

Vérification formelle d'un front-end pour un compilateur C

Projet ANR ARA SSIA CompCert (2005-2008)

Sandrine Blazy, Xavier Leroy

CEDRIC-ENSIIE et INRIA Rocquencourt

Journée PPF-LS, 7 juin 2007



Plan

- 1 Vérification formelle de compilateurs
- 2 Compilateur CompCert
- 3 Un front-end pour le compilateur CompCert
 - CIL
 - Sémantique du langage Clight
 - 1^{ere} passe : de Clight à Csharpminor
 - Correction de la traduction
 - 2^e passe : de Csharpminor à Cminor
- 4 Conclusion

La compilation

Au sens large : toute traduction automatique d'un langage informatique vers un autre.

Au sens restreint : traduction **efficace** (“optimisante”) d'un langage **source** (compréhensible par les programmeurs) vers un langage **machine** (compréhensible par le *hardware*).

Un domaine mature :

- 50 ans ! (Fortran I : 1957)
- Énorme corpus d'algorithmes (optimisations).
- Nombreux compilateurs qui effectuent des transformations très subtiles.

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 Alpha et retranscrit à peu près en C ...

```

double dotproduct(int n, double a[], double b[]) {
    dp = 0.0;
    if (n <= 0) goto L5;
    r2 = n - 3; f1 = 0.0; r1 = 0; f10 = 0.0; f11 = 0.0;
    if (r2 > n || r2 <= 0) goto L19;
    prefetch(a[16]); prefetch(b[16]);
    if (4 >= r2) goto L14;
    prefetch(a[20]); prefetch(b[20]);
    f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
    r1 = 8; if (8 >= r2) goto L16;
L17 : f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
    f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
    f12 = a[4]; f16 = f18 * f16;
    f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
    f11 += f17; r1 += 4; f10 += f15;
    f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
    f1 += f16; dp += f19; b += 4;
    if (r1 < r2) goto L17;
L16 : f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f12 * f13;
    f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
    f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10 + f15;
    a += 4; b += 4; f14 = a[8]; f15 = b[8];
    f11 += f22; f1 += f21; dp += f24;
L18 : f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
    f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27 * f26;
    a += 4; f28 = f29 * f28; b += 4;
    f10 += f14; f11 += f12; f1 += f26;
    dp += f28; dp += f1; dp += f10; dp += f11;
    if (r1 >= n) goto L5;
L19 : f30 = a[0]; f18 = b[0]; r1 += 1; a += 8; f18 = f30 * f18; b += 8;
    dp += f18;
    if (r1 < n) goto L19;
L5 : return dp;
L14 : f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1]; goto L18;
}

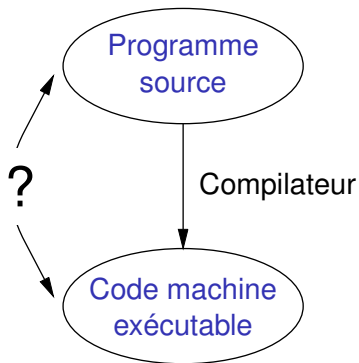
```

```

if (4 >= r2) goto L14;
prefetch(a[20]); prefetch(b[20]);
f12 = a[0]; f13 = b[0]; f14 = a[1]; f15 = b[1];
r1 = 8; if (8 >= r2) goto L16;
L17 : f16 = b[2]; f18 = a[2]; f17 = f12 * f13;
      f19 = b[3]; f20 = a[3]; f15 = f14 * f15;
      f12 = a[4]; f16 = f18 * f16;
      f19 = f29 * f19; f13 = b[4]; a += 4; f14 = a[1];
      f11 += f17; r1 += 4; f10 += f15;
      f15 = b[5]; prefetch(a[20]); prefetch(b[24]);
      f1 += f16; dp += f19; b += 4;
      if (r1 < r2) goto L17;
L16 : f15 = f14 * f15; f21 = b[2]; f23 = a[2]; f22 = f1
      f24 = b[3]; f25 = a[3]; f21 = f23 * f21;
      f12 = a[4]; f13 = b[4]; f24 = f25 * f24; f10 = f10
      a += 4; b += 4; f14 = a[8]; f15 = b[8];
      f11 += f22; f1 += f21; dp += f24;
L18 : f26 = b[2]; f27 = a[2]; f14 = f14 * f15;
      f28 = b[3]; f29 = a[3]; f12 = f12 * f13; f26 = f27
      a += 4; f28 = f29 * f28; b += 4;
      f10 += f14; f11 += f12; f1 += f26;

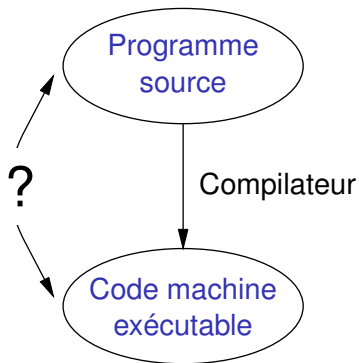
```

Faites-vous confiance à votre compilateur ?



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

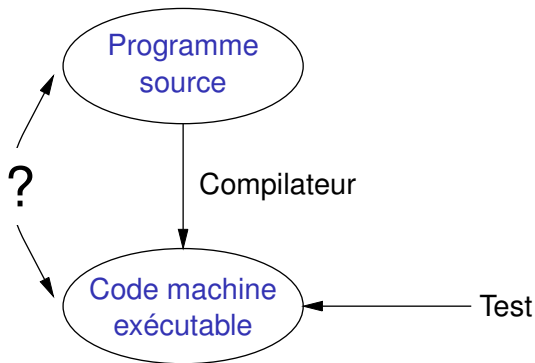
Faites-vous confiance à votre compilateur ?



Logiciel non critique :

Les bugs du compilateur sont négligeables devant ceux du logiciel.

Faites-vous confiance à votre compilateur ?

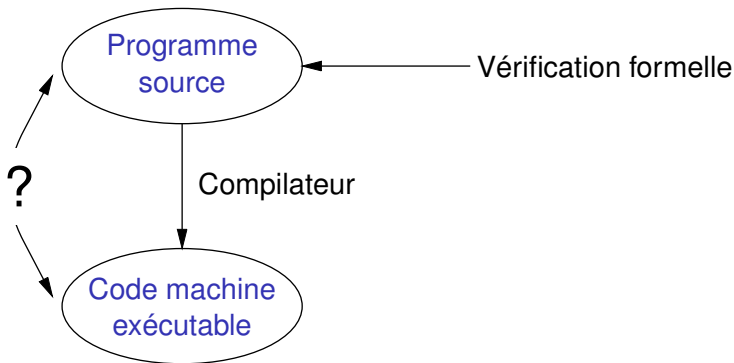


Logiciel critique certifié par test systématique :

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

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

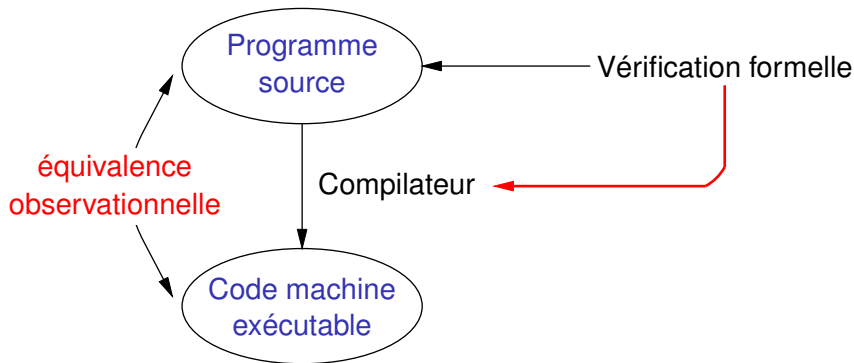
Faites-vous confiance à votre compilateur ?



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.

Un maillon faible

Dans le monde des méthodes formelles, le compilateur est un “maillon faible” entre un programme source vérifié formellement et un microprocesseur (partiellement) vérifié formellement.

Non-solutions utilisées dans l'industrie :

- Compiler sans aucune optimisation.
- Revues manuelles du code assembleur produit.

La vraie solution :

- Vérifier le compilateur lui-même en lui appliquant des méthodes formelles.

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 la même sémantique que S à équivalence observationnelle près.*

État de l'art

- **Compilateur certifié**

$CComp : Source \rightarrow option(Code)$

$CComp$ est certifié si

$\forall S, C, CComp(S) = Some(C) \Rightarrow Prop(S, C)$

- **Compilateur certifiant (Code auto-certifié)**

$PPC : Source \rightarrow option(Code \times Proof)$

Si $PPC(S) = Some(C, \pi)$ et $\pi \vdash Prop(S, C)$, la compilation est correcte.

- **Validation *a posteriori* Compilateur + vérificateur indépendant**

$Comp : Source \rightarrow option Code$

$Verif : Source \times Code \rightarrow bool$

Le vérificateur est supposé certifié :

$\forall S, C, Verif(S, C) = true \Rightarrow Prop(S, C)$

Si $Comp(S) = Some(C)$ et $Verif(S, C) = true$, la compilation est correcte.

Méthodologie

- Découpage du compilateur en passes de transformations successives, chacune étant certifiée.
- Nécessite des sémantiques formelles pour tous les langages intermédiaires.
- Chaque passe peut être soit certifiée directement, soit effectuée par du code non certifié + vérification des résultats par un vérificateur certifié.

Plan

- 1 Vérification formelle de compilateurs
- 2 **Compilateur CompCert**
- 3 Un front-end pour le compilateur CompCert
 - CIL
 - Sémantique du langage Clight
 - 1^{ere} passe : de Clight à Csharpminor
 - Correction de la traduction
 - 2^e passe : de Csharpminor à Cminor
- 4 Conclusion

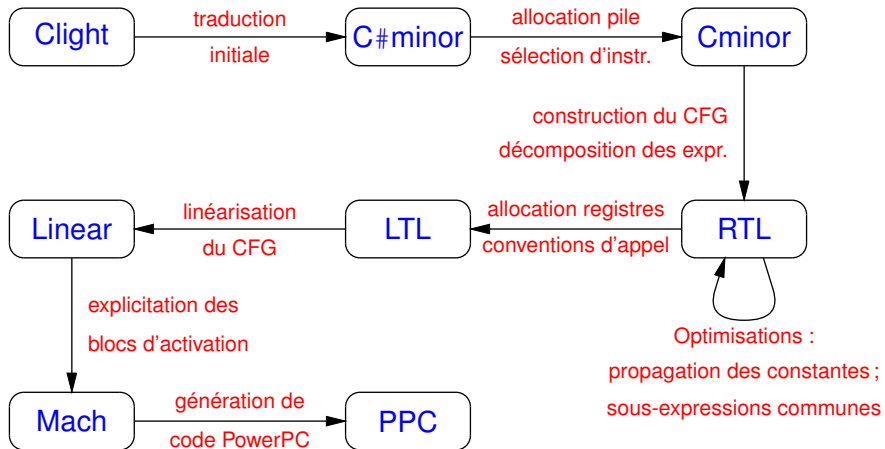
Le projet CompCert : objectifs généraux

Développer et vérifier en Coq 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
⇒ optimisations.

Améliorer les outils et techniques (de preuve, de compilation, de sémantique opérationnelle) nécessaires à cet effort.

Diagramme du compilateur Compcert



Vérifié en Coq

La preuve de correction du compilateur (préservation de la sémantique) est faite entièrement sur machine, avec l'assistant de preuve Coq.

```
Theorem transf_c_program_correct :  
  forall prog tprog trace n,  
    transf_c_program prog = Some tprog ->  
    Csem.exec_program prog trace (Vint n) ->  
    PPC.exec_program tprog trace (Vint n).
```

Environ 40000 lignes de Coq :

13%	8%	22%	50%	7%
Code	Sém.	Enoncés	Preuves	Autres

Programmé en Coq

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

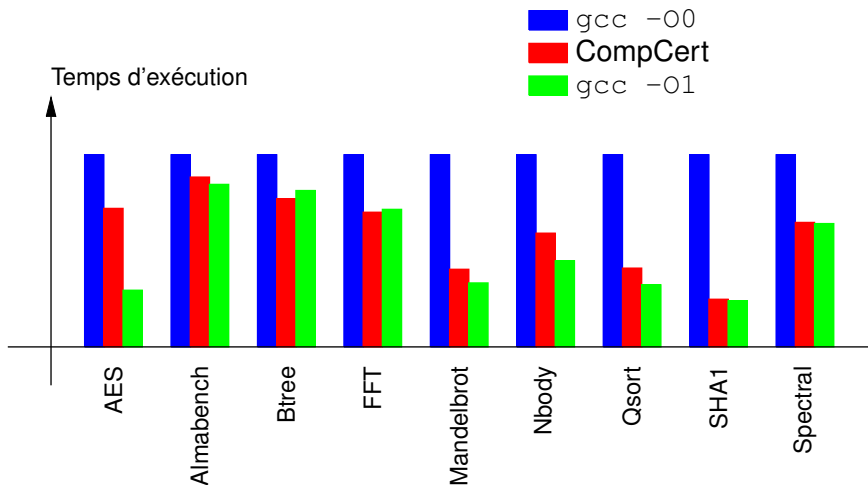
- Utilisation de monades pour traiter les erreurs et les états.
- Structures de données purement fonctionnelles (persistantes).

(4500 lignes de Coq + 1300 lignes de code Caml non vérifié.)

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

Sans doute le plus gros programme extrait d'un développement Coq.

Performances du code produit



Plan

- 1 Vérification formelle de compilateurs
- 2 Compilateur CompCert
- 3 Un front-end pour le compilateur CompCert**
 - CIL
 - Sémantique du langage Clight
 - 1^{ere} passe : de Clight à Csharpminor
 - Correction de la traduction
 - 2^e passe : de Csharpminor à Cminor
- 4 Conclusion

- Un analyseur très complet pour ISO C + extensions GNU, MS.
→ arbre d'analyse.
- Typage, simplifications et mise en forme.
→ arbre de syntaxe abstraite copieusement annoté.
- Outils pour l'analyse statique et la transformation.
(Calcul du CFG, visiteurs, pretty-printer, etc)

CIL : Exemples de simplifications

Expressions sans effets de bord :

```
return x++ + f(x); ---> tmp1 = x; x = x + 1;
                        tmp2 = f(x); return tmp1 + tmp2;
```

Explicitation des casts implicites :

```
short s; double d; ---> short s; double d;
s += d;                s = (short) ((double) s + d);
```

Élimination des variables locales à un bloc :

```
int f(int x) {          ---> static int x_0;
    static int x = 3;   int f(int x_1) {
    { int x; ... }      int x_2; ...
}                       }
```


CIL : Exemples de simplifications

Mise au carré des déclarations de `struct` et `union` :

```
struct foo {          ---> struct bar { int x; double y; };
  struct bar {
    int x; double y;  struct foo {struct bar b; char z;};
  } b;
  char z;              struct foo t;
} t;
```

Normalisation des initialisations :

```
int x[] = { 1, 2, 3 }; ---> int x[3] = { 1, 2, 3 };

int x[3] = { 42 };          int x[3] = { 42, 0, 0 };
```

CIL : Exemples de simplifications indésirables

Certaines simplifications de CIL ont été désactivées, essentiellement celles qui produisent des `goto`.

```
for (i = 0; i < 5; i++) ----> i = 0;
{
    if (i == 5) continue;
    if (i == 4) break;
    i += 2;
}
while (i < 5) {
    if (i == 5) goto __Cont;
    if (i == 4) break;
    i = i + 2;
__Cont :
    i = i + 1;
}
```

Le langage Clight

Types : tout ISO C sauf `long long`, `long double` et `typedef`.
`struct` et `union` représentés de manière structurelle.

Expressions : tout sauf les affectations.
Expressions annotées par leurs types.

Statements : affectations ; contrôle structuré.
Pas de `goto` ; `switch` limité.

Struct et unions : pas d'affectations directes ni de passage par valeur.
(Passage par référence OK.)

Programmes : définitions de variables globales avec initialisations ;
définitions de fonctions ; déclarations de fonctions externes.

Le puzzle des structures récursives

```
struct intlist {  
    int hd;  
    struct intlist * tl; }  
  
sizeof(struct intlist) = 8  
OK
```

```
struct intlist {  
    int hd;  
    struct intlist tl; }  
  
sizeof (struct intlist) = ∞  
Incorrect
```

Représentation structurelle naïve : $\tau ::= \dots \text{struct} \dots x_i : \tau_i \dots$

Termes finis (inductifs) : pas assez expressifs.

Termes infinis (coinductifs) : comment définir `sizeof` ?

Représentation nominale : Il faut propager l'environnement Γ partout.
`sizeof` n'est pas trivial à définir (pas récursif structurel).

Prise en compte des comportements non spécifiés

Standard ISO C : sémantique informelle, laissant certains comportements non spécifiés.

Certains de ces comportements ont été entièrement spécifiés dans CompCert :

- Ordre d'évaluation des expressions (gauche à droite).
- Tailles et valeurs des types intégraux.

D'autres sont rejetés par la sémantique :

- Débordements de tableaux.
- Division par zéro, etc.
- Observation au niveau des bits de la représentation des nombres et des pointeurs.

⇒ Un raffinement du C ISO.

Semantique formelle de Clight

Sémantique naturelle à grands pas avec traces,
opérant sur des termes annotés par des types.

$G, E \vdash$ objet syntaxique, $M \Rightarrow$ objet sémantique, trace, M'

Semantique formelle de Clight

Sémantique naturelle à grands pas avec traces,
opérant sur des termes annotés par des types.

$G, E \vdash$ objet syntaxique, $M \Rightarrow$ objet sémantique, trace, M'

M, M'	états mémoire	memory block reference \rightarrow bounds \times (offset \rightarrow kind \rightarrow value)
E	environnement local	identifiant \rightarrow block reference
G	environnement global	identifiant \rightarrow block reference \times function reference \rightarrow function body

Semantique formelle de Clight

Sémantique naturelle à grands pas avec traces,
opérant sur des termes annotés par des types.

$G, E \vdash$ objet syntaxique, $M \Rightarrow$ objet sémantique, trace, M'

Expressions en position de valeur droite :

$\text{expr} : \text{type} \Rightarrow \text{value} = \left\{ \begin{array}{l} \text{undef} \\ 32\text{-bit integer} \\ 64\text{-bit float} \\ \text{pointer} = (\text{block reference}, \text{byte offset}) \end{array} \right.$

Semantique formelle de Clight

Sémantique naturelle à grands pas avec traces,
opérant sur des termes annotés par des types.

$$G, E \vdash \text{objet syntaxique}, M \Rightarrow \text{objet sémantique}, \text{trace}, M'$$

Expressions en position de valeur gauche :

$\text{expr} : \text{type} \Rightarrow \text{location} = (\text{block reference}, \text{byte offset})$

Semantique formelle de Clight

Sémantique naturelle à grands pas avec traces,
opérant sur des termes annotés par des types.

$G, E \vdash$ objet syntaxique, $M \Rightarrow$ objet sémantique, trace, M'

Instructions :

$$\text{stmt} \Rightarrow \text{outcome} = \begin{cases} \text{Normal} \\ \text{Break} \\ \text{Continue} \\ \text{Return} \\ \text{Return}(\text{value}) \end{cases}$$

48 règles d'inférence, représentées en Coq par des constructeurs de prédicats inductifs.

Les traces d'exécution

Représentent les interactions entre le programme et le monde extérieur (entrées/sorties).

Listes finies d'événements d'E/S :

$ev ::= \{ \begin{array}{ll} id : ident & \text{fourni par le programme} \\ args : list(int + float) & \text{fournis par le programme} \\ res : (int + float) \} & \text{fourni par l'extérieur} \end{array}$

$E0$ = aucun événement.

$t1 ** t2$ = concaténation de traces.

Un événement est produit lors de l'exécution d'un appel de fonction externe.

Les autres règles d'évaluation propagent / concatènent les traces.

Quelques règles de sémantique

Séquence d'instructions

$$\frac{G, E \vdash s_1, M \Rightarrow \text{Normal}, t_1, M_1 \quad G, E \vdash s_2, M_1 \Rightarrow \text{out}, t_2, M_2}{G, E \vdash (s_1; s_2), M \Rightarrow \text{out}, t_1 ** t_2, M_2}$$

$$\frac{G, E \vdash s_1, M \Rightarrow \text{out}, t, M' \quad \text{out} \neq \text{Normal}}{G, E \vdash (s_1; s_2), M \Rightarrow \text{out}, t, M'}$$

Quelques règles de sémantique

Boucles `while`

$$\frac{G, E \vdash a, M \Rightarrow v, t, M' \quad \text{is_false}(v)}{G, E \vdash (\text{while}(a) s), M \Rightarrow \text{Normal}, t, M'}$$

$$\frac{G, E \vdash a, M \Rightarrow v, t_1, M_1 \quad \text{is_true}(v) \quad G, E \vdash s, M_1 \Rightarrow \text{Break}, t_2, M_2}{G, E \vdash (\text{while}(a) s), M \Rightarrow \text{Normal}, t_1 ** t_2, M_2}$$

$$\frac{G, E \vdash a, M \Rightarrow v, t_1, M_1 \quad \text{is_true}(v) \quad G, E \vdash s, M_1 \Rightarrow \text{out}, t_2, M_2 \quad \text{out} \in \{\text{Normal}, \text{Continue}\}}{G, E \vdash (\text{while}(a) s), M_2 \Rightarrow \text{out}', t_3, M_3}$$

$$\frac{G, E \vdash (\text{while}(a) s), M_2 \Rightarrow \text{out}', t_3, M_3}{G, E \vdash (\text{while}(a) s), M \Rightarrow \text{out}', t_1 ** t_2 ** t_3, M_3}$$

Le langage Csharpminor

- Un langage impératif, structuré en expressions, *statements*, et fonctions.
- De bas niveau : accès à la mémoire par `load` et `store` de quantités élémentaires (entier 8 bits signé, flottant 64 bits, etc), avec calculs d'adresses explicites.
- Opérateurs arithmétiques différents pour entiers/pointeurs et flottants. Conversions explicites.
- Contrôle structuré : `if/then/else`, boucles infinies, blocs et sortie de bloc.

Traduction Clight → Csharpminor

Traduction des opérations qui dépendent des types

- Résolution de la surcharge des opérateurs arithmétiques.

```
int x, y;    ... x + y ...    ---> addint(x, y)
double x, y; ... x + y ...    ---> addfloat(x, y)
```

- Calculs d'adresses.

```
int * p, i; ... p[i] ... ---> addint(p, mulint(i, 4))
struct intlist * s; ... s->t1 ... ---> addint(s, 4)
```

- Insertion de `load` et `store` lors des accès aux l-values.

```
int p[2][2]; p[0][0] = 42; ---> store(int32, p, 42)
int **p; p[0][0]=42; ---> store(int32, load(int32,p), 42)
```

Traduction Clight \rightarrow Csharpminor

Codage des structures de contrôle de Clight dans celles de Csharpminor
(identiques à celles de Cminor)

- `if ... then ... else`
- `boucle infinie`
- `block et exit n`
- `switch(e) { ... case n_i : exit k_i ... default: exit k }`

Traduction Clight \rightarrow Csharpminor

Codage de la boucle `for`

```
for ( $s_1$ ;  $e$ ;  $s_2$ ) {  
  
    ...  
    exit;  
    ...  
    continue;  
    ...  
  
}
```

```
 $s_1$ ;  
block { loop {  
    if (! $e$ ) exit 0;  
    block {  
        ...  
        exit 1;  
        ...  
        exit 0;  
        ...  
    }  
     $s_2$ ;  
} }
```

Traduction Clight \rightarrow Csharpminor

Codage du switch

```
switch(e) {
  case 1 :
    S1 ;
  case 2 : case 3 :
    S2 ; break ;
  case 4 :
    S3 ; break ;
  default :
    S4 ;
}

block {
  block {
    block {
      block {
        switch(e) {
          case 1 : exit 0 ;
          case 2 : exit 1 ;
          case 3 : exit 1 ;
          case 4 : exit 2 ;
          default : exit 3 ;
        }
      }
    }
  }
  S1
}
S2 ; exit 2 ;
}
S3 ; exit 1 ;
}
S4 }
```

Traduction Clight \rightarrow Csharpminor

En Coq ...

```
Fixpoint transl_expr
  (a : Csyntax.expr)
  {struct a} : option Cshminor.expr := ...
```

```
with transl_lvalue
  (a : Csyntax.expr)
  {struct a} : option Cshminor.expr := ...
```

```
Fixpoint transl_statement
  (nbrk ncnt : nat) (s : Csyntax.statement)
  {struct s} : option Cshminor.stmt := ...
```

```
Definition transl_function (f : Csyntax.function)
  : option Cshminor.function := ...
```

Traduction Clight \rightarrow Csharpminor

Zoom

```
Fixpoint transl_expr
  (a : Csyntax.expr)
  {struct a} : option Cshminor.expr :=
match a with
| Expr (Csyntax.Econst_int n) _ =>
  Some (make_intconst n)
| Expr (Csyntax.Evar id) ty =>
  var_get id ty
| Expr (Csyntax.Ebinop op b c) _ =>
  do tb <- transl_expr b;
  do tc <- transl_expr c;
  transl_binop op tb (typeof b) tc (typeof c)
| ...
```

Préservation sémantique avec traces

Les traces permettent de montrer des théorèmes de préservation sémantique beaucoup plus intéressants :

```
Theorem transf_program_correct :  
  forall prog tprog trace exitcode,  
    transf_program prog = Some tprog ->  
    SourceLanguage.exec_program prog trace exitcode ->  
    TargetLanguage.exec_program prog trace exitcode.
```

Toutes les passes du compilateur Compcert ont été équipées de traces et les théorèmes de préservation sémantiques renforcés en conséquence.

Correction de la traduction Clight \rightarrow Csharpminor

Le théorème de préservation de la sémantique

```
Theorem transl_program_correct :  
  forall prog tprog t r,  
    transl_program prog = Some tprog ->  
    Ctyping.wt_program prog ->  
    Csem.exec_program prog t r ->  
    Csharpminor.exec_program tprog t r.
```

L'hypothèse de typage `wt_program` demande uniquement que toutes les occurrences d'une même variable (locale ou globale) soient décorées avec le même type.

Le compilateur appelle une procédure de décision `typecheck_program` qui vérifie cette hypothèse.

Traduction Csharpminor → Cminor

Placement des variables locales

Les variables locales Csharpminor résident en mémoire, on peut toujours en prendre l'adresse. Ce n'est pas le cas en Cminor.

⇒ allocation explicite en pile des variables tableaux et scalaires dont on prend l'adresse.

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

Traduction Csharpminor \rightarrow Cminor

Difficulté

L'allocation des blocs de mémoire n'est pas la même en Csharpminor et en Cminor.

- \Rightarrow un bloc par variable locale, pour chaque appel de fonction
- \Rightarrow un bloc d'activation par appel de fonction
- \Rightarrow Définition d'**injections entre mémoires**.

Injections entre mémoires

Fonctions $\alpha : \text{Csharpminor block} \rightarrow \text{option}(\text{Cminor block} \times \text{offset})$.

$\alpha(b) = \text{None}$:

le bloc Csharpminor b disparaît pendant la traduction.

$\alpha(b) = \text{Some}(b', \delta)$:

le bloc Csharpminor b devient un sous-bloc du bloc Cminor b' à l'offset δ .

\Rightarrow Correspondance $\alpha \vdash v \approx v'$ entre valeurs.

$$\alpha \vdash i \approx i \qquad \alpha \vdash f \approx f \qquad \frac{\alpha(b) = \text{Some}(b', \delta)}{\alpha \vdash (b, o) \approx (b', o + \delta)}$$

\Rightarrow Correspondance $\alpha \vdash M \approx M'$ entre états mémoire.

(les contenus des blocs coïncident + pas de chevauchement entre sous-blocs.)

Préservation de la sémantique : preuve

Par induction sur les dérivations des exécutions des programmes Csharpminor, en utilisant des diagrammes de simulation de la forme suivante :

$$\begin{array}{ccc} G, E \vdash a, M_1 & \xrightarrow{\text{Inv}} & G', sp \vdash a', E'_1, M'_1 \\ \downarrow \text{évaluation C\#minor} & & \text{évaluation Cminor} \downarrow \\ G, E \vdash v, M_2 & \xrightarrow{\text{Inv} \wedge \alpha \vdash v \approx v'} & G', sp \vdash v', E'_2, M'_2 \end{array}$$

avec a' traduction de a .

L'invariant Inv représente la correspondance entre les états mémoire ($\alpha \vdash M \approx M'$).

Monomorphisation des fonctions variadiques externes

Le générateur de Clight déclare une fonction externe par nombre et types d'arguments :

```
extern int printf(char * fmt, ...);

printf("%s\n", s);
printf("%d %g\n", d, f);
--->

extern int printf$ii(char *, int);
extern int printf$iiif(char *, int, double);

printf$ii((int) "%s\n", (int) s);
printf$iiif((int) "%s\n", d, f);
```

Le générateur d'assembleur PowerPC reconnaît les fonctions `xxx$iiif` et produit du “stub code” qui réorganise les arguments et appelle la vraie fonction `xxx`.

Plan

- 1 Vérification formelle de compilateurs
- 2 Compilateur CompCert
- 3 Un front-end pour le compilateur CompCert
 - CIL
 - Sémantique du langage Clight
 - 1^{ere} passe : de Clight à Csharpminor
 - Correction de la traduction
 - 2^e passe : de Csharpminor à Cminor
- 4 Conclusion

Retour d'expérience

La structure générale du compilateur est conditionnée non pas par les transformations de programmes, mais par le **choix des langages** du compilateur, et aussi par le **style de sémantique** donné à ces langages.
→ Les langages intermédiaires du compilateur ont été conçus dans le but de faciliter les preuves de traduction (**conception assistée par la preuve**).

C'est lorsqu'une traduction de C vers Cminor a été prouvée, qu'il a été décidé de concevoir Csharpminor.

Validation des sémantiques.

Travaux actuels

- Modéliser les programmes qui ne terminent pas :
 - ⇒ sémantiques à petits pas,
 - ⇒ sémantiques coinductives.
- Liens avec les logiques de programmes : sémantiques axiomatiques et logique de séparation.
- Cminor :
 - ⇒ langage intermédiaire pour d'autres compilateurs (mini-ML, Featherweight Java)
 - ⇒ extension à un langage concurrent (Concurrent Cminor).

Conclusion

Vérifier (en Coq) un compilateur réaliste semble faisable.

Compcert est un exemple d'une problématique plus générale :
la **vérification formelle des outils** intervenant dans la production et la certification des logiciels critiques.