

Vérification formelle d'un compilateur C: un tutoriel

Sandrine Blazy¹ Xavier Leroy²

¹ENSIIE

²INRIA Paris-Rocquencourt

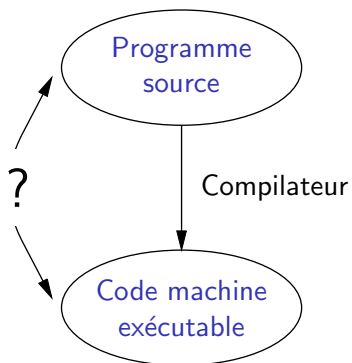
2009-02-24

ensiie

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

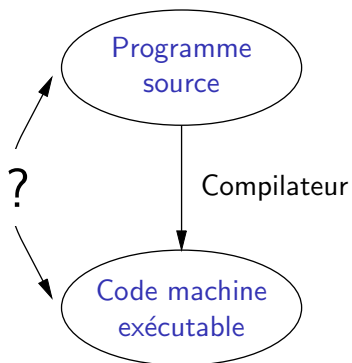
 **INRIA**

Faites-vous confiance à votre compilateur ?



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

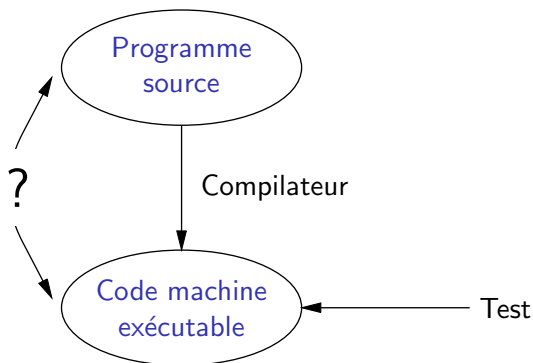
Faites-vous confiance à votre compilateur ?



Logiciel non critique :

Les bugs du compilateur sont négligeables devant ceux du logiciel.

Faites-vous confiance à votre compilateur ?

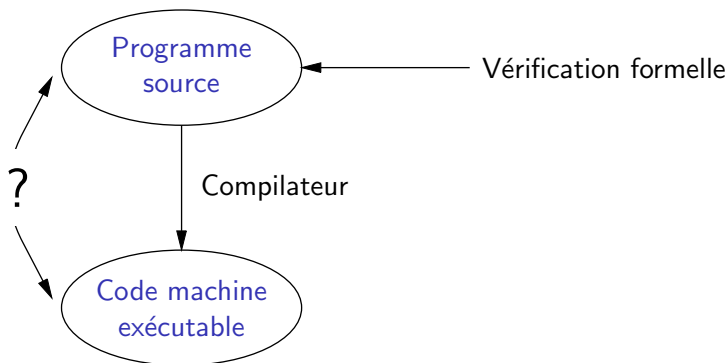


Logiciel critique validé par test systématique :

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

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

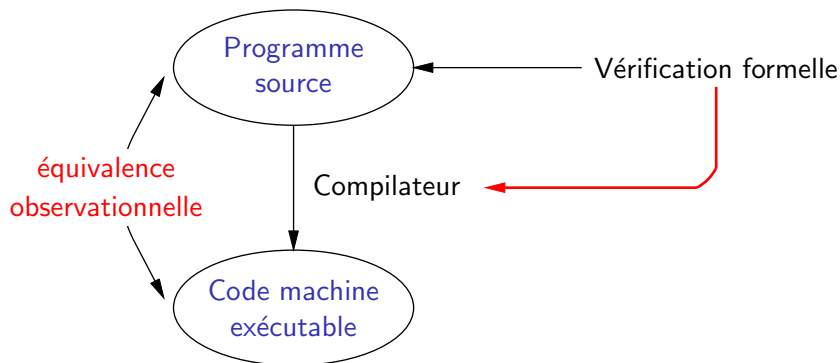
Faites-vous confiance à votre compilateur ?



Logiciel critique vérifié par méthodes formelles :

Ce qui est formellement vérifié 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.

Les propriétés vérifiées sur le source restent vraies pour le code exécutable.

Le projet CompCert

Développer et vérifier formellement un compilateur réaliste, utilisable pour le logiciel embarqué critique.

- Langage source : un sous-ensemble de C.
- Langage cible : assembleur du processeur PowerPC.
- Produit du code raisonnablement compact et efficace
⇒ plusieurs passes et quelques optimisations.

Nous suivons une approche de “software-proof codesign” (par opposition à prouver correct un compilateur existant).

CompCert et Coq

Le projet utilise l'assistant de preuve Coq pour :

- 1 Mener la preuve de correction du compilateur et la faire vérifier par machine.
(48000 lignes de Coq, 3 hommes-années.)
- 2 Programmer les parties vérifiées du compilateur directement dans le langage de spécification de Coq, en style fonctionnel pur.
→ Facilite grandement la preuve.
Un compilateur exécutable est obtenu par extraction automatique de code Caml depuis les specs Coq, puis compilation par Caml.

CompCert en pratique

Un compilateur “en ligne de commande” très classique :

```
ccomp -c src1.c
ccomp -c src2.c
ccomp -o executable src1.o src2.o -lm
```

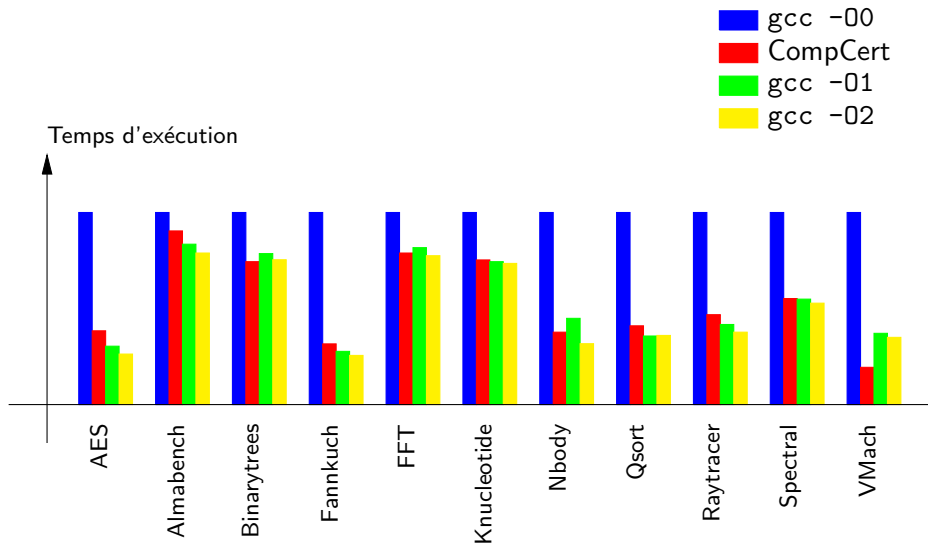
Tourne sur toute plate-forme supportée par Caml.
(Mais : a besoin d'un assembleur et d'un linker externes.)

Produit du code pour les plate-formes suivantes :

| | |
|-------------------|---------------------|
| PowerPC / MacOS X | (mature) |
| PowerPC / EABI | (récent) |
| ARM / old-ABI | (très expérimental) |

Performances du code produit : > 2 fois plus rapide que gcc -O0 ; 7% plus lent que gcc -O1 ; 12% plus lent que gcc -O2.

Performances du code produit



Disponibilité

Distribution source + preuves disponible librement à des fins d'évaluation et de recherche :

<http://compcert.inria.fr/>

(+ publications scientifiques)

Plan

- 1 Vue d'ensemble
- 2 Notions de préservation sémantique
- 3 Liens avec la validation de traduction
- 4 Liens avec la preuve de programmes
- 5 Le sous-ensemble Clight de C
- 6 Transformations et optimisations effectuées

CompCert, un compilateur formellement vérifié

Propriété de **préservation sémantique** (i.e. correction) du compilateur :

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 .

Remarques :

- La propriété est vérifiée en Coq **une fois pour toutes**.
- La compilation **peut échouer**.
- La sémantique de S **peut être indéfinie**.

Événements observables

Quels événements observer ?

- le comportement d'un programme
 - ▶ terminaison, erreur à l'exécution, divergence
- les valeurs calculées,
- les appels de fonctions,
- les accès à la mémoire,
- le code mort,
- ...

Compromis à trouver : précision des événements vs. confiance en la compilation

Sémantique formelle d'un programme

Description formelle des exécutions possibles d'un programme

Plusieurs styles sont possibles, et ces styles sont équivalents.

- Sémantique opérationnelle (à grands pas ou à petits pas)
 - ▶ jugements d'évaluation (ex. : $\rho \vdash a \Rightarrow v$)
 - ▶ adaptée à la vérification formelle de propriétés sémantiques
- Sémantique axiomatique
 - ▶ assertions (ex. : $\{P\}i\{Q\}$)
 - ▶ adaptée à la preuve de programme

Sémantique opérationnelle à grands pas

Jugement d'évaluation :

$$\rho \vdash \text{objet syntaxique} \Rightarrow \text{comportement observé}, \rho'$$

Exemples :

- $\rho \vdash e \Rightarrow v$ si pas d'effet de bord dans l'évaluation des expressions,
- $\rho \vdash e \Rightarrow v, \rho'$ sinon

Chaque langage de CompCert est défini par une sémantique opérationnelle.

Évaluation des expressions : sémantique à grands pas

$$\rho \vdash c \Rightarrow c \qquad \frac{\{x \mapsto v\} \in \rho}{\rho \vdash x \Rightarrow v}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \quad \rho \vdash e_2 \Rightarrow v_2 \quad \text{plus}(v_1, v_2) = v}{\rho \vdash e_1 + e_2 \Rightarrow v}$$

$$\frac{\rho \vdash e_1 \Rightarrow v_1 \quad \rho \vdash e_2 \Rightarrow v_2 \quad \text{div}(v_1, v_2) = v \quad v_2 \neq 0}{\rho \vdash e_1 / e_2 \Rightarrow v}$$

Seuls les comportements attendus sont spécifiés.

Exemple d'évaluation

$$\{x \mapsto 1; y \mapsto 2\} \vdash x \Rightarrow 1$$

$$\{x \mapsto 1; y \mapsto 2\} \vdash y \Rightarrow 2$$

$$\text{plus}(1, 2) = 3$$

$$\{x \mapsto 1; y \mapsto 2\} \vdash x + y \Rightarrow 3 \quad \{x \mapsto 1; y \mapsto 2\} \vdash 4 \Rightarrow 4$$

$$\text{plus}(3, 4) = 7$$

$$\{x \mapsto 1; y \mapsto 2\} \vdash (x + y) + 4 \Rightarrow 7$$

Évaluer une expression revient à construire un arbre de dérivation.

Propriétés sémantiques

Une sémantique formelle définit une relation d'exécution. Vérifier des propriétés sémantiques revient à raisonner sur cette relation.

Exemple : déterminisme de l'évaluation des expressions

$$\frac{\rho \vdash e \Rightarrow v_1 \quad \rho \vdash e \Rightarrow v_2}{v_1 = v_2}$$

Cette propriété se démontre par induction, en considérant toutes les formes possibles d'évaluation (i.e. toutes les règles de sémantique) d'une expression.

Exécution d'instructions : sémantique à transitions (petits pas)

Jugement $i, \rho \rightarrow i', \rho'$

Définit une étape élémentaire de calcul :

i est l'instruction courante

ρ l'environnement initial

ρ' l'environnement modifié après l'étape

i' l'instruction résiduelle.

Règles de transition

$$\frac{\rho \vdash e \Rightarrow v}{x=e, \rho \rightarrow \text{skip}, \rho[x \leftarrow v]}$$

$$\frac{i_1, \rho \rightarrow i'_1, \rho'}{i_1; i_2, \rho \rightarrow i'_1; i_2, \rho'}$$

$$\text{skip}; i_2, \rho \rightarrow i_2, \rho$$

$$\frac{\rho \vdash e \Rightarrow v \quad \text{is_true}(v)}{\text{if } e \text{ then } i_1 \text{ else } i_2, \rho \rightarrow i_1, \rho}$$

$$\frac{\rho \vdash e \Rightarrow v \quad \text{is_false}(v)}{\text{if } e \text{ then } i_1 \text{ else } i_2, \rho \rightarrow i_2, \rho}$$

$$\frac{\rho \vdash e \Rightarrow v \quad \text{is_true}(v)}{\text{while } (e)\{i\}, \rho \rightarrow i; \text{while } (e)\{i\}, \rho}$$

$$\frac{\rho \vdash e \Rightarrow v \quad \text{is_false}(v)}{\text{while } (e)\{i\}, \rho \rightarrow \text{skip}, \rho}$$

Séquences de transitions

Le comportement d'un programme i dans un environnement initial ρ se définit en enchaînant les transitions :

- 1 **Terminaison** avec état final ρ' :
séquence de transitions qui finit sur skip, ρ' .

$$i, \rho \rightarrow i_1, \rho_1 \rightarrow \dots \rightarrow \text{skip}, \rho'$$

- 2 **Divergence** : séquence infinie de transitions.

$$i, \rho \rightarrow i_1, \rho_1 \rightarrow \dots \rightarrow \dots$$

- 3 **Comportement indéfini (plantage)** :
séquence de transitions qui bloque.

$$i, \rho \rightarrow i_1, \rho_1 \rightarrow \dots \rightarrow i_n, \rho_n \not\rightarrow$$

avec $i_n \neq \text{skip}$. (Exemple : $x := 1; y := 0; z := x/y;$)

Équivalence observationnelle entre deux programmes

Soient P et P' deux programmes.

(Par exemple, P' est le résultat d'une passe de compilation appliquée à P .)

Comment définir sémantiquement le fait que P et P' se comportent «pareil» ?

Observation du comportement global

P et P' se comportent «pareil» si :

- Si P termine sur l'état ρ , alors P' termine sur un état ρ' , et ρ, ρ' sont égaux (ou équivalents modulo un certain critère).
- Si P diverge, alors P' diverge.
- Si P plante, alors P' plante. (Ou : fait ce qu'il veut.)

Exemples

Programmes qui terminent :

```
i = 0; x = 0;
while (i < 10) {
  x = x + i;
  i = i + 1;
}
```

```
i = 10;
x = 45;
```

Programmes qui divergent :

```
i = 0;
while (1) {
  print(i);
  i = i + 1;
}
```

```
while (1) {
  skip;
}
```

Notion d'équivalence sémantique mal adaptée aux programmes réactifs
(qui font des entrées-sorties).

Observation de traces complètes

P et P' se comportent «pareil» s'ils ont exactement les mêmes séquences de transitions :

$$\begin{array}{c}
 P, \rho \rightarrow i_1, \rho_1 \rightarrow i_2, \rho_2 \rightarrow \dots \\
 \Downarrow \\
 P', \rho \rightarrow i_1, \rho_1 \rightarrow i_2, \rho_2 \rightarrow \dots
 \end{array}$$

Exemples

Deux programmes qui ont les mêmes traces complètes :

```
y = x * 2;
```

```
y = x + x;
```

Deux programmes qui n'ont pas les mêmes traces complètes :

```
while (i < 10) {
  z = (x + y) + i;
  i = i + 1;
}
```

```
t = x + y;
while (i < 10) {
  z = t + i;
  i = i + 1;
}
```

Notion d'équivalence sémantique trop forte : interdit beaucoup d'optimisations.

Observation de traces partielles

Parmi les transitions, on distingue celles qui sont **observables** de celles qui sont **silencieuses**.

Deux programmes sont équivalents s'ils ont les mêmes traces de transitions observables.

Choix possibles de transitions observables :

- Appels et retours de fonctions.
(Mais : non préservé par *inlining* de fonctions.)
- Appels de fonctions d'entrées-sorties.
(Préservés par toutes les optimisations imaginables.)

Les choix faits dans CompCert

Le comportement observable d'un programme Clight P est :

- $P \Downarrow \text{terminates}(t, n)$:
terminaison, avec trace t d'entrées-sorties, et code de retour n
(valeur renvoyée par la fonction `main`).
- $P \Downarrow \text{diverges}(T)$:
divergence, avec trace T d'entrées-sorties (finie ou infinie).
- $P \Downarrow\!\!\Downarrow$:
plantage (comportement indéfini).

Les choix faits dans CompCert

Les fonctions d'un programme Clight sont ou bien :

- **Internes** : définies en Clight ; leurs appels ne sont pas observés dans les traces.
- **Externes** : déclarées mais non définies ; vues comme des opérations d'E/S ou « appels système » ; leurs appels sont enregistrés dans les traces.

Exemples

```
int main(void) { return 42; }
```

Comportement : $\text{terminates}(\epsilon, 42)$.

```
extern void print(int x); // appel système
int main(void) { print(42); return 0; }
```

Comportement : $\text{terminates}(t, 0)$ avec $t = \text{print}(42 \mapsto _)$.

```
extern void print(int x); // appel système
int main(void) { while (1) { print(42); } }
```

Comportement : $\text{diverges}(T)$ avec $T = (\text{print}(42 \mapsto _))^\omega$.

Le théorème de préservation sémantique

$(\forall P, P', B, \text{ si } P \Downarrow B \text{ et } P \text{ est compilé en } P', \text{ alors } P' \Downarrow B).$

En 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.
```

Ce théorème se décompose en autant de propriétés que de passes de compilation.

Preuve de préservation sémantique

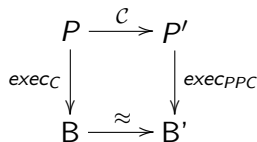
Diagramme commutatif (programme)

La compilation du programme P produit le programme P' ($P' = \mathcal{C}(P)$).

Supposons que $P \Downarrow B$ (langage C) et $P' \Downarrow B'$ (langage PPC).

On montre alors que $B \approx B'$.

Schéma de la preuve :



Preuve de préservation sémantique

Diagramme commutatif global (instruction)

L'instruction i est compilée en l'instruction i' .

Supposons que $i, \rho \rightarrow i_1, \rho_1$ (langage C) et $i', \rho' \rightarrow i'_1, \rho'_1$ (langage PPC).

On montre alors que si $\rho \approx \rho'$, alors $\rho_1 \approx \rho'_1$ (« l'invariant est préservé »).

Schéma de la preuve :

$$\begin{array}{ccc}
 i, \rho & \xrightarrow{\rho \approx \rho'} & i', \rho' \\
 \downarrow 1 \text{ pas} & & \downarrow n \text{ pas} \\
 i_1, \rho_1 & \xrightarrow{\rho_1 \approx \rho'_1} & i'_1, \rho'_1
 \end{array}$$

Comment valider une sémantique formelle ?

La vérification formelle de la propriété de préservation sémantique permet de gagner confiance en la sémantique de C.

D'autres propriétés sémantiques ont été vérifiées formellement, en particulier des propriétés d'équivalence entre différents styles.

Plan

- 1 Vue d'ensemble
- 2 Notions de préservation sémantique
- 3 Liens avec la validation de traduction**
- 4 Liens avec la preuve de programmes
- 5 Le sous-ensemble Clight de C
- 6 Transformations et optimisations effectuées

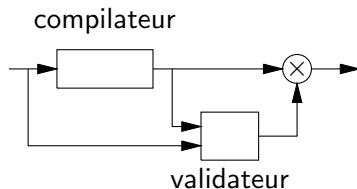
La validation de traduction

Au lieu d'établir de la confiance dans le compilateur lui-même (par tests, vérification formelle, etc), on valide a posteriori les résultats de la compilation :

Compilateur : $Source \rightarrow Code + Erreur$

Valideur : $Source \times Code \rightarrow \text{booléen}$

Si $Compilateur(S) = C$ and $Valideur(S, C) = \text{true}$, succès.
Sinon, erreur.



(\approx automatiser les revues manuelles du code assembleur engendré.)

Quelques techniques pour la validation

L'exécution symbolique de code :

```

int f(int x)
{
    int t = 6 * x;
    return t - 2;
}

```

```

        .globl _f
_f :
        stwu    r1, -32(r1)
        mflr    r12
        stw     r12, 12(r1)
        rlwinm  r5, r3, 1, 0xfffffffffe
        rlwinm  r4, r3, 2, 0xfffffffffc
        add     r3, r5, r4
        addi    r3, r3, -2
        lwz     r12, 12(r1)
        mtlr    r12
        lwz     r1, 0(r1)
        blr

```

$res = 6 * arg - 2$

$$\begin{aligned}
 r3_{end} = & (rol(r3_{start}, 1) \wedge (-2)) \\
 & + (rol(r3_{start}, 2) \wedge (-4)) \\
 & + (-2)
 \end{aligned}$$

Quelques techniques pour la validation

Combiner exécution symbolique et propriétés algébriques des opérateurs :

$$\begin{array}{ll}
 \text{rol}(x, n) \wedge (-2^n) = x \ll n & x * y = y * x \\
 x \ll n = x * 2^n & x - y = x + (-y) \\
 x * y + x * z = x * (y + z) & 2 + 4 = 6
 \end{array}$$

Cela montre que les deux exécutions symboliques sont égales :

$$\begin{aligned}
 & (\text{arg} \mapsto 6 * \text{arg} - 2) = \\
 & (\text{r3}_{start} \mapsto (\text{rol}(\text{r3}_{start}, 1) \wedge (-2)) + (\text{rol}(\text{r3}_{start}, 2) \wedge (-4)) + (-2))
 \end{aligned}$$

Connaissant en plus les conventions d'appel du processeur (argument et résultat dans le registre `r3` pour une fonction de type `int → int`), le validateur “voit bien que” le code machine calcule la même fonction que le source.

Quelques techniques pour la validation

- Exécution symbolique et simplifications algébriques.
- Heuristiques pour mettre en correspondance les points de contrôle (début de boucles, etc).
- Analyses statiques (non-aliasing, etc) pour justifier les simplifications algébriques.
- Éventuellement : model-checking, démonstration automatique.

Quelle confiance accorder à la validation ?

Compilateur non vérifié

(confiance faible)

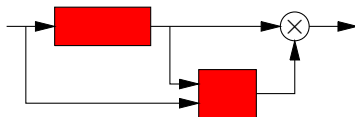
compilateur



Valdateur non vérifié

(confiance modérée)

compilateur



valdateur

Compilateur vérifié

(confiance élevée)

compilateur



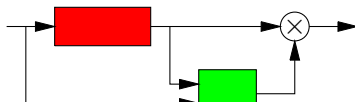
= vérifié formellement

= non vérifié

Valdateur vérifié

(confiance élevée)

compilateur



valdateur

Vérification formelle de validateurs

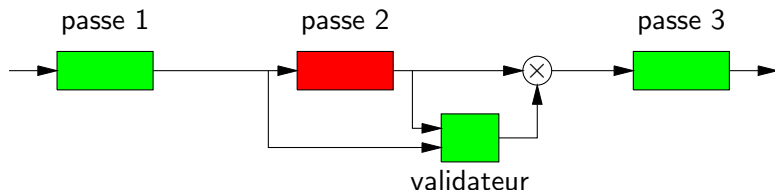
Un validateur est formellement vérifié si on a prouvé la propriété de correction suivante :

si $Validateur(S, C) = \text{true}$, alors C se comporte comme S

La combinaison d'un compilateur non vérifié et d'un validateur vérifié fournit des garanties de sûreté aussi fortes qu'un compilateur vérifié.

→ Une alternative intéressante si le validateur est plus simple que le compilateur.

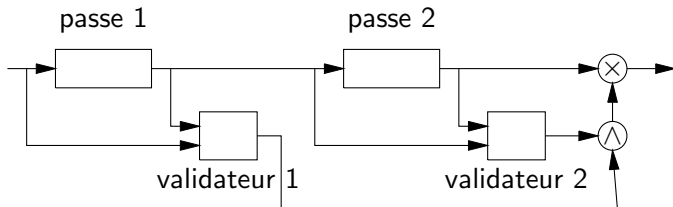
Combiner compilation vérifiée et validation vérifiée



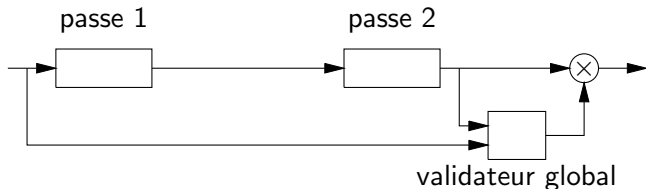
Validation et composition de passes de compilation

Validation compositionnelle, pour chaque passe

(si accès aux représentations intermédiaires du compilateur) :



Validation globale (si compilateur "boîte noire") :



État de l'art en validation

Validation de passes de compilation par des validateurs spécialisés :

| | |
|-------------------------------|---------------------------------------|
| Necula (2000) | plusieurs passes de gcc version 2 |
| Huang, Childers, Soffa (2006) | allocation de registres |
| Tristan, Leroy (2008) | list scheduling, trace scheduling (*) |
| Tristan, Leroy (2009) | lazy code motion (*) |
| Kundu, Tatlock, Lerner (2009) | software pipelining |

(*) Validateur formellement vérifié.

Validation globale par des validateurs généralistes :

| | |
|---------------------------|--------------|
| Rival (2004) | (implémenté) |
| Zuck, Pnueli (2002, 2005) | (théorique) |

Plan

- 1 Vue d'ensemble
- 2 Notions de préservation sémantique
- 3 Liens avec la validation de traduction
- 4 Liens avec la preuve de programmes**
- 5 Le sous-ensemble Clight de C
- 6 Transformations et optimisations effectuées

La preuve de programmes

1- Annoter le code source avec des **assertions logiques** : préconditions, postconditions, invariants de boucles, invariants de structures de données.

```

/*@ requires x >= 0 && x <= 9 ;
    ensures \result == min(x+1, 9) ;
    modifies \nothing ;
*/
int saturated_incr(int x)
{
    if (x >= 9) return x ; else return x + 1 ;
}

```

2- Vérifier que ces assertions sont vraies en utilisant une **logique de programme** (p.ex. la logique de Hoare), aussi connue sous le nom de **sémantique axiomatique**

Preuve de programmes et compilation

Objectif (vague) : toutes les propriétés du code source établies par preuve de programme doivent rester vraies pour le code compilé.

Deux approches :

- 1 Compilation des assertions logiques & de leurs preuves.
- 2 Raisonner plus globalement sur les comportements observables du programme et exploiter un résultat de préservation sémantique lors de la compilation.

Compilation des assertions et des preuves

(*Proof-preserving compilation*; *Certificate translation*.)

Étant donné :

- un code source S
- ... annoté par des assertions logiques
- et des preuves (en logique de Hoare) de la validité de ces assertions,

le compilateur produit :

- un code machine C
- ... annoté par des assertions logiques
- et des preuves (en logique de Hoare) de la validité de ces assertions.

Exemple de compilation d'assertions

Le code source :

```
/*@ requires x >= 0 && x <= 9 ;
    ensures \result == min(x+1, 9) ;
    modifies \nothing ;
*/
int saturated_incr(int x)
{
    if (x >= 9) return x; else return x + 1;
}
```

Exemple de compilation d'assertions

Le code compilé PowerPC :

```

/*@ requires r3 >= 0 && r3 <= 9 ;
   ensures  r3 == min(\old(r3)+1, 9)
           && pc = \old(lr)
   modifies r12, cr0, mem(r1-32 ... r1) ;
*/
saturated_incr :
    stwu  r1, -32(r1)
    mflr  r12
    stw   r12, 12(r1)
    cmpwi cr0, r3, 9
    bf    0, L100
    addi  r3, r3, 1
L100 :
    lwz   r12, 12(r1)
    mtlr  r12
    lwz   r1, 0(r1)
    blr

```

État de l'art

On sait faire la compilation d'assertions et de preuves pour un certain nombre de passes de compilation :

- Traduction code structuré → graphe de flot de contrôle.
- Propagation des constantes.
- Élimination du code mort.
- Allocation de registres.

(G. Barthe, B. Grégoire, C. Kunz, T. Rezk, 2006, 2008.)

Il reste beaucoup de travail pour étendre cela à un compilateur complet.

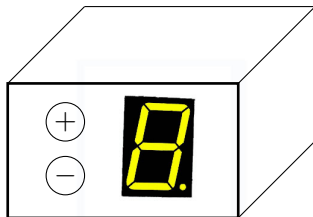
Assertions et comportements globaux

Thèse :

Les assertions logiques et leur vérification déductive («unit proofs») ne sont qu'une étape intermédiaire pour prouver des propriétés de correction globale du programme tout entier («functional proofs»).

Assertions et comportements globaux

Exemple :



Spécification informelle :

l'affichage augmente de 1 si on presse le bouton “+” (mais 9 reste à 9) ;
diminue de 1 si on presse le bouton “-” (mais 0 reste à 0) ;
et ne change pas si on presse 0 ou 2 boutons.

Le code source (1)

La fonction de transition :

```

/*@ requires x >= 0 && x <= 9 ;
    ensures (bplus && !bminus ==> \result = min(x + 1, 9))
           && (!bplus && bminus ==> \result = max(x - 1, 0))
           && (!bplus && !bminus ==> \result = x)
           && (bplus && bminus ==> \result = x) ;
*/
int step(int x, int bplus, int bminus)
{
    if (bplus) x = x + 1 ;
    if (bminus) x = x - 1 ;
    if (x > 9) x = 9 ;
    if (x < 0) x = 0 ;
    return x ;
}

```

Le code source (2)

Le programme réactif principal :

```
extern int readButton(int button_number); /* appel système */
extern void updateDisplay(int value);     /* appel système */

void main(void)
{
    int x = 0, bplus, bminus;

    while(1) {
        updateDisplay(x);
        bplus = readButton(1);
        bminus = readButton(2);
        x = step(x, bplus, bminus);
        sleep(0.25);
    }
}
```

Spécification de l'application

Ce programme s'exécute infiniment (sans planter), et sa trace d'événements d'entrée-sortie est de la forme

...updateDisplay($n_i \mapsto _$).readButton($1 \mapsto bp_i$).readButton($2 \mapsto bm_i$)..

avec $n_0 = 0$ et pour tout i ,

$$n_{i+1} = \begin{cases} n_i + 1 & \text{si } n_i < 9 \text{ et } bp_i \neq 0 \text{ et } bm_i = 0; \\ n_i - 1 & \text{si } n_i > 0 \text{ et } bp_i = 0 \text{ et } bm_i \neq 0; \\ n_i & \text{sinon.} \end{cases}$$

Compilation vérifiée et préservation de la spécification

Définition :

Un programme S satisfait une spécification fonctionnelle $Spec$ si S s'exécute sans planter, et tout comportement observable B de S satisfait $Spec(B)$.

Théorème :

Si C est le résultat de la compilation de S avec un compilateur formellement vérifié, alors

$$S \text{ satisfait } Spec \implies C \text{ satisfait } Spec$$

Cohérence entre sémantiques opérationnelle et axiomatique

Si un programme C est vérifié avec Frama-C et compilé par CompCert, est-on vraiment sûr que les propriétés établies par Frama-C sont vraies dans toute exécution ?

Cela pourrait être faux si la sémantique axiomatique utilisée par Frama-C et la sémantique opérationnelle utilisée par CompCert ne sont pas cohérentes.

Exemple d'incohérence

Scénario :

- La sémantique axiomatique utilise des entiers exacts (mathématiques).
- La sémantique opérationnelle utilise des entiers machine (modulo 2^{32}).

Exemple :

$$\{x \geq 0\} \quad x = x + 1; \quad \{x > 0\}$$

est prouvable dans la sémantique axiomatique, mais on peut observer l'exécution suivante :

- État initial : $x = 2^{31} - 1$ (la précondition $x \geq 0$ est vraie)
- État final : $x = -2^{31}$ (la postcondition $x > 0$ est fausse)

Exemple d'incohérence

Scénario :

- La sémantique opérationnelle a oublié de définir le comportement de l'arithmétique mixte "entier + flottant".
- La sémantique axiomatique traite correctement ce cas.

Exemple :

$$\{\text{true}\} \quad x = (\text{int})(3.0 + 2); \quad \{x = 5\}$$

est prouvable dans la sémantique axiomatique, mais l'exécution bloque sur un comportement non défini.

Preuves de cohérence

Pour éliminer ces risques d'incohérence, on peut prouver des théorèmes de **validité relative** d'une sémantique axiomatique par-rapport à une sémantique opérationnelle :

Sémantique axiomatique forte :

si $[P] \ i \ [Q]$,

pour tout état initial ρ satisfaisant P ,

il existe un état final ρ'

tel que $i, \rho \rightarrow \dots \rightarrow \text{skip}, \rho'$ et ρ' satisfait Q .

Sémantique axiomatique faible :

si $\{P\} \ i \ \{Q\}$,

pour tout état initial ρ satisfaisant P ,

ou bien i, ρ diverge (se réduit infiniment sans bloquer),

ou bien il existe un état final ρ'

tel que $i, \rho \rightarrow \dots \rightarrow \text{skip}, \rho'$ et ρ' satisfait Q .

Plan

- 1 Vue d'ensemble
- 2 Notions de préservation sémantique
- 3 Liens avec la validation de traduction
- 4 Liens avec la preuve de programmes
- 5 Le sous-ensemble Clight de C**
- 6 Transformations et optimisations effectuées

Le sous-ensemble Clight de C

Clight est :

- un **sous-ensemble** de C ANSI (C89)
(certaines constructions manquent) ;
- un **raffinement** de C ANSI
(la sémantique de Clight définit un certain nombre de comportements laissés indéfinis ou non spécifiés en C ANSI).

Constructions de C ANSI présentes dans Clight

Types :

entiers, flottants, tableaux, pointeurs, struct, union.

Expressions, opérateurs :

Toute l'arithmétique de C, y compris arithmétique de pointeurs.
Pas d'effets de bord dans les expressions (voir plus loin).

Instructions, structures de contrôle :

Tout le contrôle structuré : `if/then/else`, boucles `while/do/for`, `break/continue`, `switch` réguliers.

Fonctions :

y compris fonctions récursives et pointeurs vers fonctions.

Constructions de C ANSI absentes de Clight

Arithmétique en précision étendue : les types long long et long double.

Le contrôle non structuré : le goto, les switch irréguliers (Duff's device).

Les fonctions à nombre variable d'arguments.

Les «bit-fields» dans les struct.

Le passage par valeur et les affectations de struct et union :

```

struct s x, y;
x = y;           // rejeté
x = f(y);       // rejeté (passage par valeur)
g(&x, &y);      // accepté (passage par référence)

```

Le compilateur CompCert rejette à la compilation les programmes contenant ces constructions.

Constructions absentes de Clight mais émulées au parsing

CompCert utilise CIL comme parser. CIL expande plusieurs constructions de C ANSI en constructions plus simples qui sont dans Clight :

- Effets de bord à l'intérieur des expressions (*p++, etc).
- Variables locales à un bloc.
- Initialiseurs incomplets.
- Etc.

(Frama-C utilise également CIL et fait à peu près les mêmes expansions.)

CompCert effectue quelques simplifications supplémentaires au parsing :

- Expansion des typedef.
- Les attributs de types sont ignorés : const, volatile, restrict. ...

Comportements non spécifiés en C ANSI, spécifiés en Clight

Standard C ANSI : sémantique informelle, laissant certains comportements non spécifiés ou non définis.

Nombre de ces comportements ont été entièrement spécifiés dans la sémantique de Clight :

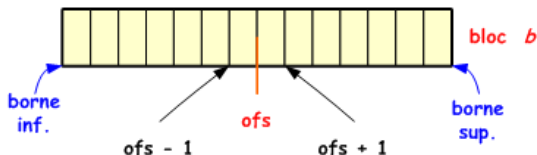
- Ordre d'évaluation des expressions (gauche à droite).
- Tailles et valeurs des types intégraux.
- Sémantique des débordements arithmétiques.
- Casts entre types de pointeurs incompatibles.

Les valeurs dans la sémantique de Clight

$$v = \begin{cases} \text{int}(n) & \text{entier 32 bits} \\ \text{float}(f) & \text{flottant 64 bits} \\ \text{ptr}(adr) & \text{pointeur} \end{cases}$$

Définition intuitive de la mémoire

- La mémoire est une collection de blocs séparés.
- Chaque bloc b se comporte comme un tableau d'octets.
- Une adresse adr est un couple (b, ofs) , avec ofs décalage (entier machine) en octets dans le bloc b .



Types entiers

En C : `char`, `short`, `int`, `long`, éventuellement signés.
 Les domaines de valeurs ne sont pas complètement définis
 (ex. `char ≤ short ≤ int ≤ long`).

En Clight : `int(tailleint, signe)`, avec $taille_{int} \in \{8; 16; 32\}$ et
 $signe \in \{S; U\}$

Toutes les correspondances sont spécifiées (ex. `int` devient `int(32, S)`).

Arithmétique entière complètement spécifiée et formalisée en Coq :

- Débordements : traités modulo 2^{32}
- Entiers signés : $[-2^{31}, 2^{31} - 1]$ (complément à 2).

Types flottants

En C : `float`, `double`.

En Clight : `float(taillefloat)`, avec $taille_{float} \in \{32; 64\}$

L'arithmétique des flottants est IEEE 754.

(Non complètement définie en Coq ; une axiomatisation légère suffit pour la preuve de CompCert.)

Casts

Entre types numériques (entiers, flottants) :
comme spécifié en C ANSI.

Entre types de pointeurs ou types entiers 32 bits :
on garantit que la valeur ne change pas.

```
int x;  int * p;

(int *) ((double *) p) == p
(int *) ((int) p) == p
(int) ((void *) x) == x
```

(Les casts pointeur \leftrightarrow entier 8/16 bits et pointeur \leftrightarrow flottant restent non spécifiés en Clight.)

Comportements non définis en Clight

Un certain nombre de comportements non définis en ANSI C sont également non définis dans la sémantique de Clight :

- Accès hors bornes de tableaux.
- Déréférencer NULL ou $(\text{char } *)\ n$ quand n est un entier.
- Division entière par zéro.
- Décalages de n bits quand $n < 0$ ou $n \geq 32$.
- Observation au niveau des bits de la représentation des nombres et des pointeurs (voir plus loin).

Que signifie «non défini» ?

Pour la sémantique formelle de Clight, «non défini» = plantage :

- aucune règle de sémantique ne s'applique ;
- le programme n'a donc pas de comportement observable (ni terminaison, ni divergence).

Pour le compilateur CompCert :

- Le compilateur ne détecte pas ces comportements non définis.
- En général, il produit du code PowerPC.
- Ce code peut faire n'importe quoi.
(Les hypothèses du résultat de préservation sémantique sont fausses.)
- En général, le code produit fait ce à quoi le programmeur s'attend.
- Mais ce n'est pas garanti formellement.

Exemple de comportement non défini

Code source :

```
int f(int x, int n)
{
    return x << n;
}
```

Code compilé :

```
f :
    slw r3, r3, r4
    blr
```

Sémantique Clight du code source :

- Si $0 \leq n < 32$, calcule $x * 2^n$.
- Sinon, comportement non défini = plantage.

Sémantique réelle du code PowerPC compilé :

- Si $n \bmod 64 < 32$, le résultat est $x * 2^{(n \bmod 64)}$.
- Si $n \bmod 64 \geq 32$, le résultat est 0.
- Ne plante jamais le processeur.

Comportements non définis liés à la mémoire

Écriture dans une case mémoire avec un type τ
 puis lecture dans cette même case avec un type τ' incompatible.

Avec des casts de pointeurs :

```
double x ;
char c ;

x = 3.14159 ;
c = *((char *) &x) ;
```

avec des unions :

```
union { double x ; char c ; } u ;
char c ;

u.x = 3.14159 ;
c = u.c ;
```

Types incompatibles : de tailles différentes, ou bien l'un est float et l'autre est int ou pointeur.

Comportements non définis liés à la mémoire

Laisser ce comportement non défini permet de rendre la sémantique indépendante des représentations des données au niveau du bit et de l'endianness du processeur.

Exemple :

```
unsigned char c[4] = { 0x12, 0x34, 0x56, 0x78 };
int x = *((int *) c);
```

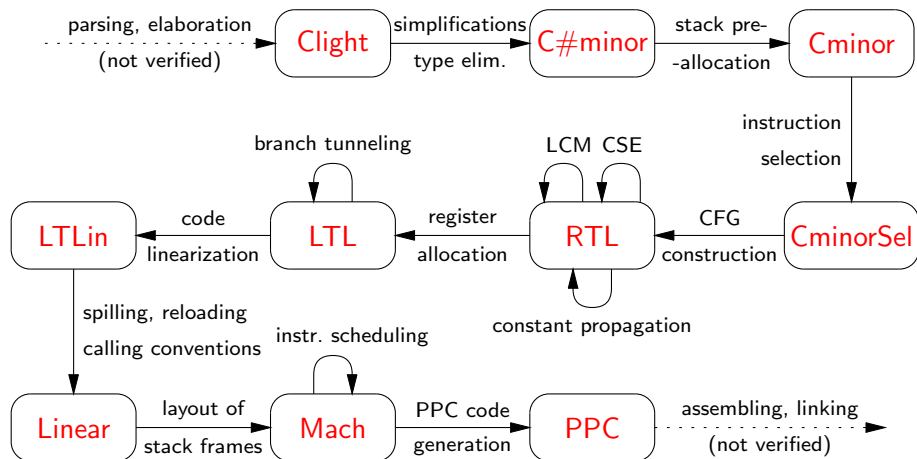
| | | |
|-------------|------------------------------------|--------------------------------|
| PowerPC : | x = 0x12345678 | (car big-endian) |
| Intel x86 : | x = 0x78563412 | (car little-endian) |
| Sparc : | x = 0x12345678 ou <i>bus error</i> | (car contraintes d'alignement) |
| Clight : | non défini | |

Travail envisagé dans le cadre d'U3CAT : raffiner le modèle mémoire de Clight pour pouvoir définir exactement la sémantique dans ce cas.

Plan

- 1 Vue d'ensemble
- 2 Notions de préservation sémantique
- 3 Liens avec la validation de traduction
- 4 Liens avec la preuve de programmes
- 5 Le sous-ensemble Clight de C
- 6 Transformations et optimisations effectuées**

Diagramme du compilateur CompCert C



1. Parsing et production de Clight

La bibliothèque CIL de Berkeley :

- Analyse syntaxique (parsing).
- Typage.
- Nombreuses simplifications.

+ une passe de simplifications supplémentaires.

(Transformations non vérifiées !)

Exemples de simplifications

Suppression des déclarations locales à un bloc :

```

if (x) {
  double x = a + b;
  ...
}
---->
double x2;
if (x) {
  x2 = a + b;
  ...
}

```

Suppression des effets de bord dans les expressions :

```

x = f(y) + *p++;
---->
tmp1 = *p;    p = p + 1;
tmp2 = f(y);  x = tmp2 + tmp1;

```

Explicitation des casts implicites :

```

int i; double f;
i += f;
---->
int i; double f;
i = (int) ((double) i + f);

```

2. De Clight à C#minor

Résolution de la surcharge des opérateurs :

```

x + y          ---->  x +i y
                   x +f y
                   x +i y *i sizeof(ty)
                   x *i sizeof(ty) +i y
  
```

(Selon les types de x et y.)

Explicitation des accès mémoire et des calculs d'adressage :

```

a[i] = p->field;  ---->  float64[a + i * 8] = float64[p + 16];
  
```

(Si a est un double[] et field est à l'offset 16.)

2. De Clight à C#minor

Traduction des boucles de C en boucles infinies + block + exit :

```

while (x) {
  ...
  if (y) break;
  if (z) continue;
  ...
}
    ---->
    block {
      loop {
        if (!x) exit 1;
        block {
          ...
          if (y) exit 2;
          if (z) exit 1;
          ...
        }
      }
    }
  
```

3. De C#minor à Cminor

Allocation explicite en pile des variables locales

- scalaires dont l'adresse est prise avec l'opérateur &
- ou non scalaires (tableaux, struct).

```

{                               --->  {
    char t[8];                   stacksize 12; // 8 + sizeof(int)
    int x, y;                     var y;
    f(t, &x);                     f(stack(0), stack(8));
    x = x + 1;                     int32[stack(8)] = int32[stack(8)] + 1;
    y = y - 1;                     y = y - 1;
}                               }
  
```

t et x sont placés en pile, aux offsets 0 et 8 respectivement.

y reste en variable locale et peut être alloué à un registre plus tard.

4. Sélection d'instructions

Remplacement des opérateurs standards de Cminor par des opérateurs spécifiques PowerPC représentant ce que le processeur sait faire en 1 instruction.

$(x + 1) + (y - 3)$ ---> `addimm(addint(x, y), -2)`

$x * 4$ ---> `rolm(x, 2, -4)`

$x * 6$ ---> `add(rolm(x, 2, -4), rolm(x, 1, -2))`

$(x + 2) * 4 - 1$ ---> `addimm(rolm(x, 2, -4), 7)`

$(x \ll 5) | (x \gg 27)$ ---> `rolm(x, 5, -1)`

`int8[x + 4]` ---> `load(int32, indexed_imm, x, 4)`

`int32[x + y]` ---> `load(int32, indexed_2, x, y)`

5. Conversion en RTL-CFG

RTL = *Register Transfer Language* = “code 3 adresses”

CFG = *Control-Flow Graph*

Le code d'une fonction est représenté par un CFG :

- Nœuds = instructions correspondant à celles du processeur, opérant sur des variables (ou temporaires).

$z = x +_f y$ addition flottante

$i = i + 1$ addition entière immédiate

$\text{if } (x > y)$ test et branchement conditionnel

- Arc de I à J = J est un successeur de I
(J peut s'exécuter juste après I).

5. Conversion en RTL-CFG

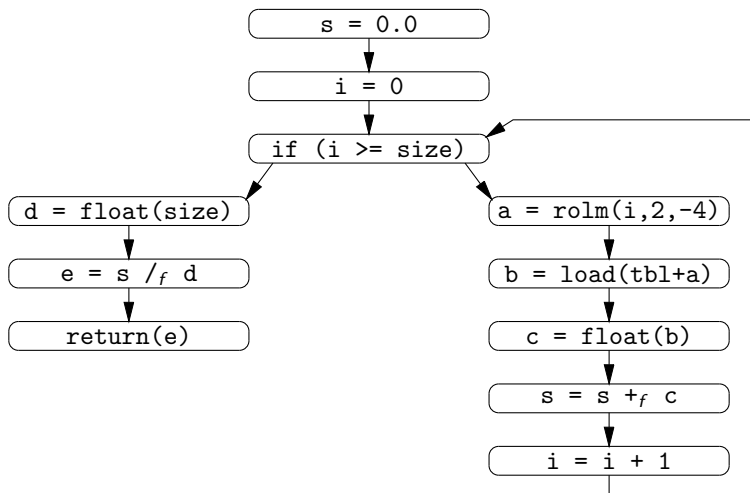
- Décomposer les expressions en instructions élémentaires + introduction de temporaires pour les résultats intermédiaires.
- Traduire les structures de contrôle (conditionnelles, boucles) en arcs du CFG.

Exemple : on part du code CminorSel correspondant au code C suivant.

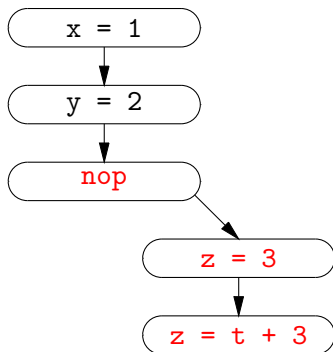
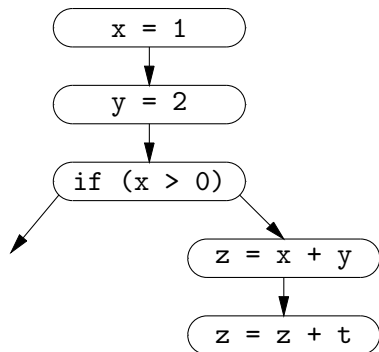
```
double average(int * tbl, int size)
{
    double s = 0; int i;
    for (i = 0; i < size; i++) s += tbl[i];
    return s / size;
}
```

5. Conversion en RTL-CFG

Le CFG correspondant est :

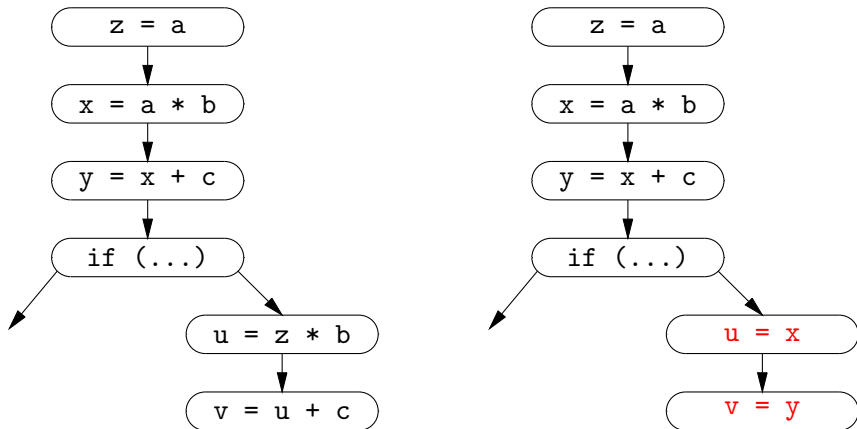


6. Propagation des constantes



S'appuie sur les résultats d'une analyse statique de type *dataflow*.

7. Élimination des sous-expressions communes (CSE)



Algorithme utilisé : *value numbering* à l'échelle des blocs de base étendus.

8. Lazy Code Motion

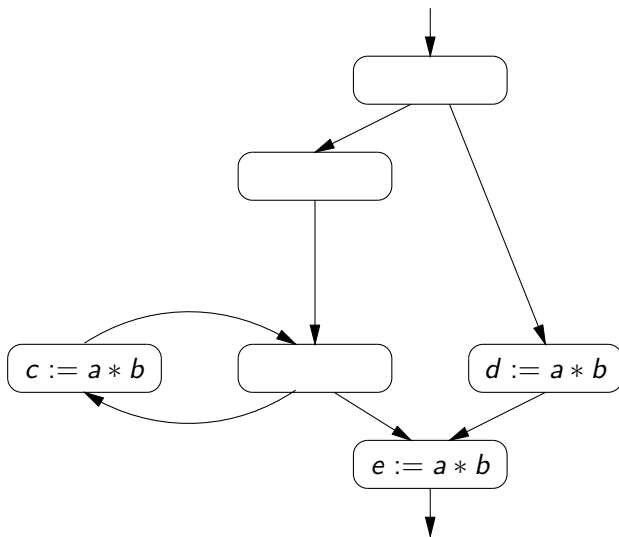
Une généralisation de la CSE qui peut déplacer des instructions pour éliminer davantage de calculs redondants.

(En particulier, «sortir» d'une boucle des calculs invariants.)

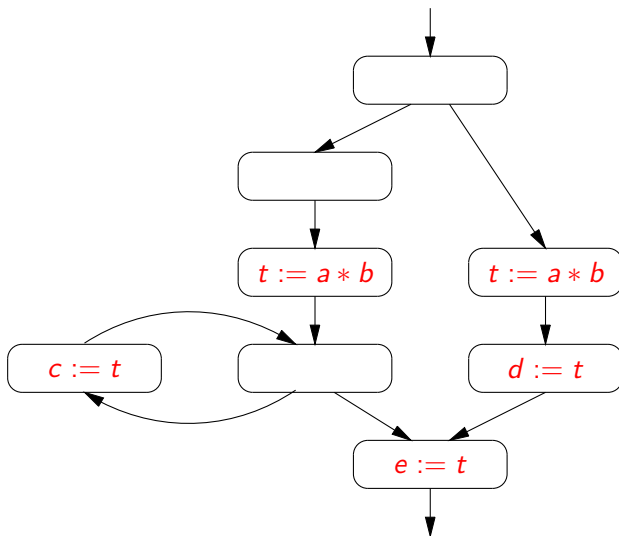
Utilise l'approche de validation a posteriori avec un validateur prouvé.

Pas encore intégré dans la distribution CompCert.

8. Lazy Code Motion



8. Lazy Code Motion



9. Allocation de registres

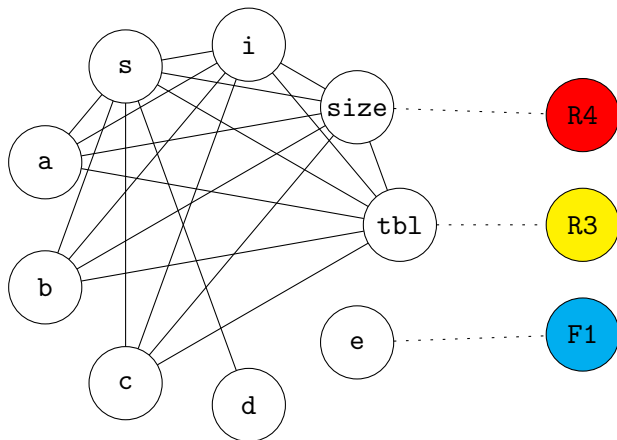
- En entrée : RTL.
Des variables, en quantité arbitraire.
- En sortie : LTL.
Un nombre fixé de registres hardware.
Un nombre arbitraire d'emplacements de pile.

Objectifs :

- Maximiser l'utilisation des registres hardware.
- Au passage : suppression du «code mort».
(Instructions pures dont les résultats sont inutilisés.)
- Au passage : réduction des *moves* $x = y$.
(On essaye de placer x et y dans le même registre.)

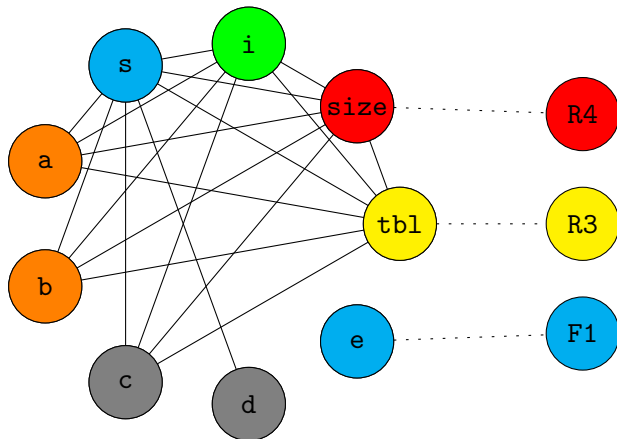
Technique utilisée : coloriage d'un graphe d'interférences (Chaitin) + heuristique de coloriage avec préférences (Appel et George).

9. Allocation de registres — Le graphe d'interférence



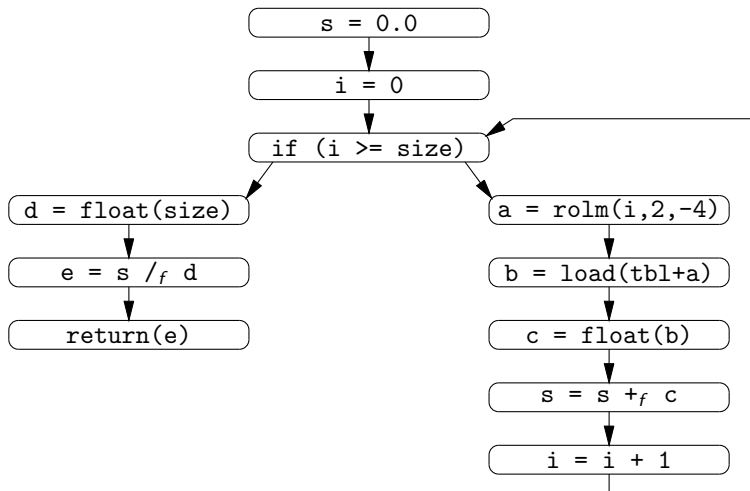
(Arc plein entre x et $y = x$ et y sont vivants simultanément en un point du programme $\Rightarrow x$ et y ne peuvent pas partager le même registre.)

9. Allocation de registres — Le graphe d'interférence

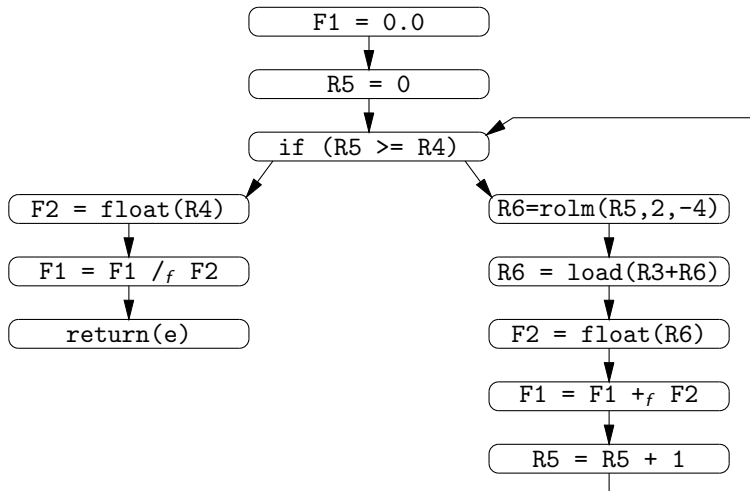


(Arc plein entre x et $y = x$ et y sont vivants simultanément en un point du programme $\Rightarrow x$ et y ne peuvent pas partager le même registre.)

9. Allocation de registres — Réécriture du CFG

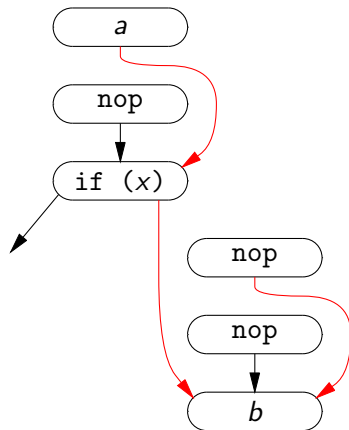
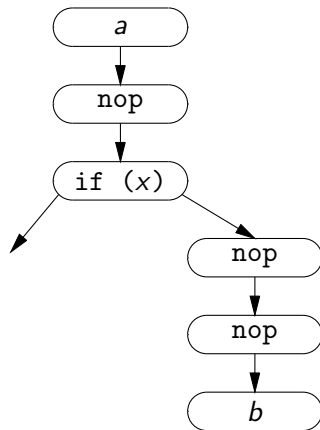


9. Allocation de registres — Réécriture du CFG



10. Branch tunneling

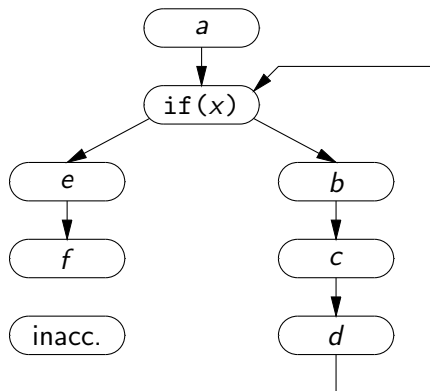
«Raccourcir» les arcs du CFG qui passent à travers des nœuds nop.



11. Linéarisation du contrôle

Traduction du CFG en contrôle de type «assembleur» :
instructions en séquence + étiquettes + branchements.

Au passage : suppression du code inaccessible.



```

a
L1 : if (!x) goto L2
    b
    c
    d
    goto L1
L2 : e
    f
  
```

12. Spilling, reloading, conventions d'appel

Les instructions arithmétiques ne peuvent opérer que sur des registres.
 ⇒ Insertion de rechargements (reloads) et déchargements (spills) autour des instructions dont les opérands ont été placés en pile.

```

stk(0) = stk(8) + F1
F12 = stk(8)      // reload
F12 = F12 + F1
stk(0) = F12      // spill
  
```

Insertion de moves autour des appels de fonction pour respecter les conventions d'appel (arguments et résultats sont passés dans tel et tel registres en fonction du type de la fonction appelée).

```

F5 = call "f"(R3, stk(12))
R4 = stk(12)
call "f"      // (R3, R4) -> F1
F5 = F1
  
```

13. Construction du bloc d'activation (stack frame)

Réserver de la place pour :

- Les variables qui ont été allouées dans la pile.
- Sauvegarder les registres *callee save*.
- Chaîner les blocs d'activation.

Réécrire les rechargements et déchargements comme des accès mémoire dans la pile.

```
F12 = stk(8)
```

```
F12 = F12 + F1
```

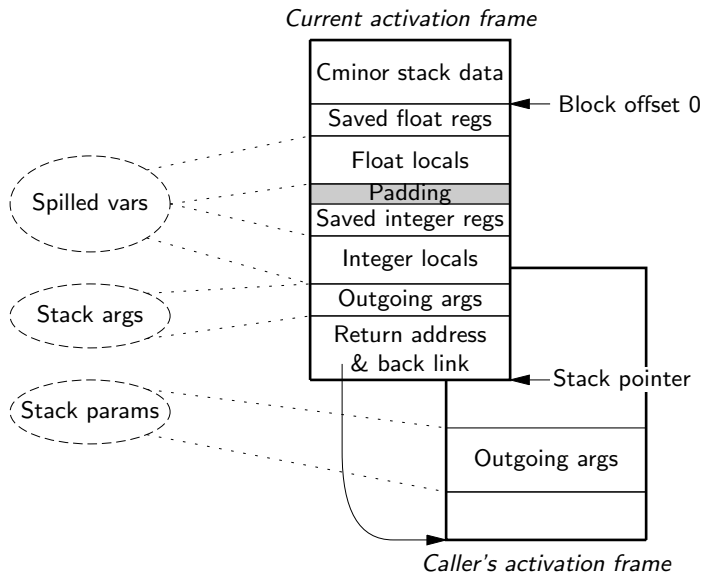
```
stk(0) = F12
```

```
F12 = load(int64, SP + 40)
```

```
F12 = F12 + F1
```

```
store(int64, SP + 32, F12)
```

Structure du bloc d'activation



14. Réordonnancement d'instructions

Réordonner les instructions indépendantes dans les blocs de base (étendus) pour mieux exploiter le parallélisme d'instructions du processeur (pipeline, superscalaire).

```
R5 = rolm(R3,3,-8)
```

```
R5 = R3 + R5
```

```
F3 = F1 +f F2
```

```
store(R5, F3)
```

```
F3 = F1 +f F2
```

```
R5 = rolm(R3,3,-8)
```

```
R5 = R3 + R5
```

```
store(R5, F3)
```

Utilise l'approche de validation a posteriori avec un validateur prouvé.

Pas encore intégré dans la distribution CompCert.

15. Production de code PowerPC

Expanser chaque instruction en une séquence d'instructions PowerPC.

Tout en contournant diverses irrégularités du jeu d'instructions PowerPC.

```
if (R3 < R4) goto L1           cmpw cr0, R3, R4
                                bf    0, L1

R3 = R3 + 123456               addis R3, R3, 2
                                addi  R3, R3, -7616
```

16. Production d'un exécutable

- 1 Impression de l'AST PPC en syntaxe concrète assembleur.
- 2 Assemblage par l'assembleur du système.
- 3 Édition de liens par le linker du système.

(Transformations non vérifiées !)

Happy end !

```
double average(int * tbl, int size)
{
    double s = 0; int i;
    for (i = 0; i < size; i++) s += tbl[i];
    return s / size;
}
```

```
94 21 ff e0 7d 88 02 a6 91 81 00 0c 3d 80 00 00
c8 2c 20 58 38 a0 00 00 7c 05 20 00 40 80 00 3c
54 a6 10 3a 7c c3 30 2e 3d 80 43 30 95 81 ff f8
3d 86 80 00 91 81 00 04 3d 80 00 00 c9 ac 20 60
c8 61 00 00 38 21 00 08 fc 63 68 28 fc 21 18 2a
38 a5 00 01 4b ff ff c4 3d 80 43 30 95 81 ff f8
3d 84 80 00 91 81 00 04 3d 80 00 00 c9 ac 20 68
c8 41 00 00 38 21 00 08 fc 42 68 28 fc 21 10 24
81 81 00 0c 7d 88 03 a6 80 21 00 00 4e 80 00 20
```