

Un parcours de la compilation certifiée à la preuve de programmes

Les sémantiques formelles de Cminor et
le modèle mémoire du compilateur CompCert

Sandrine Blazy

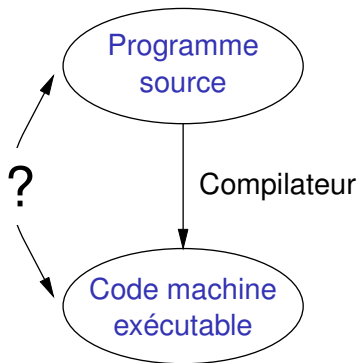
ENSIIE, laboratoire CEDRIC

séminaire du LACL, 8 décembre 2008

ensiie

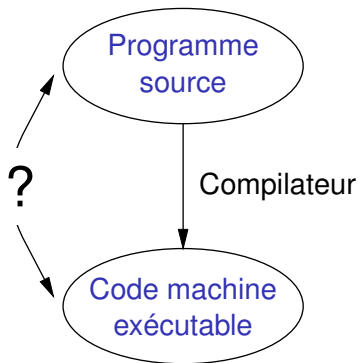


Faites-vous confiance à votre compilateur ?



Un bug dans le compilateur peut faire produire du code machine faux à partir d'un programme correct.

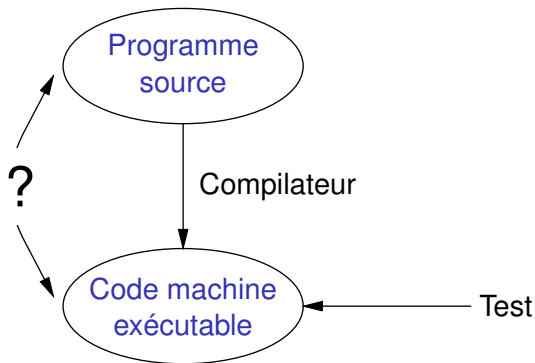
Faites-vous confiance à votre compilateur ?



Logiciel non critique :

Les bugs du compilateur sont négligeables devant ceux du logiciel.

Faites-vous confiance à votre compilateur ?

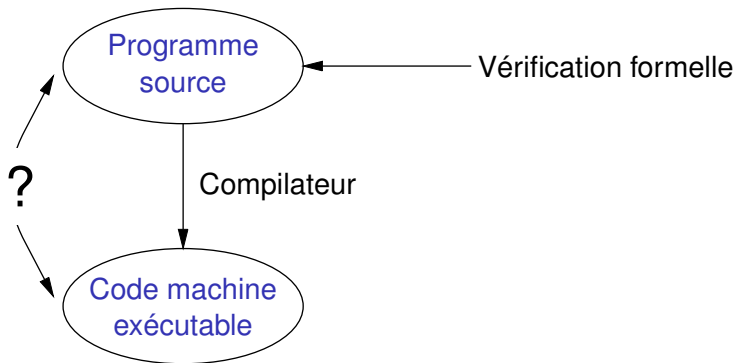


Logiciel critique certifié par test systématique :

Ce qui est testé : le code exécutable produit par le compilateur.

Les bugs du compilateur sont détectés en même temps que ceux du programme.

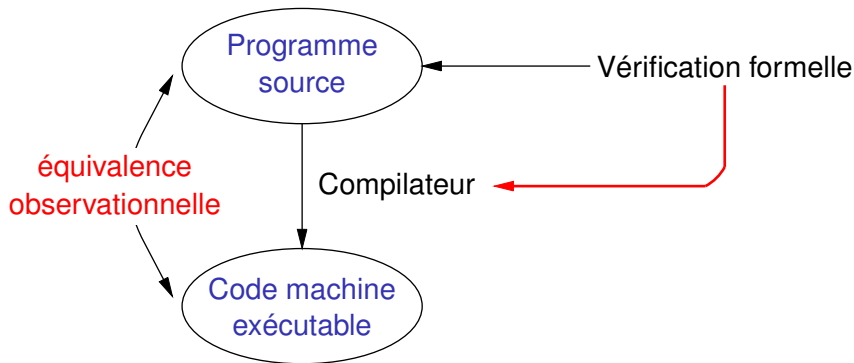
Faites-vous confiance à votre compilateur ?



Logiciel critique certifié par méthodes formelles :

Ce qui est prouvé est le code source, pas le code exécutable.
Les bugs du compilateur peuvent invalider l'approche.

Faites-vous confiance à votre compilateur ?



Compilateur formellement vérifié :

Garantit que le code produit se comporte comme prescrit par la sémantique du programme source.

La vérification formelle de compilateurs

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 le même comportement observable que S .*

Remarque : la compilation peut échouer.

Le compilateur certifié CompCert

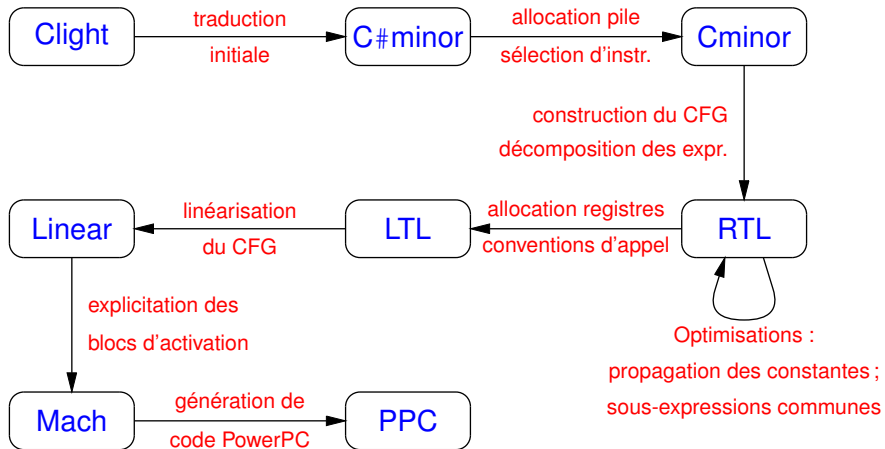
`compcert.inria.fr`

2003-2008 : ARC Concert, puis
ANR SSIA CompCert coordonné par Xavier Leroy

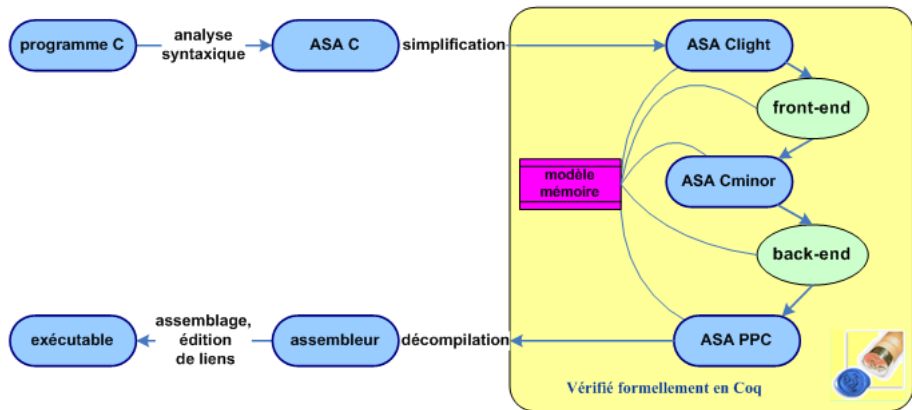
Compilateur réaliste utilisable pour le logiciel embarqué critique.

- Langage source = vaste sous-ensemble de C.
- Langage cible = assembleur du processeur PowerPC.
- Produit du code raisonnablement compact et efficace
⇒ optimisations.
- Compilateur formellement vérifié en Coq.

La partie formellement vérifiée du compilateur



Le compilateur CompCert au complet



Le sous-ensemble de C traité

Traité :

- Types : entiers, flottants, tableaux, pointeurs, `struct`, `union`.
- Opérateurs : toute l'arithmétique de C, y compris sur pointeurs.
- Contrôle structuré : `if/then/else`, boucles, `switch` réguliers.
- Fonctions, fonctions récursives, pointeurs vers fonctions.

Non traités :

- Les types `long long` et `long double`.
- Le `goto`, les `switch` irréguliers.
- Les fonctions à nombre variable d'arguments.
- Passage de `struct` et `union` par valeur.

Traités par expansion après parsing (CIL) :

- Effets de bord à l'intérieur des expressions.
- Variables locales à un bloc.

Formellement vérifié en Coq

La preuve de correction (préservation de la sémantique) du compilateur est entièrement formalisée sur machine, à l'aide de l'assistant de preuve Coq. (48000 lignes de Coq.)

```
Theorem transf_c_program_correct :  
  forall prog tprog behavior,  
    transf_c_program prog = OK tprog ->  
    Clight.exec_program prog behavior ->  
    PPC.exec_program tprog behavior.
```

Les comportements observables sont

- Terminaison, avec une trace finie d'événements d'entrée-sortie (appels systèmes) et l'entier renvoyé par la fonction `main`.
- Divergence, avec une trace finie ou infinie d'événements d'entrée-sortie.

Programmé en Coq

Toutes les parties vérifiées du compilateur sont programmées directement dans le langage de spécification de Coq, en style fonctionnel pur.

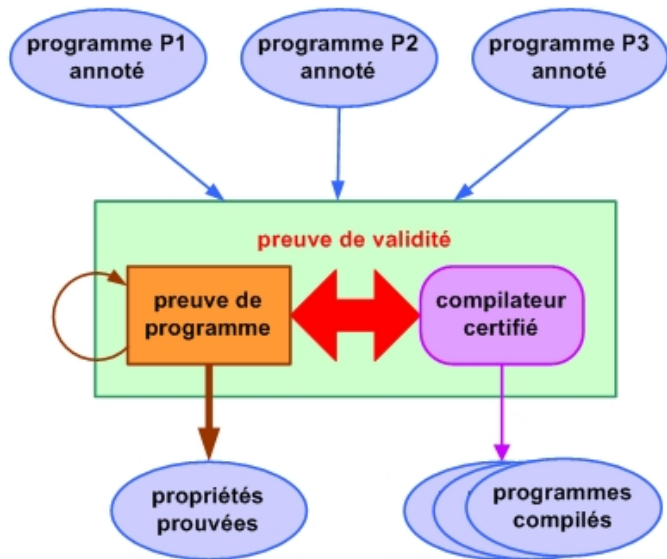
- Utilisation de monades pour traiter les erreurs et les états.
- Structures de données purement fonctionnelles (persistantes).

(6000 lignes de Coq + 2000 lignes de Caml non vérifié.)

Le mécanisme d'extraction de Coq produit du code Caml exécutable depuis ces spécifications.

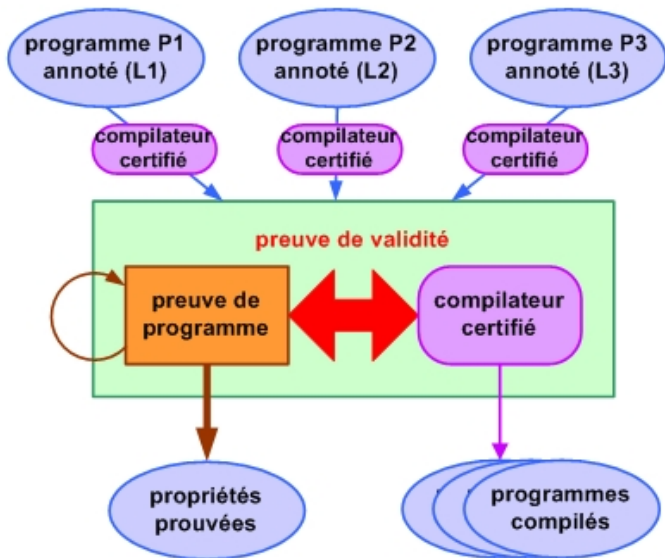
Un environnement pour la preuve de programmes

projet Concurrent Cminor, A. Appel, www.cs.princeton.edu/~appel/cminor

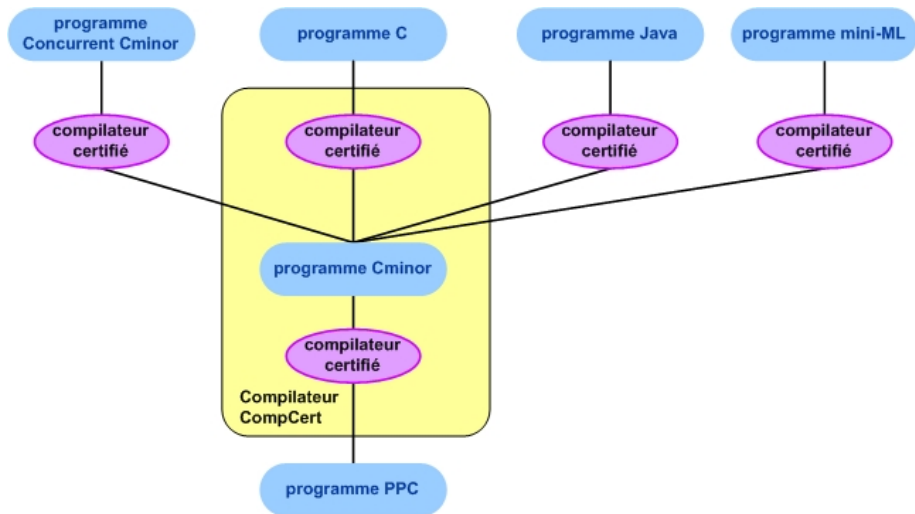


Un environnement pour la preuve de programmes

projet Concurrent Cminor, A. Appel, www.cs.princeton.edu/~appel/cminor



Le langage intermédiaire Cminor



Plan

- 1 Les sémantiques de Cminor
- 2 Un modèle mémoire pour un compilateur du langage C
- 3 Conclusions et perspectives

Le langage Cminor

Langage proche de C, mais de plus bas niveau que C :

- pas de surcharge des opérateurs arithmétiques
- calculs d'adresses explicites
- allocation explicite en pile de certaines variables locales
- instructions de plus bas niveau

Exemple : placement en pile de variables locales

```
{                                     {  
  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 ... ;  
}                                     }
```

Le langage Cminor

Deuxième exemple : Traduction d'une boucle

```
for ( $S_1$ ;  $e$ ;  $S_2$ ) {  
  
    ...  
    exit;  
    ...  
    continue;  
    ...  
  
}
```

```
 $S_1$ ;  
block { loop {  
    if (! $e$ ) exit 0;  
    block {  
        ...  
        exit 1;  
        ...  
        exit 0;  
        ...  
    }  
     $S_2$ ;  
} }
```

Infrastructure pour décrire les sémantiques

Valeurs : $v ::= \text{int}(n) \mid \text{float}(f) \mid \text{ptr}(b, ofs) \mid \text{undef}$

Environnement local : $E ::= id \mapsto v$

Environnement global : $G ::= (id \mapsto b) \times (b \mapsto def_fn)$

Mémoire : $M ::= b \mapsto (lo, hi, ofs \mapsto v)$

Sortie d'instruction :

<i>out</i>	$::=$	Normal	(aller à la prochaine instruction)
		Exit(<i>n</i>)	(sortie du bloc de niveau <i>n</i> + 1)
		Return	(sortie de fonction)
		Return(<i>v</i>)	(sortie de fonction, avec valeur <i>v</i>)

Infrastructure pour décrire les sémantiques

Valeurs : $v ::= \text{int}(n) \mid \text{float}(f) \mid \text{ptr}(b, ofs) \mid \text{undef}$

Environnement local : $E ::= id \mapsto v$

Environnement global : $G ::= (id \mapsto b) \times (b \mapsto \text{def_fn})$

Mémoire : $M ::= b \mapsto (lo, hi, ofs \mapsto v)$

Sortie d'instruction :

$out ::=$	$Normal$	(aller à la prochaine instruction)
	$ Exit(n)$	(sortie du bloc de niveau $n + 1$)
	$ Return$	(sortie de fonction)
	$ Return(v)$	(sortie de fonction, avec valeur v)

Traces :

$v_\nu ::=$	$\text{int}(n) \mid \text{float}(f)$	valeur d'entrée-sortie
$\nu ::=$	$id(v_\nu^* \mapsto v_\nu)$	événement d'entrée-sortie
$t ::=$	$\epsilon \mid \nu.t$	traces finies (inductives)
$T ::=$	$\epsilon \mid \nu.T$	traces finies ou infinies (coinductives)
$B ::=$	$\text{terminates}(t, n)$	terminaison avec trace t et code de sortie n
	$ \text{diverges}(T)$	divergence avec trace T

Sémantique à grands pas

Jugements d'évaluation

- $G, E, sp \vdash a, M \Rightarrow v$ (expressions)
- $G, E, sp \vdash a^*, M \Rightarrow v^*$ (liste d'expressions)
- $G, E, sp \vdash i, M \xRightarrow{t} out, E', M'$ (instructions)
- $G \vdash Fd(v^*), M \xRightarrow{t} v, M'$ (invocations de fonctions)
-
- $\vdash p \Rightarrow \text{terminates}(t, n)$ (programmes)

Sémantique à grands pas

Jugements d'évaluation

$G, E, sp \vdash a, M \Rightarrow v$	(expressions)
$G, E, sp \vdash a^*, M \Rightarrow v^*$	(liste d'expressions)
$G, E, sp \vdash i, M \xRightarrow{t} out, E', M'$	(instructions)
$G \vdash Fd(v^*), M \xRightarrow{t} v, M'$	(invocations de fonctions)
$G, E \vdash s, M \xRightarrow{T} \infty$	(instructions, cas DV)
$G \vdash Fd(v^*), M \xRightarrow{T} \infty$	(invocations de fonctions, cas DV)
$\vdash p \Rightarrow B$	(programmes)

35 règles de sémantique (2 fois moins que pour C)

Exemples de règles de sémantique à grands pas

(évaluation d'expressions)

$$\frac{G, E, sp \vdash a_1, M \Rightarrow v_1 \quad G, E, sp \vdash a_2, M \Rightarrow v_2 \quad \text{eval_binop}(v_1, v_2, op) = [v]}{G, E, sp \vdash (a_1 op a_2), M \Rightarrow v}$$

$$\frac{G, E, sp \vdash adr, M \Rightarrow ptr(b, ofs) \quad \text{Mem.load}(\kappa, M, b, ofs) = [v]}{G, E, sp \vdash (\text{load } \kappa \text{ } adr), M \Rightarrow v}$$

Une valeur de type `option (t)` est soit ϵ (échec), soit $[x]$ (succès produisant le résultat $x : t$).

Exemples de règles de sémantique à grands pas

(exécution de boucles)

$$\frac{G, E, sp \vdash s, M \xRightarrow{t_1} \text{Normal}, E_1, M_1 \quad G, E_1, sp \vdash (\text{loop } s), M_1 \xRightarrow{t_2} \text{out}, E_2, M_2}{G, E, sp \vdash (\text{loop } s), M \xRightarrow{t_1 \circ t_2} \text{out}, E_2, M_2}$$

$$G, E, sp \vdash (\text{loop } s), M \xRightarrow{t_1 \circ t_2} \text{out}, E_2, M_2$$

$$\frac{G, E, sp \vdash s, M \xRightarrow{t} \text{out}, E_1, M_1 \quad \text{out} \neq \text{Normal}}{G, E, sp \vdash (\text{loop } s), M \xRightarrow{t} \text{out}, E_1, M_1}$$

$$G, E, sp \vdash (\text{loop } s), M \xRightarrow{t} \text{out}, E_1, M_1$$

Exemples de règles de sémantique à grands pas

(exécution de boucles)

$$G, E, sp \vdash s, M \xRightarrow{t_1} \text{Normal}, E_1, M_1$$

$$G, E_1, sp \vdash (\text{loop } s), M_1 \xRightarrow{t_2} \text{out}, E_2, M_2$$

$$G, E, sp \vdash (\text{loop } s), M \xRightarrow{t_1 \circ t_2} \text{out}, E_2, M_2$$

$$G, E, sp \vdash s, M \xRightarrow{t} \text{out}, E_1, M_1 \quad \text{out} \neq \text{Normal}$$

$$G, E, sp \vdash (\text{loop } s), M \xRightarrow{t} \text{out}, E_1, M_1$$

$$G, E, sp \vdash s, M \xRightarrow{T} \infty$$

$$G, E, sp \vdash (\text{loop } s), M \xRightarrow{T} \infty$$

$$G, E, sp \vdash s, M \xRightarrow{t_1} \text{Normal}, E_1, M_1 \quad G, E_1, sp \vdash (\text{loop } s), M_1 \xRightarrow{t_2} \infty$$

$$G, E, sp \vdash (\text{loop } s), M \xRightarrow{t_1 \circ t_2} \infty$$

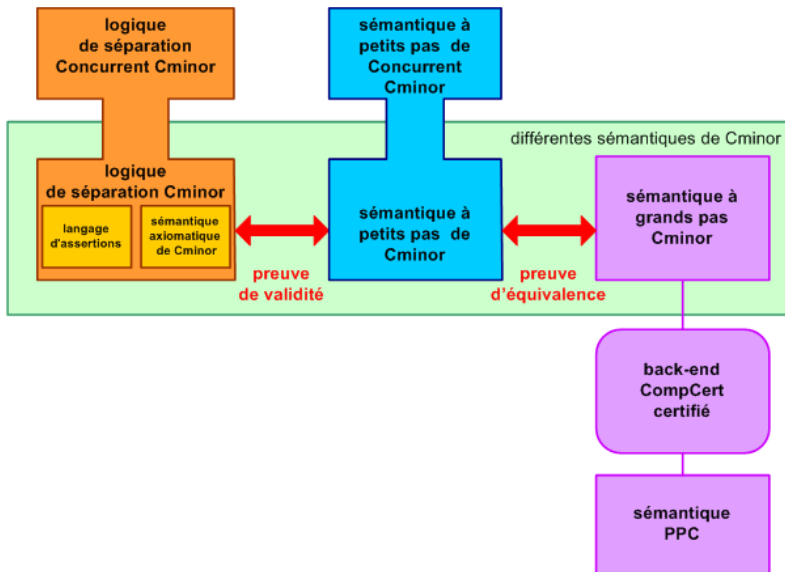
Exemples de règles de sémantique à grands pas (entrées et sorties de blocs)

$\text{Exit}(0) \overset{\text{block}}{\rightsquigarrow} \text{Normal}$ $\text{Exit}(n+1) \overset{\text{block}}{\rightsquigarrow} \text{Exit}(n)$

$$\frac{G, E, sp \vdash s, M \overset{t}{\Rightarrow} \text{out}, E_1, M_1 \quad \text{out} \overset{\text{block}}{\rightsquigarrow} \text{out}_1}{G, E, sp \vdash (\text{block } s), M \overset{t}{\Rightarrow} \text{out}_1, E_1, M_1}$$
$$G, E, sp \vdash (\text{exit}(n)), M \overset{\epsilon}{\Rightarrow} \text{Exit}(n), E, M$$
$$\frac{G, E, sp \vdash s, M \overset{T}{\Rightarrow} \infty}{G, E, sp \vdash (\text{block } s), M \overset{T}{\Rightarrow} \infty}$$

Différentes sémantiques de Cminor

De grands pas à petits pas



La logique de séparation en une page

Logique de Hoare : $\{P\}c\{Q\}$

Une extension de la logique de Hoare pour des programmes manipulant des **pointeurs**.

Les assertions décrivent à la fois la **pile** et le **tas**.

Certains opérateurs décrivent le tas : c.f. $e_1 \mapsto e_2$ et $p_1 * p_2$.

$(P * Q)(s, h) = \exists h_1, \exists h_2. h = h_1 \oplus h_2 \wedge P(s, h_1) \wedge Q(s, h_2)$

Raisonnement modulaire : il est possible de raisonner indépendamment sur des programmes distincts accédant à des zones séparées de la mémoire.

$$\frac{\{P\}c\{Q\}}{\{P * R\}c\{Q * R\}}$$

si aucune variable libre dans R n'est modifiée par c

Sémantique à continuations

- Évaluation des expressions : **sémantique à grands pas**

$G, \sigma \vdash a \Rightarrow v$ avec $\sigma = (sp; E; \phi; M)$ et $\phi ::= id \mapsto$ permission

→ ϕ permet une observation plus fine du tas (logique de séparation)

$$\frac{G, E, sp \vdash \mathit{adr}, M \Rightarrow \mathit{ptr}(b, ofs) \quad \mathit{adr} \in_{\mathit{load}}^{\kappa} \phi \quad \text{Mem.load}(\kappa, M, b, ofs) = [v]}{G, E, sp \vdash (\mathit{load} \kappa \mathit{adr}), M \Rightarrow v}$$

Sémantique à continuations

- Exécution des instructions : **sémantique à petits pas, à continuations**

Les **structures de contrôle non locales** (i.e. `exit` et `return`) de Cminor compliquent les preuves de propriétés sémantiques.

Les **continuations** permettent d'utiliser les principes d'induction générés par Coq.

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

$k : \text{continuation} ::= (\sigma, \kappa)$

$G \vdash k \xrightarrow{t} k'$

$G \vdash k \xrightarrow{t}^* k'$

$G \vdash k \xrightarrow{T}^* \infty$

Exemples de règles de sémantique à petits pas

$$G \vdash (\sigma, (\text{loop } \mathbf{s}) \cdot \kappa) \xrightarrow{t} (\sigma, \mathbf{s} \cdot \text{loop } \mathbf{s} \cdot \kappa)$$

$$G \vdash (\sigma, (\text{block } \mathbf{s}) \cdot \kappa) \xrightarrow{t} (\sigma, \mathbf{s} \cdot \text{Kblock } \kappa)$$

$$G \vdash (\sigma, \text{exit}(0) \cdot \mathbf{s}_0 \cdots \mathbf{s}_j \cdot \text{Kblock } \kappa) \xrightarrow{t} (\sigma, \kappa) \quad \text{pour } j \geq 0$$

$$G \vdash (\sigma, \text{exit}(n+1) \cdot \mathbf{s}_0 \cdots \mathbf{s}_j \cdot \text{Kblock } \kappa) \xrightarrow{t} (\sigma, \text{exit}(n) \cdot \kappa) \\ \text{pour } j \geq 0$$

Sémantique axiomatique

Plongement superficiel des assertions en Coq

P, Q : `assert` avec $P G \sigma$: `Prop`

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

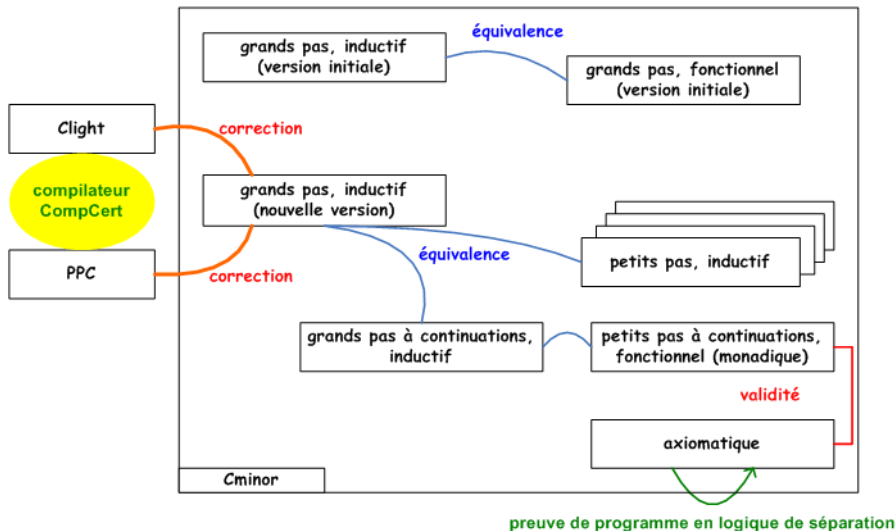
Interprétation des triplets étendus, fondée sur la notion de continuation sûre

Validité de $G; R; B \vdash \{P\}i\{Q\}$ par rapport à $G \vdash k \xrightarrow{t} k'$

Exemple :

$$\frac{G; R; B \vdash \{I\}s\{I\}}{G; R; B \vdash \{I\}\text{loop}(s)\{\mathbf{false}\}}$$

Validation de la sémantique de Cminor



Plan

- 1 Les sémantiques de Cminor
- 2 Un modèle mémoire pour un compilateur du langage C**
- 3 Conclusions et perspectives

Modèle mémoire

- But : définir la **géographie** de la mémoire, les **opérations** de gestion et leurs **propriétés** (**séparation**, **adjacence**, **confinement**)
- Besoins antagonistes :
 - ▶ pointeurs et arithmétique de pointeurs
 - ★ **chevauchement** partiel de zones
 - ▶ garanties d'isolation et de fraîcheur
 - ★ **séparation** des zones correspondant à 2 appels successifs à `malloc`
- Modèle commun à tous les langages du compilateur CompCert
- Modèle **générique** défini à différents niveaux d'abstraction

Un modèle mémoire pour un compilateur C

(niveau d'abstraction)

Quel **niveau d'abstraction** ?

- Trop concret (tableau d'octets) :
 - ▶ Les propriétés attendues ne sont pas exprimables (par ex., séparation de zones).
 - ▶ Ne permet pas de raisonner sur la mémoire.
- Trop abstrait (collection de blocs disjoints) :
 - ▶ Ne permet pas de définir l'arithmétique de pointeurs.
 - ▶ Les sémantiques deviennent incorrectes.

Un modèle mémoire pour un compilateur C

(niveau d'abstraction)

Quel **niveau d'abstraction** ?

- Trop concret (tableau d'octets) :
 - ▶ Les propriétés attendues ne sont pas exprimables (par ex., séparation de zones).
 - ▶ Ne permet pas de raisonner sur la mémoire.
- Trop abstrait (collection de blocs disjoints) :
 - ▶ Ne permet pas de définir l'arithmétique de pointeurs.
 - ▶ Les sémantiques deviennent incorrectes.

Certaines passes du compilateur effectuent des transformations des blocs de mémoire non triviales, nécessitant de raisonner sur la mémoire.

→ **Plusieurs niveaux d'abstraction**

Niveaux d'abstraction

Modèle abstrait (Coq)

- géographie de la mémoire non définie
- axiomatisation des opérations de gestion de la mémoire
- autres axiomes

Raffinement 1 (Coq)

- davantage d'axiomes
- lemmes (dérivés)

Raffinement 2 (Coq)

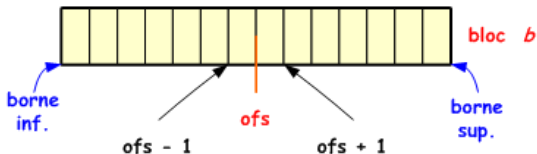
- Toutes les abstractions sont définies.
- Tous les axiomes sont prouvés.

Implémentation (Caml)

Modèle mémoire

Définition intuitive

- La mémoire est une collection de blocs séparés, et de tailles variables.
- Chaque bloc se comporte comme un tableau d'octets.
- Une adresse est un couple (b, ofs) .



Opérations de gestion de la mémoire

Lecture et écriture en mémoire

Définition abstraite

- $\text{load} : \text{memtype} \times \text{mem} \times \text{block} \times \mathbb{Z} \rightarrow \text{option val}$
- $\text{store} : \text{memtype} \times \text{mem} \times \text{block} \times \mathbb{Z} \times \text{val} \rightarrow \text{option mem}$

Une valeur $\text{option}(t)$ est soit ϵ (échec),
soit $\lfloor x \rfloor$ (succès produisant le résultat $x : t$).

Propriétés de bonne formation des variables

- Si $\text{store}(\tau, m, b, ofs, v) = \lfloor m' \rfloor$ et $\tau \sim \tau'$, alors
 $\text{load}(\tau', m', b, ofs) = \text{convert}(v, \tau')$
- Si $\text{store}(\tau, m, b, ofs, v) = \lfloor m' \rfloor$ et
 $b' \neq b \vee ofs' + |\tau'| \leq ofs \vee ofs + |\tau| \leq ofs'$, alors
 $\text{load}(\tau', m', b', ofs') = \text{load}(\tau', m, b', ofs')$

Un exemple d'utilisation des propriétés

du modèle mémoire dans la sémantique de Clight

Exemple de séquence écriture-lecture

```
union { int i; float f; } u;  
float x;  
...  
u.i = 8;  
x = u.f;  
...
```

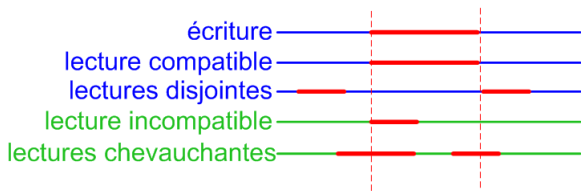
Standard C : comportement indéfini

Dans la sémantique de Clight, l'écriture d'une donnée de type τ peut être suivie d'une lecture d'une donnée d'un type τ' ssi τ est **compatible** avec τ' ($\tau \sim \tau'$).

CompCert est un compilateur certifié. Des comportements non définis dans le standard C doivent être modélisés.

Un exemple de propriété concrète

Écriture suivie d'une lecture dans un même bloc (suite)



En plus des lectures **compatibles** et **disjointes**, les seules lectures possibles suite à une écriture $\text{store}(\tau, m, b, ofs, v) = \lfloor m' \rfloor$ sont les suivantes :

- **Incompatible** : $\tau \not\sim \tau'$, auquel cas $\text{load}(\tau', m', b, ofs) = \lfloor \text{undef} \rfloor$.
- **Chevauchement** $ofs' \neq ofs$ et $ofs' + |\tau'| > ofs$ et $ofs + |\tau| > ofs'$, auquel cas $\text{load}(\tau', m', b, ofs') = \lfloor \text{undef} \rfloor$.

Lien avec la sémantique de Clight

`load` : memtype \times mem \times block \times $\mathbb{Z} \rightarrow$ option val

`loadval`($\tau, M, (b, ofs)$) = `load`(τ, M, b, ofs) if $\mathcal{A}(\tau) = \text{By_value}$
`loadval`($\tau, M, (b, ofs)$) = $\lfloor (b, ofs) \rfloor$ if $\mathcal{A}(\tau) = \text{By_reference}$
`loadval`($\tau, M, (b, ofs)$) = ϵ if $\mathcal{A}(\tau) = \text{By_nothing}$

$$\frac{G, E \vdash a, M \Leftarrow \text{adr} \quad \text{loadval}(\text{type}(a), M', \text{adr}) = \lfloor v \rfloor}{G, E \vdash a, M \Rightarrow v}$$

Autres relations définies au niveau abstrait

$$m \models b$$

b est valide dans m si b a été alloué dans m mais pas encore libéré.

Bornes d'un bloc

$$\mathcal{B}(m, b) = [l, h[$$

Accès valide à un bloc de mémoire

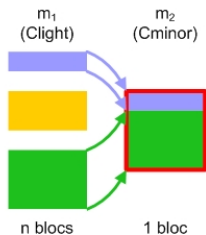
$$m \models \tau @ b, ofs$$

$$m \models b \wedge \langle \tau \rangle \text{ divise } ofs \wedge \mathcal{L}(m, b) \leq ofs \wedge ofs + |\tau| \leq \mathcal{H}(m, b)$$

Ces relations sont utiles au compilateur, ainsi qu'à la preuve de programme.

Transformations des blocs de mémoire

Au niveau abstrait : notion de **plongement** mémoire
(**invariants** reliant les états mémoire en tout point de l'exécution du programme initial et du programme transformé).



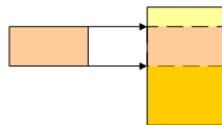
injection

allocation de registres



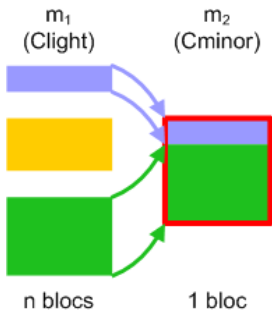
raffinement

vidage de registres

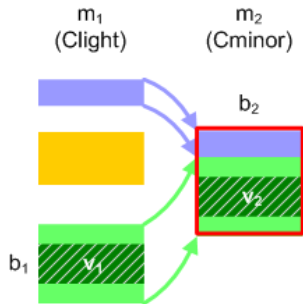
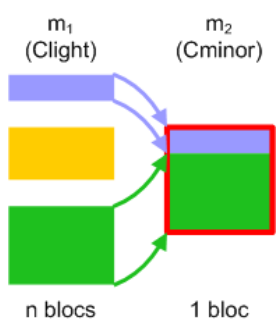


extension

Injection de mémoire



Injection de mémoire



Exemple de propriété de simulation

Plan

- 1 Les sémantiques de Cminor
- 2 Un modèle mémoire pour un compilateur du langage C
- 3 Conclusions et perspectives**

Conclusion

Bilan des travaux présentés

- Les performances de CompCert sont comparables à celles de `gcc -O1`.
- Coq est adapté à la vérification formelle de compilateurs.
 - ▶ Ouvre la voie à la vérification formelle de programmes opérant sur des programmes.
- Choisir un style de sémantique adapté à la vérification formelle d'une transformation de programme n'est pas aisé.

Perspectives d'améliorations de CompCert

à court terme

- Étendre la sémantique de C à différents niveaux d'abstraction :
 - ▶ modéliser l'instruction goto,
 - ▶ prendre en compte le non déterminisme,
 - ▶ autoriser des violations courantes du standard C.
- Étendre le modèle mémoire à différents niveaux d'abstraction.
- Utiliser de façon conjointe procédures de décision et preuve interactive.
- Renforcer la confiance en les sémantiques.
- Utiliser le compilateur sur de vrais logiciels embarqués.

Perspectives d'améliorations

à plus long terme

- Vérifier formellement davantage de transformations de programmes :
 - ▶ allocation optimale de registres (thèse de Benoît Robillard),
 - ▶ analyses statiques.
- Vérifier formellement des générateurs de code C.
- Formaliser les liens avec d'autres outils de vérification.