	identificateur $\longrightarrow$ adresse bloc
ϕ	empreinte mémoire	adresse bloc $\longrightarrow$ permission
m	mémoire	adresse bloc $\longrightarrow$ bornes $\times$ (décalage $\longrightarrow$ chunk $\longrightarrow$ valeur)

Permissions de lecture et d'écriture
[Bornat, POPL2005]

$v_1 \in^{ch}_{\text{load}} \phi$ Permission de lire les adresses $\in [v_1; v_1 + |ch| - 1]$.
$v_1 \in^{ch}_{\text{store}} \phi$ Permission d'écrire aux adresses $\in [v_1; v_1 + |ch| - 1]$.

Union disjointe $\phi = \phi_0 \oplus \phi_1$

$$v \in^{ch}_{\text{store}} \phi_0 \Rightarrow v \notin^{ch}_{\text{load}} \phi_1$$
$$v \in^{ch}_{\text{load}} \phi_0 \Rightarrow v \in^{ch}_{\text{load}} \phi$$
$$v \in^{ch}_{\text{store}} \phi_0 \Rightarrow v \in^{ch}_{\text{store}} \phi$$

Règles d'évaluation des expressions

$$\sigma \vdash \mathsf{Eval}\, v \Downarrow v \qquad \frac{x \in \mathrm{dom}\, \rho_\sigma}{\sigma \vdash \mathsf{Evar}\, x \Downarrow \rho_\sigma(x)}$$

$$\frac{\sigma \vdash el \Downarrow vl \qquad \Psi_\sigma; sp_\sigma \vdash op(vl) \Downarrow_{\mathsf{eval\_operation}} v}{\sigma \vdash \mathsf{Eop}\, op\, el \Downarrow v}$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \qquad v_1 \in^{ch}_{\mathsf{load}} \phi \qquad m \vdash v_1 \overset{ch}{\mapsto} v}{\sigma \vdash \mathsf{Eload}\, ch\, e_1 \Downarrow v}$$

Jugements d'évaluation
Instructions

Sémantique à petits pas

$k \longmapsto k'$ $\quad$ $k \longmapsto^* k'$ $\quad$ $k \longmapsto^n k'$

$$k : \text{continuation} ::= (\sigma, \kappa)$$
$$\kappa : \text{control} ::= \text{Kstop} \mid s \cdot \kappa \mid \text{Kblock}\,\kappa \mid \text{Kcall}\ xl\ f\ sp\ \rho\ \kappa$$

$$\textit{Instructions} \begin{cases} x := e & [e_1]_{ch} := e_2 \\ \text{loop}\ s & \text{if}\ e\ \text{then}\ s_1\ \text{else}\ s_2 \\ \text{block}\ s & \text{exit}\ n \\ \text{call}\ xl\ e\ el & \text{return}\ el \\ s_1 ; s_2 & \text{skip} \end{cases}$$

Sémantique dynamique
Instructions élémentaires

Écriture fonctionnelle en Coq (style monadique).

$$(\sigma, \mathsf{skip} \cdot \kappa) \longmapsto (\sigma, \kappa) \qquad (\sigma, (s_1; s_2) \cdot \kappa) \longmapsto (\sigma, s_1 \cdot s_2 \cdot \kappa)$$

$$\frac{\sigma \vdash e \Downarrow v \quad \rho' = \rho_\sigma[x := v]}{(\sigma, (x := e) \cdot \kappa) \longmapsto (\sigma[:= \rho'], \kappa)}$$

$$\frac{\sigma \vdash e_1 \Downarrow v_1 \qquad \sigma \vdash e_2 \Downarrow v_2 \qquad v_1 \in^{ch}_{\mathsf{store}} \phi_\sigma \qquad m' = m_\sigma[v_1 \overset{ch}{:=} v_2]}{(\sigma, ([e_1]_{ch} := e_2) \cdot \kappa) \longmapsto (\sigma[:= m'], \kappa)}$$

Sémantique dynamique
Autres instructions

$$\frac{\sigma \vdash e \Downarrow v \qquad \text{is\_true } v}{(\sigma,\ (\text{if } e \text{ then } s_1 \text{ else } s_2) \cdot \kappa) \longmapsto (\sigma, s_1 \cdot \kappa)}$$

$$\frac{\sigma \vdash e \Downarrow v \qquad \text{is\_false } v}{(\sigma,\ (\text{if } e \text{ then } s_1 \text{ else } s_2) \cdot \kappa) \longmapsto (\sigma, s_2 \cdot \kappa)}$$

$$(\sigma,\ (\text{loop } s) \cdot \kappa) \longmapsto (\sigma, s \cdot \text{loop } s \cdot \kappa)$$

$$(\sigma, (\text{block } s) \cdot \kappa) \longmapsto (\sigma, s \cdot \text{Kblock } \kappa)$$

$$(\sigma, \text{exit } 0 \cdot s_0 \cdot \cdots \cdot s_j \cdot \text{Kblock } \kappa) \longmapsto (\sigma, \kappa) \qquad \text{pour } j \geq 0$$

$$(\sigma, \text{exit } (n+1) \cdot s_0 \cdot \cdots \cdot s_j \cdot \text{Kblock } \kappa) \longmapsto (\sigma, \text{exit } n \cdot \kappa) \qquad \text{pour } j \geq 0$$

Appel de fonctions

$$\frac{\begin{array}{c}\sigma \vdash e_{\text{fun}} \Downarrow v_{\text{fun}} \qquad \sigma \vdash el_{\text{args}} \Downarrow vl_{\text{args}} \\ \Psi_\sigma(v_{\text{fun}}) = f \\ \text{alloc}(m_\sigma, \text{stackspace}(f)) = (m', sp') \\ \rho' = [\text{params}(f) \mapsto vl_{\text{args}}][\text{locals}(f) \mapsto \text{Vundef}] \\ \phi' = \phi_\sigma \oplus [sp', sp' + \text{stackspace}(f)) \\ \sigma' = \sigma[:= sp', \rho', \phi', m']\end{array}}{(\sigma,\ \text{call}\ xl\ e_{\text{fun}}\ el_{\text{args}} \cdot \kappa) \longmapsto (\sigma', \text{body}(f) \cdot \text{Kcall}\ xl\ f\ sp_\sigma\ \rho_\sigma\ \kappa)}$$

Appel de fonctions

$$\frac{\begin{array}{c}\sigma \vdash e_{\text{fun}} \Downarrow v_{\text{fun}} \qquad \sigma \vdash el_{\text{args}} \Downarrow vl_{\text{args}} \\ \Psi_\sigma(v_{\text{fun}}) = f \\ \text{alloc}(m_\sigma, \text{stackspace}(f)) = (m', sp') \\ \rho' = [\text{params}(f) \mapsto vl_{\text{args}}][\text{locals}(f) \mapsto \text{Vundef}] \\ \phi' = \phi_\sigma \oplus [sp', sp' + \text{stackspace}(f)) \\ \sigma' = \sigma[:= sp', \rho', \phi', m']\end{array}}{(\sigma,\ \text{call}\ xl\ e_{\text{fun}}\ el_{\text{args}} \cdot \kappa) \longrightarrow (\sigma', \text{body}(f) \cdot \text{Kcall}\ xl\ f\ sp_\sigma\ \rho_\sigma\ \kappa)}$$

$$\frac{\begin{array}{c}\kappa = \textit{séquence de} \cdot \textit{et}\ \text{Kblock}\ \textit{finissant par}\ \text{Kcall}\ xl\ f\ sp'\ \rho'\ \kappa' \\ \sigma \vdash el_{\text{args}} \Downarrow vl_{\text{args}} \qquad |\text{params}(f)| = |vl_{\text{args}}| \\ \rho'' = \rho'[xl := vl_{\text{args}}] \qquad \phi' = \phi_\sigma \backslash [sp', sp' + \text{stackspace}(f)) \\ \text{free}(m_\sigma, sp_\sigma) = m'\end{array}}{(\sigma,\ \text{return}\ el_{\text{args}} \cdot \kappa) \longrightarrow (\sigma[:= sp', \rho'', \phi', m'], \kappa')}$$

Outline

Logique de séparation
Langage d'assertions

Observation plus fine du tas et de la pile

P, Q : assert avec assert $=$ state $\rightarrow$ Prop

$P\sigma$ est une proposition

$$\mathbf{emp} = \lambda\sigma.\ \phi = \emptyset \qquad \text{Tas vide}$$

$$P \vee Q = \lambda\sigma.\ P\sigma \ \vee\ Q\sigma$$

$$P \wedge Q = \lambda\sigma.\ P\sigma \ \wedge\ Q\sigma$$

$$P * Q = \lambda\sigma.\ \exists\phi_1, \phi_2.\ \ \phi_\sigma = \phi_1 \oplus \phi_2 \ \wedge\ P(\sigma[:= \phi_1]) \ \wedge\ Q(\sigma[:= \phi_2])$$

$$\neg P = \lambda\sigma.\ \neg(P\sigma)$$

$$\exists z.P = \lambda\sigma.\ \exists z.\ P\sigma$$

$$\lceil A \rceil = \lambda\sigma.\ A \qquad \text{avec } \sigma \text{ non libre dans } A : \text{Prop}$$

$$\mathbf{true} = \lceil \mathbf{True} \rceil$$

$$\mathbf{false} = \lceil \mathbf{False} \rceil$$

Langage d'assertions

Opérateurs, 2

Logique de séparation usuelle : pas de lecture en mémoire lors de l'évaluation d'une expression.
Ici :

- notion d'expression pure (sans opérateur de lecture en mémoire).
- une expression peut ne pas être pure.

$$e \Downarrow v \;=\; \mathbf{emp} \wedge \lceil \mathrm{pure}(e) \rceil \wedge \lambda\sigma.\,(\sigma \vdash e \Downarrow v) \qquad \text{e vaut v}$$

$$\lceil e \rceil_{\mathrm{expr}} = \exists v.\, e \Downarrow v \;*\; \lceil \mathrm{is\_true}\; v \rceil \qquad \text{e s'évalue en 'vrai'}$$

$$\lceil e \overset{\mathrm{float}}{==} e \rceil_{\mathrm{expr}} \qquad \text{e est un floattant défini dans l'état courant}$$

$$\mathrm{defined}(e) = \lceil e \overset{\mathrm{int}}{==} e \rceil_{\mathrm{expr}} \vee \lceil e \overset{\mathrm{float}}{==} e \rceil_{\mathrm{expr}}$$

$$e_1 \overset{ch}{\mapsto} e_2 = \exists v_1, v_2.(e_1 \Downarrow v_1) * (e_2 \Downarrow v_2) * (\lambda\sigma, m_\sigma \vdash v_1 \overset{ch}{\mapsto} v_2) * \mathrm{defined}(v_2)$$

Sémantique axiomatique
Environnements

Extension des triplets de Hoare pour prendre en compte les sorties abruptes de blocs et de fonctions.
$\Gamma; R; B \vdash \{P\}s\{Q\}$ avec :

- Γ : assert Propriétés de l'environnement global.
- R : list val $\rightarrow$ assert *Environnement pour les retours de fonctions*. $P_f(vl_{\text{args}})Q_f(vl_{\text{results}})$ représente une fonction.
- B : nat $\rightarrow$ assert *Environnement pour les sorties de blocs*, fournissant la condition de sortie de chaque bloc contenant s. $B(n)$: pré-condition relative à exit n

Sémantique axiomatique

Règles, 1

$$\Gamma; R; B \vdash \{P\}\mathsf{skip}\{P\}$$

$$\frac{\Gamma; R; B \vdash \{P\}s_1\{P'\} \qquad \Gamma; R; B \vdash \{P'\}s_2\{Q\}}{\Gamma; R; B \vdash \{P\}s_1; s_2\{Q\}}$$

$$\frac{\mathsf{pure}\,(e) \qquad \Gamma; R; B \vdash \{P \wedge e\}s_1\{Q\} \qquad \Gamma; R; B \vdash \{P \wedge \neg e\}s_2\{Q\}}{\Gamma; R; B \vdash \{P\}\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\{Q\}}$$

$$\frac{\begin{array}{c}x \text{ non libre dans } s_1, s_2, Q \\ \Gamma; R; B \vdash \{P\}x := e; \mathsf{if}\ x\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\{Q\}\end{array}}{\Gamma; R; B \vdash \{P\}\mathsf{if}\ e\ \mathsf{then}\ s_1\ \mathsf{else}\ s_2\{Q\}}$$

Sémantique axiomatique

Règles, 2

$$\frac{\Gamma; R; B \vdash \{I\} s \{I\}}{\Gamma; R; B \vdash \{I\} \text{loop } s \{\textbf{false}\}}$$

$$\frac{\rho' = \rho_\sigma[x := v] \qquad P = (\exists v.\, e \Downarrow v \ \wedge \ \lambda\sigma.\, Q\,\sigma[:= \rho'])}{\Gamma; R; B \vdash \{P\} x := e \{Q\}}$$

$$\frac{\text{pure}\,(e) \qquad \text{pure}\,(e_2) \qquad P = (e \overset{ch}{\mapsto} e_2 \ * \ e_1 \overset{ch}{==} e_1)}{\Gamma; R; B \vdash \{P\} [e]_{ch} := e_1 \{e \overset{ch}{\mapsto} e_1\}}$$

Sémantique axiomatique

Règles, 3

$$\Gamma; R; B \vdash \{B(n)\}\text{exit}\, n\{Q\}$$

$$\frac{\Gamma; R; Q \cdot B \vdash \{P\}s\{\textbf{false}\}}{\Gamma; R; B \vdash \{P\}\text{block}\, s\{Q\}}$$

$$\begin{aligned} \text{nil}_B &= \lambda n.\ \textbf{false} \\ P \cdot B &= \lambda n.\ \lambda \sigma.\ (n = 0 \Rightarrow P\sigma) \ \wedge\ (n > 0 \Rightarrow B(n-1)\sigma) \end{aligned}$$

$$\frac{P \Rightarrow P' \qquad \Gamma; R; B \vdash \{P'\}s\{Q'\} \qquad Q' \Rightarrow Q}{\Gamma; R; B \vdash \{P\}s\{Q\}}$$

Outline

Validation des sémantiques

3 sémantiques de Cminor ont été définies :

- sémantique à grands pas (compilateur certifié)
- sémantique à petits pas
- sémantique axiomatique (preuve de programmes)

La preuve de l'équivalence entre les sémantiques à grands et petits pas a permis de valider (et aussi de définir !) la sémantique à petits pas.

La sémantique à petits pas est abstraite par rapport à la sémantique axiomatique. Les règles de la sémantique axiomatique sont des lemmes ayant été prouvés à partir de la sémantique à petits pas.

Sémantique à petits pas

Definition

$k = (\sigma, \kappa)$ est *bloquée (stuck)* si $\kappa \neq$ Kstop et $\nexists k'.k \longmapsto^* k'$.

Definition

Une continuation est *sure* (notation $\mathrm{safe(k)}$) si elle ne permet pas d'atteindre une continuation bloquée par $\longmapsto^*$.

Definition

P *garde* κ (notation $P \,\square\, \kappa$) si $\forall \sigma.\, P\sigma \Rightarrow \mathrm{safe}(\sigma, \kappa)$.

Sémantique à petits pas et sémantique axiomatique

- $P \Box \kappa \;=\; \forall \sigma.\, P\sigma \Rightarrow \mathrm{safe}(\sigma, \kappa)$
- $R \,\boxed{\mathrm{R}}\, \kappa \;=\; \forall vl, R(vl) \Box \mathrm{return}\ vl \cdot \kappa$
- $B \,\boxed{\mathrm{B}}\, \kappa \;=\; \forall n, B(n) \Box \mathrm{exit}\ n \cdot \kappa$

D'où la définition :

$\Gamma; R; B \vdash \{P\}s\{Q\} \;=\; \forall \kappa.\; R \,\boxed{\mathrm{R}}\, \kappa \wedge B \,\boxed{\mathrm{B}}\, \kappa \wedge Q \Box \kappa \;\Rightarrow\; P \Box s \cdot \kappa$

Validation de la sémantique axiomatique

Soundness - exemple de lemmes auxiliaires

If $\sigma \vdash e \Downarrow v$ and $\phi_\sigma \subset \phi'$ then $\sigma[:= \phi'] \vdash e \Downarrow v$.

If $P \,\square\, s_1 \cdot s_2 \cdot \kappa$ then $P \,\square\, (s_1 ; s_2) \cdot \kappa$.

If $R \,\mathbb{R}\, \kappa$ then $\forall s, R \,\mathbb{R}\, s \cdot \kappa$.

Validation de la sémantique axiomatique
Soundness (boucle)

La preuve la plus difficile est celle de la règle de la boucle. Elle nécessite de compter le nombre de pas d'exécution.

Soient $s = \text{if } b \text{ then exit } 2 \text{ else (skip; } x := y)$ et $(\sigma, s \cdot \kappa) \longmapsto^n (\sigma', \kappa')$.

- Pour n petit, κ reste inchangé.
- Ceci change si s est exécutée en moins de n pas.

Si $\rho_\sigma b$ vaut vrai, alors pour $n \leq 1$, s peut *absorber* n pas indépendamment de κ ;
sinon s peut absorber jusqu'à trois pas.

Validation de la sémantique axiomatique
Soundness (boucle)

Étant donné un état σ, une instruction *s* absorbe *n* pas si
$\forall j \leq n.\, \exists \kappa_{\text{prefix}} \exists \sigma'.\, \forall \kappa. (\sigma, s \cdot \kappa) \longmapsto^{j} (\sigma', \kappa_{\text{prefix}} \circ \kappa)$

$\circ$ opérateur de concaténation défini par :

$$\begin{aligned}
\text{Kstop} \circ \kappa &=_{\text{def}} \kappa \\
\text{Kblock}\, \kappa' \circ \kappa &=_{\text{def}} \text{Kblock}\, (\kappa' \circ \kappa) \\
s \cdot \kappa' \circ \kappa &=_{\text{def}} s \cdot (\kappa' \circ \kappa) \\
\text{Kcall}\, xl\ f\ sp\, \rho\, \kappa' \circ \kappa &=_{\text{def}} \text{Kcall}\, xl\ f\ sp\, \rho\, (\kappa' \circ \kappa)
\end{aligned}$$

Ex. Une instruction exit n'absorbe aucun pas, mais block (exit 0) peut absorber 2 pas :
$(\sigma, \text{block}\, (\text{exit}\, 0) \cdot \kappa) \longmapsto (\sigma, \text{exit}\, 0 \cdot \text{Kblock}\, \kappa) \longmapsto (\sigma, \kappa)$

Validation de la sémantique axiomatique
Soundness (boucle)

Propriétés

1. absorb$(0, s, \sigma)$.
2. absorb$(n+1, s, \sigma) \Rightarrow$ absorb(n, s, σ).
3. If $\neg$absorb(n, s, σ), then
   $\exists i < n$, absorb$(i, s, \sigma) \wedge \neg$absorb$(i+1, s, \sigma)$.
   s absorbe au plus i pas dans l'état σ.

Validation de la sémantique axiomatique
Soundness (loop)

$$(s\cdot)^n s' = \underbrace{s \cdot s \cdot \ldots \cdot s \cdot}_{n} s'$$

$$\frac{\Gamma; R; B \vdash \{I\} s \{I\}}{\Gamma; R; B \vdash \{I\} (s\cdot)^n \mathsf{loop}\,\mathsf{skip} \{\mathbf{false}\}}$$

$$\frac{\Gamma; R; B \vdash \{I\} s \{I\}}{\Gamma; R; B \vdash \{I\} \mathsf{loop}\, s \{\mathbf{false}\}}$$

Validation de la sémantique à petits pas

Équivalence petits pas/ grands pas pour les programmes qui terminent

Définition d'une sémantique à grands pas avec continuations (liens entre outcomes et continuations).

$$\frac{}{(\sigma, \mathsf{Kstop}); \mathit{out} \rightsquigarrow \sigma; \mathit{out}}$$

$$\frac{\vdash (\sigma, s) \Downarrow_{\mathit{out}_1} \sigma_1 \qquad \vdash (\sigma_1, \kappa); \mathit{out}_1 \rightsquigarrow \sigma'; \mathit{out}'}{(\sigma, (s \cdot \kappa)); \mathsf{Out}_{\mathsf{normal}} \rightsquigarrow \sigma'; \mathit{out}'}$$

$$\frac{\vdash (\sigma, s) \Downarrow_{\mathit{out}_1} \sigma_1 \qquad \vdash (\sigma_1, \kappa); \mathit{out}_1 \rightsquigarrow \sigma_2; \mathit{out}_2}{(\sigma, s \cdot \kappa) \Downarrow_{\mathit{out}_2} \sigma_2}$$

Validation de la sémantique axiomatique

Équivalence petits pas/ grands pas pour les programmes qui terminent

If $\vdash (\sigma, s) \Downarrow_{out} \sigma'$, then for all κ and κ', $\vdash \ (\sigma, s \cdot \kappa) \longmapsto^* (\sigma', s' \cdot \kappa)$ for $s' = \text{stmt\_of\_outcome}(out)$.

If $(\sigma, s \cdot \kappa) \longmapsto^* (\sigma', s' \cdot \kappa)$ then $\vdash (\sigma, s) \Downarrow_{out} \sigma'$
for $out = \text{outcome\_of\_stmt}(s')$.

If $(\sigma, s \cdot \kappa) \longmapsto (\sigma_1, s_1 \cdot \kappa_1)$ and $\vdash (\sigma_1, s_1 \cdot \kappa_1) \Downarrow_{out} \sigma'$ then
$\vdash (\sigma, s \cdot \kappa) \Downarrow_{out} \sigma'$.

If $\vdash (\sigma, \text{exit}\, n \cdot \kappa) \Downarrow_{out} \sigma'$ then $\vdash (\sigma, \text{exit}\, (n+1) \cdot (\text{Kblock}\, \kappa)) \Downarrow_{out} \sigma'$.

Conclusion

- Raisonner en Coq sur une sémantique à petits pas est possible.
- Difficulté : mélange de structures de contrôle non locales (exit, return) avec des boucles.

Perspectives

- Intérêt d'une sémantique à petits pas pour la preuve de compilateurs ?
  Petis pas ou coinductif ([Leroy, ESOP 2006]) ?
- Intérêt d'une sémantique axiomatique pour la preuve de compilateurs ?
- Définir d'autres tactiques pour faciliter les preuves faites sur les assertions.
- Traiter le langage C.
- Définir un langage concurrent (Concurrent C minor, `http://www.cs.princeton.edu/~appel/cminor`).