

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

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

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

Outline

- 1 Le langage Cminor
- 2 Logique de séparation
- 3 Preuves en Coq

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 $m \vdash v_1 \xrightarrow{ch} v$
 - ▶ 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 → 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.
⇒ allocation explicite en pile des variables tableaux et scalaires dont on prend l'adresse.

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

Valeurs {
undef
entier 32 bits
flottant 64 bits
pointeur = (adresse de bloc, décalage en octets)

Représenté par une fonction Coq.

Jugements d'évaluation

Expressions

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

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

Permissions de lecture et d'écriture

[Bornat, POPL2005]

$v_1 \in_{\text{load}}^{ch} \phi$ Permission de lire les adresses $\in [v_1; v_1 + |ch| - 1]$.

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

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

$$v \in_{\text{store}}^{ch} \phi_0 \Rightarrow v \notin_{\text{load}}^{ch} \phi_1$$

$$v \in_{\text{load}}^{ch} \phi_0 \Rightarrow v \in_{\text{load}}^{ch} \phi$$

$$v \in_{\text{store}}^{ch} \phi_0 \Rightarrow v \in_{\text{store}}^{ch} \phi$$

Règles d'évaluation des expressions

$$\begin{array}{c} \sigma \vdash \text{Eval } v \Downarrow v \qquad \frac{x \in \text{dom } \rho_\sigma}{\sigma \vdash \text{Evar } x \Downarrow \rho_\sigma(x)} \\ \\ \frac{\sigma \vdash e_l \Downarrow v_l \quad \Psi_\sigma; sp_\sigma \vdash op(v_l) \Downarrow_{\text{eval_operation}} v}{\sigma \vdash \text{Eop } op \ e_l \Downarrow v} \\ \\ \frac{\sigma \vdash e_1 \Downarrow v_1 \quad v_1 \in_{\text{load}}^{ch} \phi \quad m \vdash v_1 \xrightarrow{ch} v}{\sigma \vdash \text{Eload } ch \ e_1 \Downarrow v} \end{array}$$

Jugements d'évaluation

Instructions

Sémantique à petits pas

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

k : continuation ::= (σ, κ)

κ : control ::= $\text{Kstop} \mid s \cdot \kappa \mid \text{Kblock } \kappa \mid \text{Kcall } x \mid f \text{ sp } \rho \kappa$

$$\text{Instructions} \left\{ \begin{array}{ll} x := e & [e_1]_{ch} := e_2 \\ \text{loop } s & \text{if } e \text{ then } s_1 \text{ else } s_2 \\ \text{block } s & \text{exit } n \\ \text{call } x \mid e \text{ el} & \text{return } e \mid \\ s_1; s_2 & \text{skip} \end{array} \right.$$

Écriture fonctionnelle en Coq (style monadique).

$$(\sigma, \text{skip} \cdot \kappa) \mapsto (\sigma, \kappa) \quad (\sigma, (\mathbf{s}_1; \mathbf{s}_2) \cdot \kappa) \mapsto (\sigma, \mathbf{s}_1 \cdot \mathbf{s}_2 \cdot \kappa)$$

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

$$\frac{\sigma \vdash \mathbf{e}_1 \Downarrow \mathbf{v}_1 \quad \sigma \vdash \mathbf{e}_2 \Downarrow \mathbf{v}_2 \quad \mathbf{v}_1 \in_{\text{store}}^{ch} \phi_\sigma \quad m' = m_\sigma[\mathbf{v}_1 \stackrel{ch}{:=} \mathbf{v}_2]}{(\sigma, ([\mathbf{e}_1]_{ch} := \mathbf{e}_2) \cdot \kappa) \mapsto (\sigma[:=m'], \kappa)}$$

Sémantique dynamique

Autres instructions

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

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

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

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

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

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

Appel de fonctions

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

Appel de fonctions

$$\begin{array}{c} \sigma \vdash \mathbf{efun} \Downarrow \mathbf{vfun} \qquad \sigma \vdash \mathbf{elargs} \Downarrow \mathbf{vlargs} \\ \Psi_\sigma(\mathbf{vfun}) = f \\ \text{alloc}(m_\sigma, \text{stackspace}(f)) = (m', sp') \\ \rho' = [\text{params}(f) \mapsto \mathbf{vlargs}][\text{locals}(f) \mapsto \mathbf{Vundef}] \\ \phi' = \phi_\sigma \oplus [sp', sp' + \text{stackspace}(f)] \\ \sigma' = \sigma[:= sp', \rho', \phi', m'] \end{array}$$

$$(\sigma, \text{call } x! \mathbf{efun} \mathbf{elargs} \cdot \kappa) \longmapsto (\sigma', \text{body}(f) \cdot \text{Kcall } x! f \mathbf{sp}_\sigma \rho_\sigma \kappa)$$

$\kappa =$ séquence de \cdot et Kblock finissant par Kcall $x! f \mathbf{sp}' \rho' \kappa'$

$$\begin{array}{c} \sigma \vdash \mathbf{elargs} \Downarrow \mathbf{vlargs} \qquad |\text{params}(f)| = |\mathbf{vlargs}| \\ \rho'' = \rho' [x! := \mathbf{vlargs}] \qquad \phi' = \phi_\sigma \setminus [sp', sp' + \text{stackspace}(f)] \\ \text{free}(m_\sigma, \mathbf{sp}_\sigma) = m' \end{array}$$

$$(\sigma, \text{return } \mathbf{elargs} \cdot \kappa) \longmapsto (\sigma[:= sp', \rho'', \phi', m'], \kappa')$$

Outline

1 Le langage Cminor

2 Logique de séparation

3 Preuves en Coq

Logique de séparation

Langage d'assertions

Observation plus fine du tas et de la pile

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

$P\sigma$ est une proposition

emp = $\lambda\sigma. \phi = \emptyset$ Tas vide

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

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

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

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

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

$\lceil A \rceil = \lambda\sigma. A$ avec σ non libre dans A : Prop

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

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

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.

$$e \Downarrow v = \mathbf{emp} \wedge [\mathbf{pure}(e)] \wedge \lambda\sigma. (\sigma \vdash e \Downarrow v) \quad e \text{ vaut } v$$

$$[e]_{\text{expr}} = \exists v. e \Downarrow v * [\mathbf{is_true } v] \quad e \text{ s'évalue en 'vrai'}$$

$$[e \stackrel{\text{float}}{==} e]_{\text{expr}} \quad e \text{ est un flottant défini dans l'état courant}$$

$$\mathbf{defined}(e) = [e \stackrel{\text{int}}{==} e]_{\text{expr}} \vee [e \stackrel{\text{float}}{==} e]_{\text{expr}}$$

$$e_1 \xrightarrow{\text{ch}} e_2 = \exists v_1, v_2. (e_1 \Downarrow v_1) * (e_2 \Downarrow v_2) * (\lambda\sigma, m_\sigma \vdash v_1 \xrightarrow{\text{ch}} v_2) * \mathbf{defined}(v_2)$$

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.*
 $P_f(vl_{args})Q_f(vl_{results})$ 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

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

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

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

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

Sémantique axiomatique

Règles, 2

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

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

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

Sémantique axiomatique

Règles, 3

$$\Gamma; R; B \vdash \{B(n)\} \text{exit } n\{Q\} \qquad \frac{\Gamma; R; Q \cdot B \vdash \{P\} s\{\mathbf{false}\}}{\Gamma; R; B \vdash \{P\} \text{block } s\{Q\}}$$

$\text{nil}_B = \lambda n. \mathbf{false}$

$P \cdot B = \lambda n. \lambda \sigma. (n = 0 \Rightarrow P\sigma) \wedge (n > 0 \Rightarrow B(n-1)\sigma)$

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

Outline

- 1 Le langage Cminor
- 2 Logique de séparation
- 3 Preuves en Coq**

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 $\kappa \neq \text{Kstop}$ et $\nexists k'. k \mapsto^* k'$.

Definition

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

Definition

P garde κ (notation $P \sqsubseteq \kappa$) si $\forall \sigma. P\sigma \Rightarrow \text{safe}(\sigma, \kappa)$.

Sémantique à petits pas et sémantique axiomatique

- $P \sqsubseteq \kappa = \forall \sigma. P\sigma \Rightarrow \text{safe}(\sigma, \kappa)$
- $R \boxplus \kappa = \forall vl, R(vl) \sqsubseteq \text{return } vl \cdot \kappa$
- $B \boxplus \kappa = \forall n, B(n) \sqsubseteq \text{exit } n \cdot \kappa$

D'où la définition :

$$\Gamma; R; B \vdash \{P\} s \{Q\} = \forall \kappa. R \boxplus \kappa \wedge B \boxplus \kappa \wedge Q \sqsubseteq \kappa \Rightarrow P \sqsubseteq s \cdot \kappa$$

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 $P \sqsubseteq s_1 \cdot s_2 \cdot \kappa$ then $P \sqsubseteq (s_1; s_2) \cdot \kappa$.

If $R \sqsubseteq \kappa$ then $\forall s, R \sqsubseteq s \cdot \kappa$.

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

- 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

$$\forall j \leq n. \exists \kappa_{\text{prefix}} \exists \sigma'. \forall \kappa. (\sigma, s \cdot \kappa) \mapsto^j (\sigma', \kappa_{\text{prefix}} \circ \kappa)$$

○ opérateur de concaténation défini par :

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

Ex. Une instruction `exit` n'absorbe aucun pas, mais `block (exit 0)` peut absorber 2 pas :

$$(\sigma, \text{block } (\text{exit } 0) \cdot \kappa) \mapsto (\sigma, \text{exit } 0 \cdot \text{Kblock } \kappa) \mapsto (\sigma, \kappa)$$

Validation de la sémantique axiomatique

Soundness (boucle)

Propriétés

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

Validation de la sémantique axiomatique

Soundness (loop)

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

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

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

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).

$$\frac{}{(\sigma, \text{Kstop}); \text{out} \rightsquigarrow \sigma; \text{out}}$$
$$\frac{\vdash (\sigma, \mathbf{s}) \Downarrow_{\text{out}_1} \sigma_1 \quad \vdash (\sigma_1, \kappa); \text{out}_1 \rightsquigarrow \sigma'; \text{out}'}{(\sigma, (\mathbf{s} \cdot \kappa)); \text{Out}_{\text{normal}} \rightsquigarrow \sigma'; \text{out}'}$$
$$\frac{\vdash (\sigma, \mathbf{s}) \Downarrow_{\text{out}_1} \sigma_1 \quad \vdash (\sigma_1, \kappa); \text{out}_1 \rightsquigarrow \sigma_2; \text{out}_2}{(\sigma, \mathbf{s} \cdot \kappa) \Downarrow_{\text{out}_2} \sigma_2}$$

Validation de la sémantique axiomatique

Équivalence petits pas/ grands pas pour les programmes qui terminent

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

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

If $(\sigma, \mathbf{s} \cdot \kappa) \longmapsto (\sigma_1, \mathbf{s}_1 \cdot \kappa_1)$ and $\vdash (\sigma_1, \mathbf{s}_1 \cdot \kappa_1) \Downarrow_{out} \sigma'$ then
 $\vdash (\sigma, \mathbf{s} \cdot \kappa) \Downarrow_{out} \sigma'$.

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>).