Vérification formelle d'un compilateur C réaliste

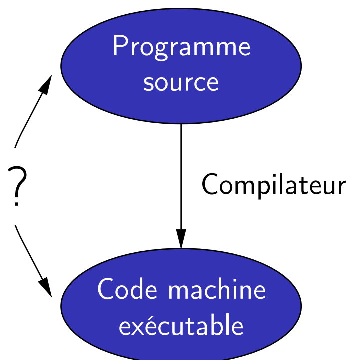
Sandrine Blazy

IRISA - INRIA Rennes Bretagne Atlantique, EPI CELTIQUE

Séminaire ENS-DIT, Ker Lann, 06-04-2010

UMR IRISA

Faites-vous confiance à votre compilateur ?



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

Un exemple de compilation optimisante

```
double dotproduct(int n, double * a, double * b)
{
    double dp = 0.0;
    int i;
    for (i = 0; i < n; i++) dp += a[i] * b[i];
    return dp;
}
```

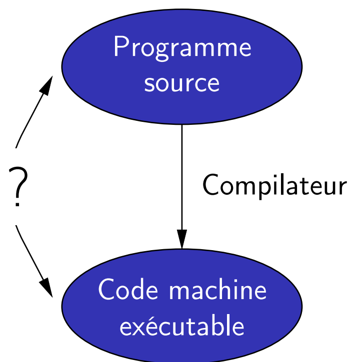
Compilé pour le processeur Alpha avec toutes les optimisations et décompilé à la main en C ...

```
double dotproduct(int n, double * a, double * b)
{
     double dp, a0, a1, a2, a3, b0, b1, b2, b3;
     double s0, s1, s2, s3, t0, t1, t2, t3;
     int i, k;
     dp = 0.0;
     if (n <= 0) goto L5;
     s0 = s1 = s2 = s3 = 0.0;
     i = 0; k = n - 3;
     if (k <= 0 || k > n) goto L19;
     i = 4; if (k <= i) goto L14;
     a0 = a[0];  b0 = b[0];  a1 = a[1];  b1 = b[1];
     i = 8; if (k <= i) goto L16;
L17 : a2 = a[2];  b2 = b[2];  t0 = a0 * b0;
     a3 = a[3];  b3 = b[3];  t1 = a1 * b1;
     a0 = a[4];  b0 = b[4];  t2 = a2 * b2;  t3 = a3 * b3;
     a1 = a[5];  b1 = b[5];
     s0 += t0; s1 += t1;  s2 += t2;  s3 += t3;
     a += 4; i += 4; b += 4;
     prefetch(a + 20); prefetch(b + 20);
     if (i < k) goto L17;
L16 : s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1];
L18 : s0 += a0 * b0; s1 += a1 * b1; s2 += a[2] * b[2]; s3 += a[3] * b[3];
     a += 4; b += 4;
     dp = s0 + s1 + s2 + s3;
     if (i >= n) goto L5;
L19 : dp += a[0] * b[0];
     i += 1; a += 1; b += 1;
     if (i < n) goto L19;
L5 :  return dp;
L14 : a0 = a[0]; b0 = b[0]; a1 = a[1]; b1 = b[1]; goto L18;
}
```

```
    i = 0 ; k = n - 3 ;
    if (k <= 0 || k > n) goto L19 ;
    i = 4 ; if (k <= i) goto L14 ;
    a0 = a[0] ;  b0 = b[0] ;  a1 = a[1] ;  b1 = b[1] ;
    i = 8 ; if (k <= i) goto L16 ;
L17 : a2 = a[2] ;  b2 = b[2] ;  t0 = a0 * b0 ;
    a3 = a[3] ;  b3 = b[3] ;  t1 = a1 * b1 ;
    a0 = a[4] ;  b0 = b[4] ;  t2 = a2 * b2 ;  t3 = a3 * b3 ;
    a1 = a[5] ;  b1 = b[5] ;
    s0 += t0 ; s1 += t1 ;  s2 += t2 ;  s3 += t3 ;
    a += 4 ; i += 4 ; b += 4 ;
    prefetch(a + 20) ; prefetch(b + 20) ;
    if (i < k) goto L17 ;
L16 : s0 += a0 * b0 ; s1 += a1 * b1 ; s2 += a[2] * b[2] ; s3
    a += 4 ; b += 4 ;
    a0 = a[0] ; b0 = b[0] ; a1 = a[1] ; b1 = b[1] ;
L18 : s0 += a0 * b0 ; s1 += a1 * b1 ; s2 += a[2] * b[2] ; s3
    a += 4 ; b += 4 ;
    dp = s0 + s1 + s2 + s3 ;
```

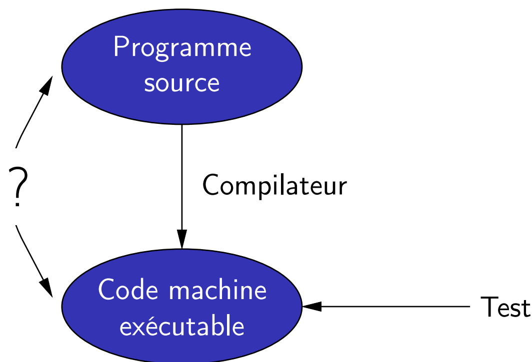
```
double dotproduct(int n, double * a, double * b)
{
     double dp, a0, a1, a2, a3, b0, b1, b2, b3 ;
     double s0, s1, s2, s3, t0, t1, t2, t3 ;
     int i, k ;
     dp = 0.0 ;
     if (n <= 0) goto L5 ;
     s0 = s1 = s2 = s3 = 0.0 ;
     i = 0 ; k = n - 3 ;
     if (k <= 0 || k > n) goto L19 ;
     i = 4 ; if (k <= i) goto L14 ;
     a0 = a[0] ;  b0 = b[0] ;  a1 = a[1] ;  b1 = b[1] ;
     i = 8 ; if (k <= i) goto L16 ;
L17 : a2 = a[2] ;  b2 = b[2] ;  t0 = a0 * b0 ;
     a3 = a[3] ;  b3 = b[3] ;  t1 = a1 * b1 ;
     a0 = a[4] ;  b0 = b[4] ;  t2 = a2 * b2 ;  t3 = a3 * b3 ;
     a1 = a[5] ;  b1 = b[5] ;
     s0 += t0 ; s1 += t1 ;  s2 += t2 ;  s3 += t3 ;
     a += 4 ; i += 4 ; b += 4 ;
     prefetch(a + 20) ; prefetch(b + 20) ;
     if (i < k) goto L17 ;
L16 : s0 += a0 * b0 ; s1 += a1 * b1 ; s2 += a[2] * b[2] ; s3 += a[3] * b[3] ;
     a += 4 ; b += 4 ;
     a0 = a[0] ; b0 = b[0] ; a1 = a[1] ; b1 = b[1] ;
L18 : s0 += a0 * b0 ; s1 += a1 * b1 ; s2 += a[2] * b[2] ; s3 += a[3] * b[3] ;
     a += 4 ; b += 4 ;
     dp = s0 + s1 + s2 + s3 ;
     if (i >= n) goto L5 ;
L19 : dp += a[0] * b[0] ;
     i += 1 ; a += 1 ; b += 1 ;
     if (i < n) goto L19 ;
L5 :  return dp ;
L14 : a0 = a[0] ; b0 = b[0] ; a1 = a[1] ; b1 = b[1] ; goto L18 ;
}
```

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 ?

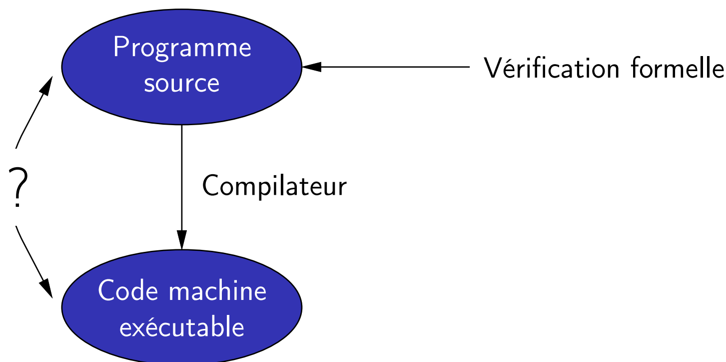


Logiciel critique certifié par test systématique :

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

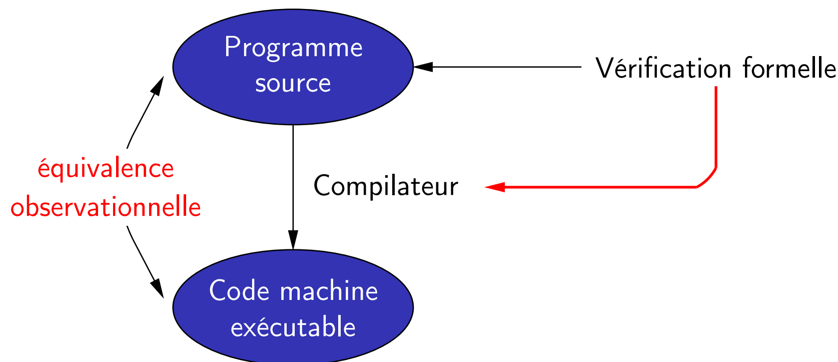
Les bugs du compilateur sont détectés en même temps que ceux du programme.

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.

Plan

1. Introduction : faites-vous confiance à votre compilateur ?
2. Compilation formellement vérifiée
3. Le projet Compcert
4. Sémantique formelle, équivalence observationnelle
5. Un exemple de vérification : la passe d'allocation de registres
6. Conclusion

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 (code source mal formé, ou dépassement de capacité).

La propriété à établir formellement

Établir formellement une propriété $Prop(S, C)$ du code compilé C et du code source S, comme par exemple :

1. S et C sont observationnellement équivalents ;
2. si S a une sémantique bien définie, S et C sont observationnellement équivalents ;
3. si S a une sémantique bien définie et satisfait la spec *Spec*, alors C satisfait *Spec* ;
4. si S est sûr vis-à-vis du typage et de la mémoire, C l'est aussi.

Approche 1 : preuve de compilateur

Compilateur certifié

Le compilateur est spécifié par une fonction totale

$$Comp : Source \rightarrow Code + \texttt{Error}$$

La propriété

$$\forall S, C, b, \quad Comp(S) = C \implies S \equiv C \text{ (équivalence observationnelle)}$$

est démontrée à l'aide d'un assistant de preuve.

Remarque : structures de données complexes + algorithmes récursifs $\Rightarrow$ la preuve est nécessairement interactive.

Approche 2 : validation de traducteurs
(A. Pnueli et al ; G. Necula ; X. Rival)

Les résultats de la compilation sont validés *a posteriori*.

$$\begin{array}{rcl} Comp & : & Source \rightarrow Code + \texttt{Error} \\ Validator & : & Source \times Code \rightarrow \texttt{bool} \end{array}$$

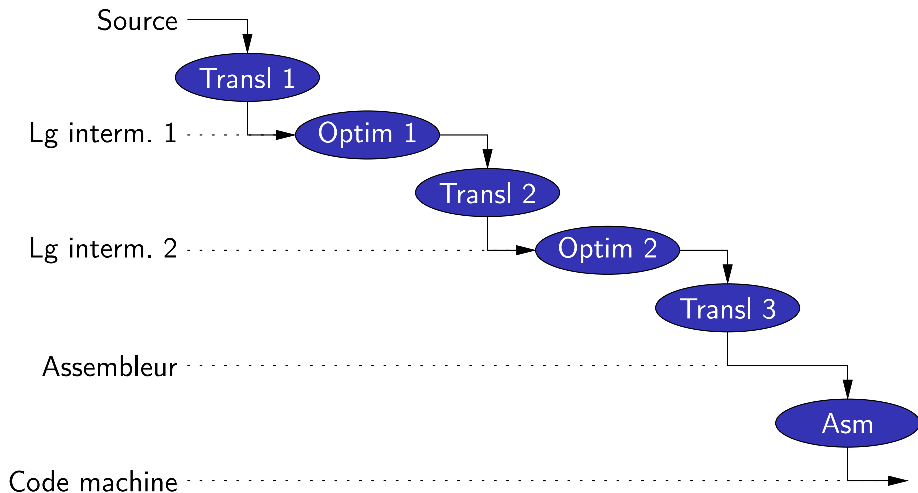
Si $Comp(S) = C$ et $Validator(S, C) = \texttt{true}$, la compilation est correcte. Sinon, erreur.

Il suffit de prouver que le vérificateur est correct :

$$\forall S, C, \quad Validator(S, C) = \texttt{true} \Rightarrow S \equiv C$$

Le compilateur n'a pas besoin d'être vérifié.

Décomposition en passes de compilation



Décomposition en passes de compilation

Si chaque passe de compilation préserve la sémantique, alors il en est de même pour leur composition !

En général, une passe de compilation peut être prouvée correcte indépendamment des autres passes.

Cependant, un sémantique formelle doit être définie pour chaque langage intermédiaire (et pas seulement pour les langages source et cible).

Pour chaque passe, nous pouvons

- soit la prouver correcte directement,
- soit la valider a posteriori et seulement prouver la correction du vérificateur.

Plan

1. Introduction : faites-vous confiance à votre compilateur ?
2. Compilation formellement vérifiée
3. Le projet Compcert
4. Sémantique formelle, équivalence observationnelle
5. Un exemple de vérification : la passe d'allocation de registres
6. Conclusion

Le compilateur certifié CompCert

compcert.inria.fr

Projet coordonné par Xavier Leroy

Compilateur réaliste utilisable pour le logiciel embarqué critique.

- Langage source = vaste sous-ensemble de C.
- Langage cible = assembleurs des processeurs PowerPC et ARM.
- Produit du code raisonnablement compact et efficace
  $\Rightarrow$ optimisations.
- Compilateur formellement vérifié en utilisant l'assistant de preuve Coq.

Le sous-ensemble Clight de C

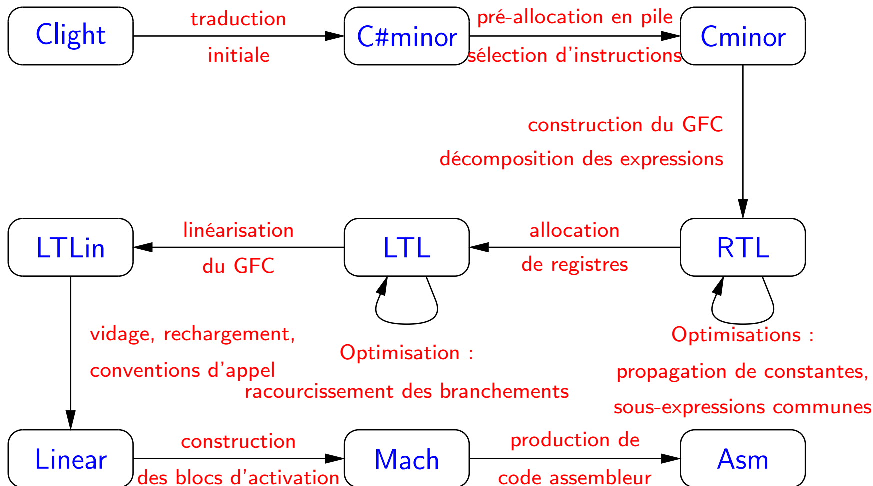
Traité :

- Types : entiers, flottants, tableaux, pointeurs, `struct`, `union`.
- Opérateurs : toute l'arithmétique du C, y compris l'arithmétique de pointeurs.
- Toutes les instructions C : `if/then/else`, boucles `while/do/for`, `switch`, `goto`.
- Fonctions, y compris les fonctions récursives et les pointeurs de fonctions.

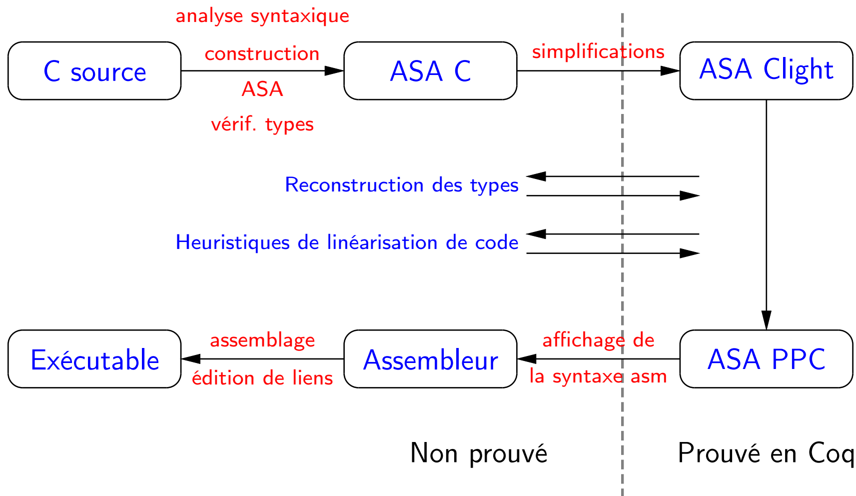
Non traité :

- Les types `long long` et `long double`.
- Fonctions à nombre variable d'arguments.
- Passage par valeur des `struct` et `union`.

La partie formellement vérifiée du compilateur



Le compilateur CompCert au complet



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.

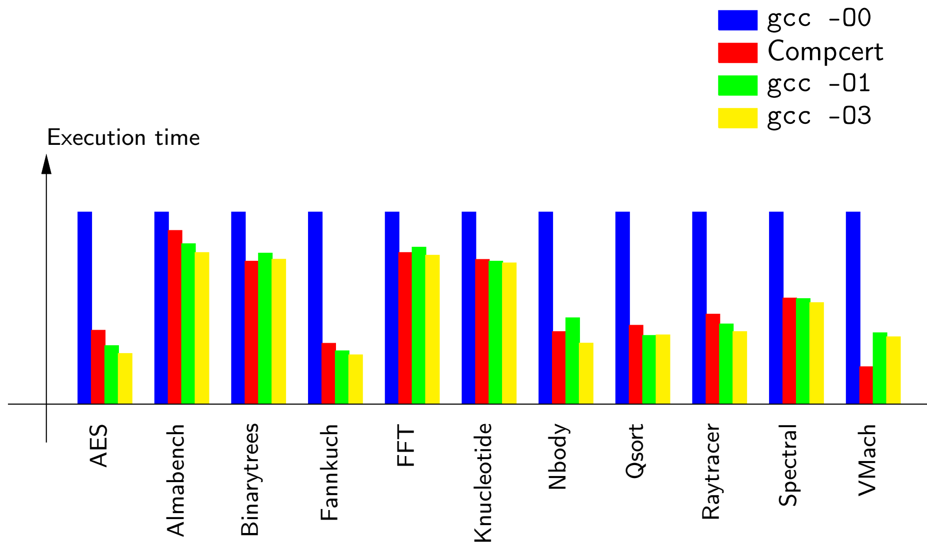
- Utilisation de monades pour traiter les erreurs et les états.
- Structures de données purement fonctionnelles (persistantes).

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

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

Probablement le plus gros programme extrait d'un développement Coq.

Performances du code généré



Plan

1. Introduction : faites-vous confiance à votre compilateur ?
2. Compilation formellement vérifiée
3. Le projet Compcert
4. **Sémantique formelle, équivalence observationnelle**
5. Un exemple de vérification : la passe d'allocation de registres
6. Conclusion

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

Chaque langage de CompCert est défini par une sémantique opérationnelle.

Évaluation des expressions : sémantique à grands pas

a.k.a. sémantique naturelle

Jugement d'évaluation :

$\rho \vdash$ objet syntaxique $\Rightarrow$ comportement observé, ρ'

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

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

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

Seuls les comportements attendus sont spécifiés.

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

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

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

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

$$\frac{i_1, \rho \rightarrow i_1', \rho'}{i_1; i_2, \rho \rightarrow i_1'; i_2, \rho'} \qquad \texttt{skip}; i_2, \rho \rightarrow i_2, \rho$$

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

$$\frac{\rho \vdash e \Rightarrow v \quad \texttt{is_true}(v)}{\texttt{while } (e)\{i\}, \rho \rightarrow i; \texttt{while } (e)\{i\}, \rho} \qquad \frac{\rho \vdash e \Rightarrow v \quad \texttt{is_false}(v)}{\texttt{while } (e)\{i\}, \rho \rightarrow \texttt{skip}, \rho}$$

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 $\texttt{skip}, \rho'$.

$$i, \rho \rightarrow i_1, \rho_1 \rightarrow \cdots \rightarrow \texttt{skip}, \rho'$$

2. Divergence : séquence infinie de transitions.

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

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

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

avec $i_n \neq \texttt{skip}$. (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» ?

P et P' se comportent «pareil» si :

- Si P termine sur l'état ρ, alors P' termine sur un état $\rho^{'}$, et $\rho, \rho^{'}$ 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

Solution 2

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

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

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

Solution 3

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 :

- $P \Downarrow \texttt{terminates}(t, n)$:
  terminaison, avec trace t d'entrées-sorties, et code de retour n (valeur renvoyée par la fonction `main`).
- $P \Downarrow \texttt{diverges}(T)$:
  divergence, avec trace T d'entrées-sorties (finie ou infinie).
- $P \not\Downarrow$:
  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 : $\texttt{terminates}(\epsilon, 42)$.

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

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

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

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

Plan

1. Introduction : faites-vous confiance à votre compilateur ?
2. Compilation formellement vérifiée
3. Le projet Compcert
4. Sémantique formelle, équivalence observationnelle
5. Un exemple de vérification : la passe d'allocation de registres
6. Conclusion

Le langage intermédiaire RTL

Langage à transfert de registres (code 3 adresses).

Le code d'une fonction est representé par un graphe de flot de contrôle :

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

`z = x +f y`	addition flottante
`i = i + 1`	addition de constante entière
`if (x > y)`	test et branchement conditionnel

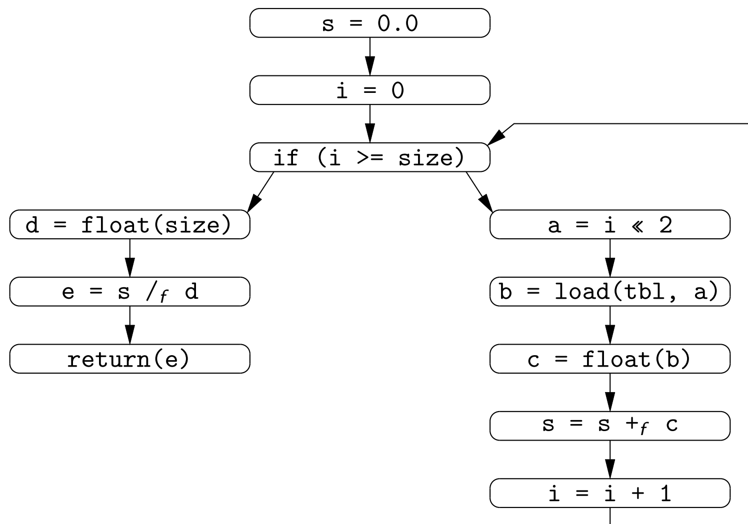
- Arc de I à J $\quad = \quad$ J est un successeur de I
  (J peut être exécuté juste après I).

Exemple : code source C

```
double average(int * tbl, int size)
{
    double s = 0 ;
    int i ;

    for (i = 0 ; i < size ; i++) s += tbl[i] ;
    return s / size ;
}
```

Example : le graphe RTL correspondant



Allocation de registres

Utilité : raffiner la notion de variable utilisée en tant qu'argument et résultat d'opérations RTL.

- RTL (avant allocation de registres) :
  quantité illimitée de variables.
- LTL (après allocation de registres) :
  nombre limité de registres machine ;
  nombre illimité d'emplacements de pile.

(L'insertion de code (vidage et rechargement) est effectuée par une passe ultérieure.)

Objectif : maximiser l'utilisation des registres.

Allocation de registres : approches

Approche naïve :

Attribuer les N registres machine aux N variables les plus utilisées ; placer les autres variables dans des emplacements de pile.

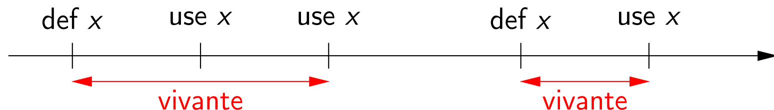
Meilleure approche :

Considérer que les registres machine peuvent être attribués à différentes variables, pourvu qu'elles ne soient pas utilisées simultanément.

Analyse de vivacité

Une variable x est vivante au point p si une instruction atteignable depuis p utilise x, et x n'est pas redéfinie entre temps.

Dans un code purement séquentiel, une variable devient vivante chaque fois qu'elle est définie. Elle meurt à l'issue de sa dernière utilisation.

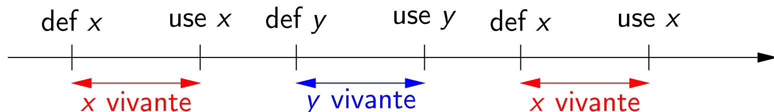


Si x meurt (n'est pas vivante) à un point de programme donné, la valeur de x en ce point n'a pas d'effet sur les résultats du calcul.

Utilisation des informations de vivacité pour l'allocation de registres

Deux variables x et y interfèrent si elles sont toutes les deux vivantes en un point de programme.

Si x et y n'interfèrent pas, elles peuvent partager un même registre ou un même emplacement de pile.



$\rightarrow$ Déterminer le nombre minimal de registres nécessaire au coloriage du graphe représentant la relation d'interférence.

$\rightarrow$ Si ce nombre est $\leq$ nombre de registres machine, nous obtenons une allocation de registres parfaite.

$\rightarrow$ Sinon, le coloriage est un bon point de départ pour déterminer quelles variables seront placées dans des registres.

Algorithme, 1 : Analyse de vivacité

Mise en place des équations de flot de données arrière :

$$L_{in}(p) = \mathtt{transf}(L_{out}(p), \mathtt{instr\text{-}at}(p))$$
$$L_{out}(p) = \bigcup \{ L_{in}(s) \mid s \text{ successeur de of } p \}$$

où, par exemple,

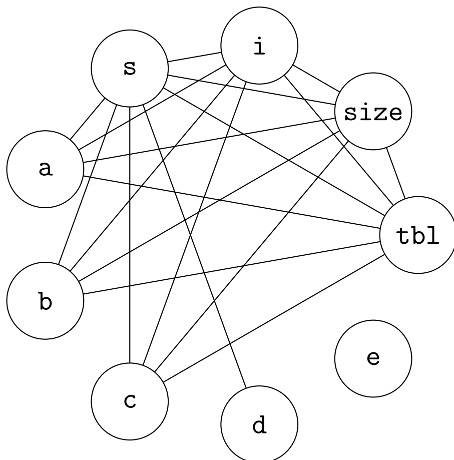
$$\mathtt{transf}(X, r := op(r_1, \ldots, r_n)) = (X \setminus \{r\}) \cup \{r_1, \ldots, r_n\}$$

Ces équations se résolvent par itération de point fixe (algorithme de Kildall).

Algorithme, 2 : construction du graphe d'interférences

Pour chaque instruction $p : \ r := \ldots$, ajouter des arcs entre r et $L_{out}(p) \setminus \{r\}$.

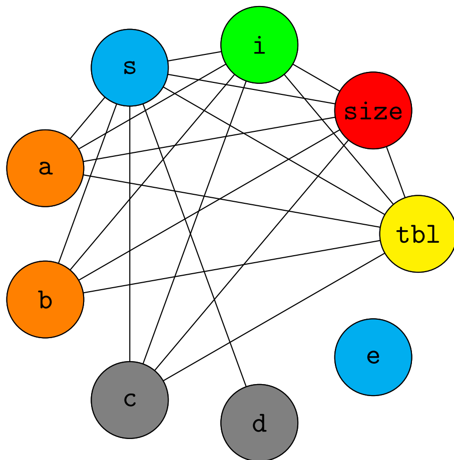
(+ Prise en compte des préférences.)



Algorithme, 3 : Coloriage du graphe d'interférences

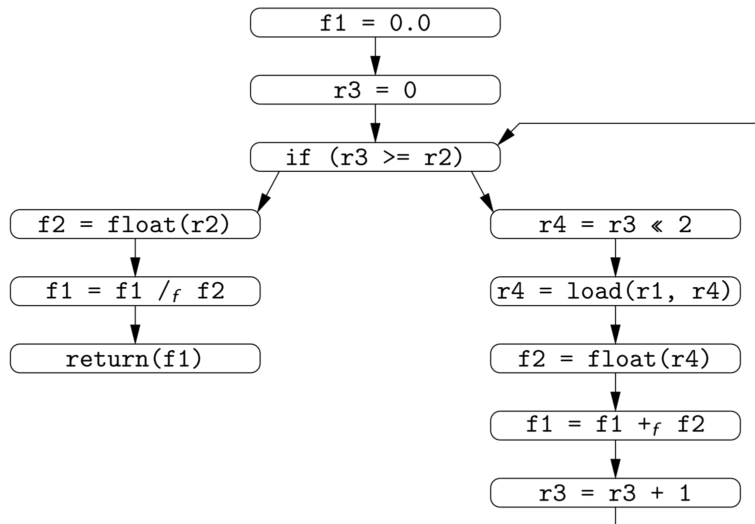
Construire une fonction ϕ : *Variable* $\rightarrow$ *Register* + *Stackslot* telle que $\phi(x) \neq \phi(y)$ si x et y interfèrent.

Nous utilisons l'heuristique de coloriage d'Appel-George.



Algorithme, 4 : Récriture du code

Remplacer toutes les variables x par leur couleur $\phi(x)$.



Quelles propriétés doivent être prouvées ?

Partie 1 : preuves des algorithmes

Analyse de vivacité : montrer (par induction sur le nombre d'itérations) que la correspondance L_{out} calculée par l'algorithme de Kildall satisfait les

$$L_{out}(p) \supseteq \texttt{transf}(L_{out}(s), \texttt{instr-at}(p)) \text{ si } s \text{ successeur de } p$$

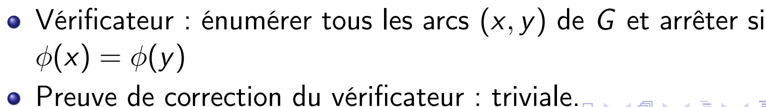
Construction du graphe d'interférences : montrer que le graphe final G contient tous les arcs attendus, e.g.

$$p : x := \ldots \;\wedge\; y \neq x \wedge y \in L_{out}(p) \Longrightarrow (x, y) \in G$$

Coloriage du graphe d'interférences : montrer que

$$(x, y) \in G \Longrightarrow \phi(x) \neq \phi(y)$$

Si nous utilisons la validation *a posteriori* :



- Vérificateur : énumérer tous les arcs (x, y) de G et arrêter si $\phi(x) = \phi(y)$
- Preuve de correction du vérificateur : triviale.

Que signifie “x est vivant au point p”, sémantiquement ?

Hmmm . . .

Que signifie “x meurt en p”, sémantiquement ?

Que le programme se comporte de façon identique, indépendamment de la valeur de x au point p.

Invariant

Soient E : variable $\rightarrow$ value les valeurs des variables au point p du programme initial. Soient R : location $\rightarrow$ value les valeurs des emplacements en mémoire au point p dans le programme transformé. E et R sont équivalents au point p (notation $p \vdash E \approx R$), ssi

$$E(x) = R(\phi(x)) \text{ pour tout } x \text{ vivant avant le point } p$$

Quelles propriétés doivent être prouvées ?

Partie 2 : preuve de préservation sémantique

Que signifie "x est vivant au point p", sémantiquement ?

Hmmm ...

Que signifie "x meurt en p", sémantiquement ?

Que le programme se comporte de façon identique, indépendamment de la valeur de x au point p.

Invariant

Soient E : variable $\rightarrow$ value les valeurs des variables au point p du programme initial. Soient R : location $\rightarrow$ value les valeurs des emplacements en mémoire au point p dans le programme transformé. E et R sont équivalents au point p (notation $p \vdash E \approx R$), ssi

$$E(x) = R(\phi(x)) \text{ pour tout } x \text{ vivant avant le point } p$$

Quelles propriétés doivent être prouvées ?

Partie 2 : preuve de préservation sémantique

Que signifie “x est vivant au point p”, sémantiquement ?

Hmmm ...

Que signifie “x meurt en p”, sémantiquement ?

Que le programme se comporte de façon identique, indépendamment de la valeur de x au point p.

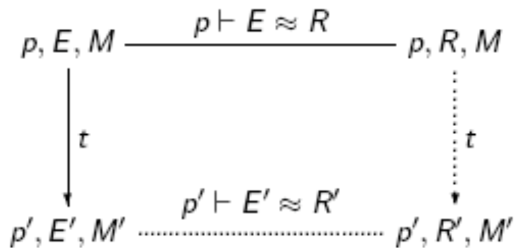
Invariant

Soient E : variable $\rightarrow$ value les valeurs des variables au point p du programme initial. Soient R : location $\rightarrow$ value les valeurs des emplacements en mémoire au point p dans le programme transformé. E et R sont équivalents au point p (notation $p \vdash E \approx R$), ssi

$$E(x) = R(\phi(x)) \textit{ pour tout } x \textit{ vivant avant le point } p$$

Prouver qu'une transformation de programme préserve la sémantique

Montrer un diagramme de simulation de la forme suivante



Hypothèses : à gauche, transition dans le code initial ; en haut, l'invariant (équivalence entre registres) avant la transition.

Conclusions : une transition dans le code transformé ; en bas, l'invariant après la transition.

Plan

1. Introduction : faites-vous confiance à votre compilateur ?
2. Compilation formellement vérifiée
3. Le projet Compcert
4. Sémantique formelle, équivalence observationnelle
5. Un exemple de vérification : la passe d'allocation de registres
6. Conclusion

Conclusion

Vérifier formellement un compilateur réaliste est possible.

De plus, les assistants à la preuve tels que Coq sont adaptés à cette tâche.

Et ensuite ?

Améliorations de Compcert

- Traiter un plus vaste sous-ensemble de C.
- Tester le compilateur sur de véritables codes.
  Expérience en cours chez Airbus
- Proposer davantage d'optimisations, prouver leur correction.
  (e.g. global value numbering, en utilisant l'approche validation *a posteriori*)
- Cibler d'autres processeurs que PowerPC et ARM.
- Prouver la préservation sémantique de programmes concurrents.
  (Difficile ! Nécessite de se restreindre à des programmes source « race-free ».)
  → projet Concurrent Cminor (Andrew Appel, Université de Princeton)

Perspectives

Vérifier d'autres outils intervenant dans la production et la vérification de logiciels critiques.