Compilation optimisante et vérification formelle de compilateurs

Xavier Leroy

INRIA Rocquencourt

École Jeunes Chercheurs en Programmation, 2007-06-07

Plan du cours

La compilation optimisante

Objectifs: produire du code machine rapide et compact. Bien exploiter les possibilités des processeurs hardware.

Les approches à base de machines abstraites ne suffisent pas:

- Interprétation du code abstrait:
  surcoût de l'interprétation (facteur 5-10).
- Expansion en code machine:
  beaucoup d'inefficacités et de redondances.

```
CONST(i)        --->    pushl $i
ADD                     popl %eax
                        addl 0(%esp), %eax
```

(Au lieu de `addl 0(%esp), $i`)

Comment produire du code machine efficace?

- Ne pas introduire d'inefficacités au cours de la compilation.
- Appliquer des optimisations pour éliminer des inefficacités.

Les inefficacités visées sont de 3 formes:

- présentes dans le code source

```
let x = 1 + 2 in ...   --->  let x = 3 in ...
```

- implicites dans le code source

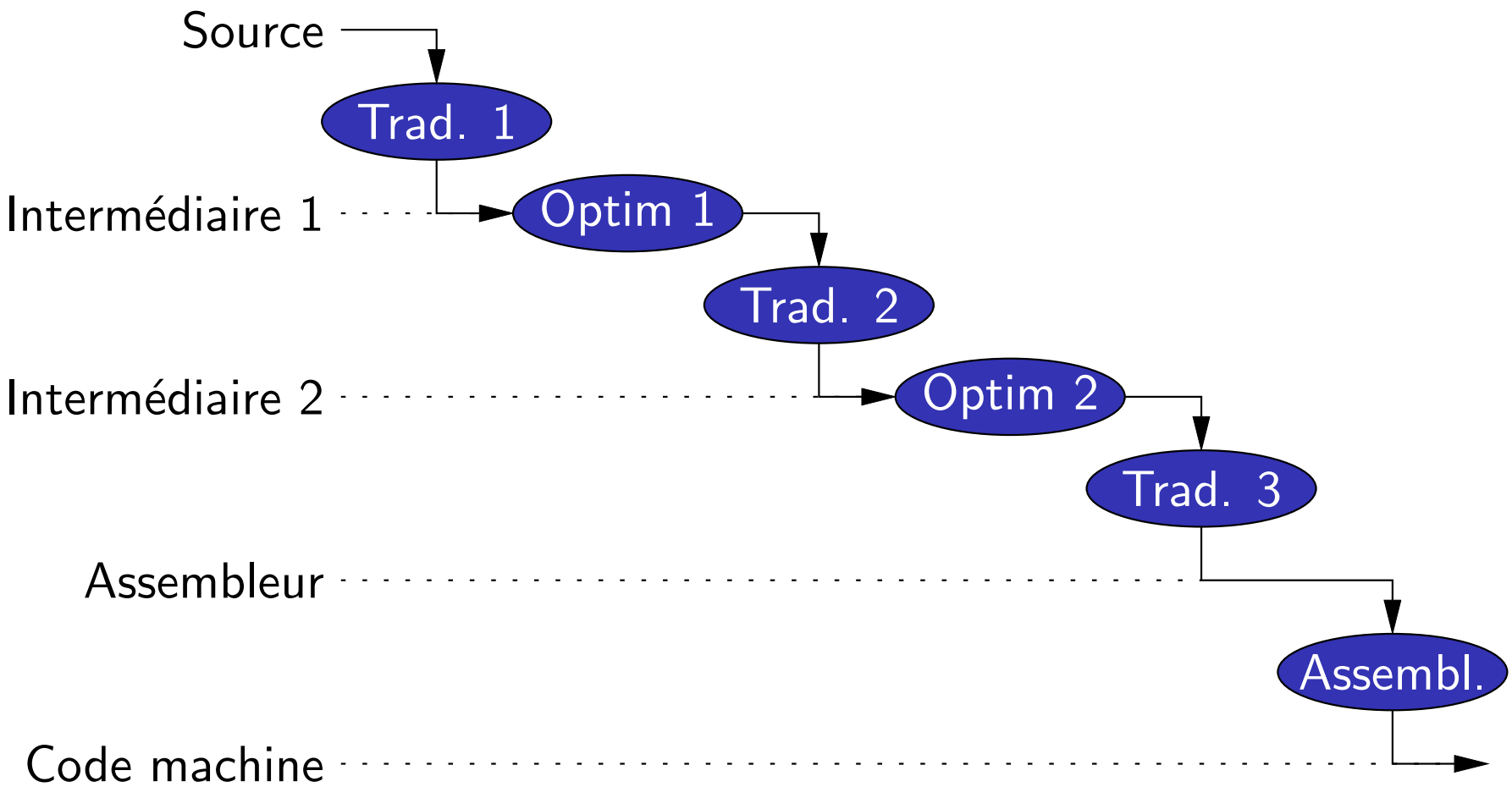
```
a.[i] + b.[i]          --->  load(a + i*4) + load(b + i*4)
```

- introduites par des passes précédentes

```
let x = 1 in          --->  let x = 1 in    --->   3
let y = x + 2 in            let y = 3 in
y                           3
```

Comment produire du code machine efficace?

Procéder en plusieurs passes de transformation successives.
Utiliser des langages intermédiaires pour tailler des marches entre le langage source et le code machine.



Le choix des langages intermédiaires

Un langage intermédiaire explicite certains aspects du programme et se prête aux optimisations portant sur ces aspects.

Exemple 1: l'expansion en ligne de fonctions se fait idéalement sur un langage de type λ-calcul. Serait beaucoup plus difficile sur des langages de plus bas niveau.

```
(fun x → a) b     --->    a{x ← b}
```

Exemple 2: la factorisation de sous-expressions communes est plus efficace sur un langage intermédiaire où les calculs d'adresses sont explicites que sur le langage source.

```
a[i] + b[i]          --->  ??

load(a+i*4) + load(b+i*4)  --->  let t = i * 4 in
                                 load(a + t) + load(b + t)
```

Le choix des langages intermédiaires

Un langage intermédiaire explicite certains aspects du programme et se prête aux optimisations portant sur ces aspects.

Exemple 1: l'expansion en ligne de fonctions se fait idéalement sur un langage de type λ-calcul. Serait beaucoup plus difficile sur des langages de plus bas niveau.

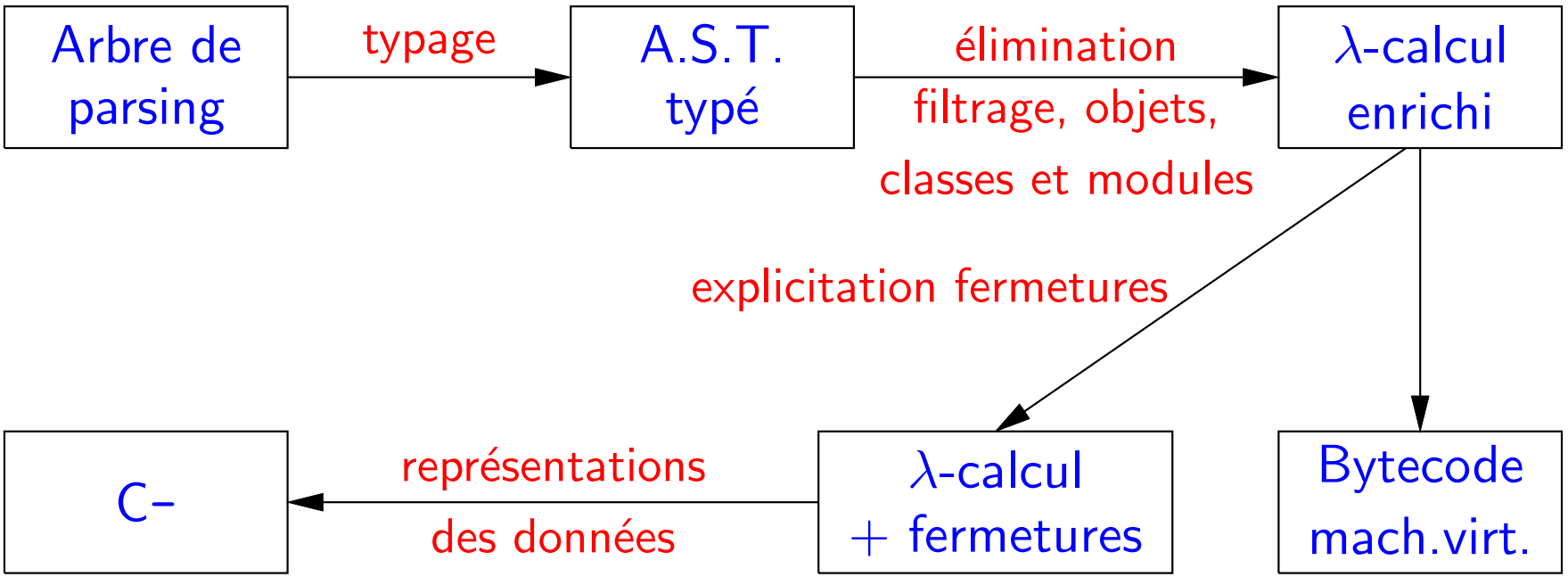
```
(fun x → a) b     --->    a{x ← b}
```

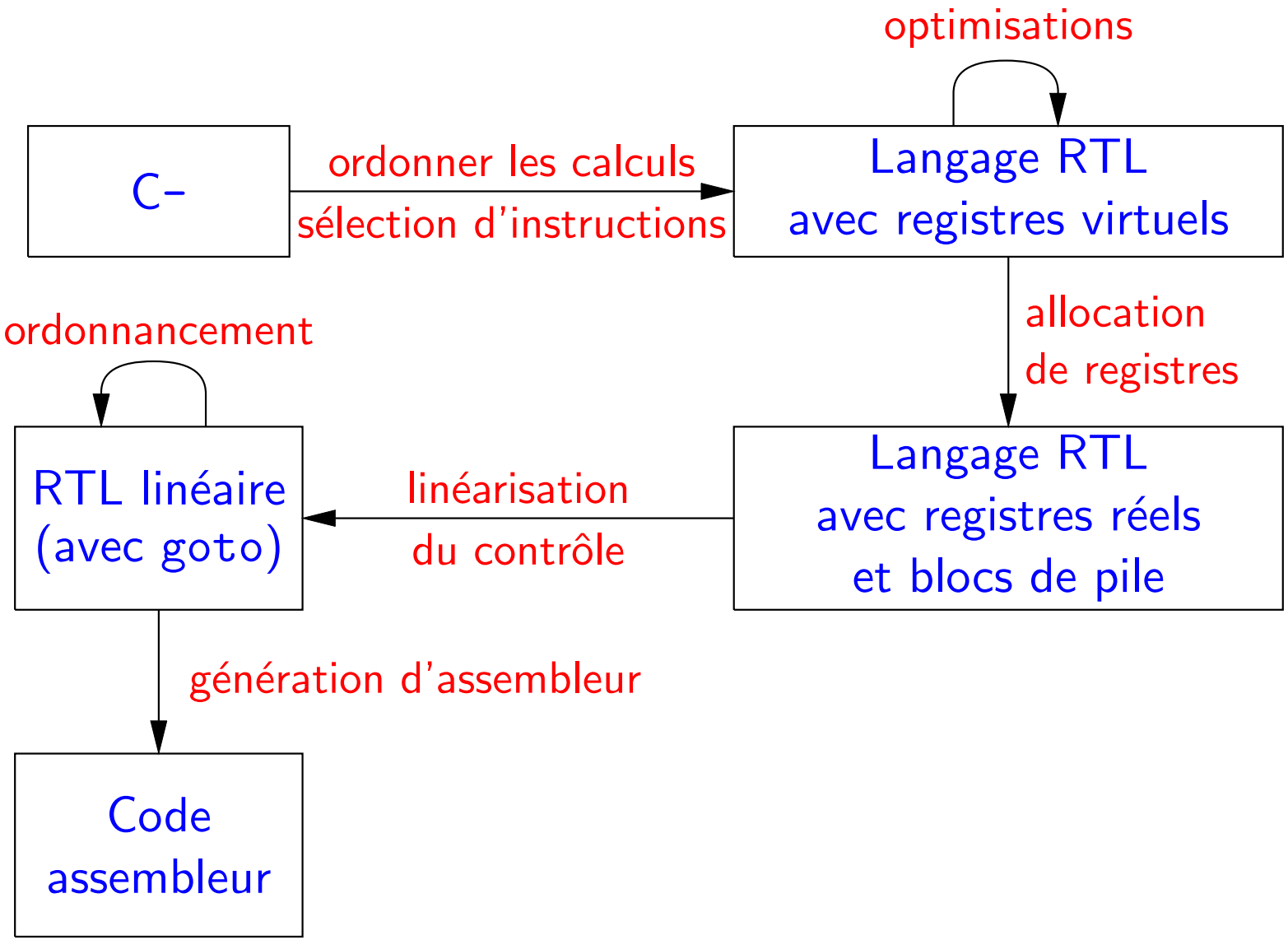
Exemple 2: la factorisation de sous-expressions communes est plus efficace sur un langage intermédiaire où les calculs d'adresses sont explicites que sur le langage source.

```
a[i] + b[i]          --->  ??

load(a+i*4) + load(b+i*4)  --->  let t = i * 4 in
                                 load(a + t) + load(b + t)
```

Exemple: le compilateur OCaml




optimisations
C-
ordonner les calculs
sélection d'instructions
Langage RTL
avec registres virtuels
allocation
de registres
ordonnancement
RTL linéaire
(avec goto)
linéarisation
du contrôle
Langage RTL
avec registres réels
et blocs de pile
génération d'assembleur
Code
assembleur

Le langage intermédiaire RTL-CFG

Aussi appelé "code 3 adresses". Intermédiaire entre un langage de haut niveau (identificateurs, expressions, structures de contrôle) et le langage machine (registres, instructions, `goto`):

- 1 opération $\approx$ 1 instruction du processeur.
- Opérandes: des temporaires (pseudo-registres) en nombre arbitraire, préservés lors de l'appel de fonctions.
- Contrôle: non structuré, exprimé par un graphe de flot.

Exemple de RTL

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

```
average(tbl, size)
  var fs, fc, fd, f1, ri, ra, rb;

I1:  fs = 0.0;              --> I2
I2:  ri = 0;                --> I3
I3:  if (ri >= size)        --> I9/I4
I4:  ra = ri * 4            --> I5
I5:  rb = load(tbl + ra)    --> I6
I6:  fc = float_of_int(rb) -> I7
I7:  fs = fs +f fc          --> I8
I8:  ri = ri + 1            --> I3
I9:  fd = float_of_int(size) --> I10
I10: f1 = fs /f fd          --> I11
I11: return f1
```

Syntaxe de RTL

Registres: $r ::= r_1, r_2, r_3, \ldots$

Instructions:
$$\begin{aligned} instr ::= {} & \texttt{nop} \\ & \mid r = op(r_1, \ldots, r_n) \\ & \mid r = \texttt{load}(mode, r_1, \ldots, r_n) \\ & \mid \texttt{store}(r, mode, r_1, \ldots, r_n) \\ & \mid \texttt{if}\ (cond, r_1, \ldots, r_n) \\ & \mid r = \texttt{call}(addr, r_1, \ldots, r_n) \\ & \mid r = \texttt{callind}(r, r_1, \ldots, r_n) \\ & \mid \texttt{return}\ r \end{aligned}$$

Fonctions:

$fn ::= \{\ params = \vec{r};$	paramètres
$\quad vars = \vec{r};$	variables locales
$\quad code = \ell \rightarrow instr * \vec{\ell};$	graphe de flot
$\quad start = \ell\ \}$	point d'entrée

Opérateurs, modes d'adressage, conditions

Ceux du processeur cible. Exemple du PowerPC:

- Opérateurs: constantes, `move`, opérateurs logiques, arithmétique entière, arithmétique flottante, ...
  Opérations composites: multiplication-et-addition, rotation-et-masquage, ...
- Modes d'adressage: $r + cst$ ou $r_1 + r_2$.
- Conditions: comparaisons entières et flottantes, tests de bits.

Sémantique opérationnelle de RTL

Relation de transition (exécution d'une instruction):

$$P, f \vdash (\ell, R, M) \rightarrow (\ell', R', M')$$

Programme P: adresse $\rightarrow$ fonction.
Fonction courante f.
Point de contrôle ℓ (avant), ℓ' (après).
État des registres (registre $\rightarrow$ valeur) R, R'.
État mémoire (adresse $\rightarrow$ valeur) M, M'.

$$\frac{f.code(\ell) = (\mathtt{nop}, [\ell'])}{P, f \vdash (\ell, R, M) \rightarrow (\ell', R, M)}$$

$$\frac{f.code(\ell) = (r = op(r_1, \ldots, r_n), [\ell']) \qquad v = \overline{op}(R(r_1), \ldots, R(r_n))}{P, f \vdash (\ell, R, M) \rightarrow (\ell', R\{r \leftarrow v\}, M)}$$

$$\frac{f.code(\ell) = (r = \texttt{load}(mode, r_1, \ldots, r_n), [\ell']) \qquad a = \overline{mode}(R(r_1), \ldots, R(r_n))}{P, f \vdash (\ell, R, M) \rightarrow (\ell', R\{r \leftarrow M(a)\}, M)}$$

$$\frac{f.code(\ell) = (\texttt{store}(r, mode, r_1, \ldots, r_n), [\ell']) \qquad a = \overline{mode}(R(r_1), \ldots, R(r_n))}{P, f \vdash (\ell, R, M) \rightarrow (\ell', R, M\{a \leftarrow R(r)\})}$$

$$\frac{f.code(\ell) = (\texttt{if}(cond, r_1, \ldots, r_n), [\ell_1; \ell_2]) \qquad \overline{cond}(R(r_1), \ldots, R(r_n)) \neq 0}{P, f \vdash (\ell, R, M) \rightarrow (\ell_1, R, M)}$$

$$\frac{f.code(\ell) = (\texttt{if}(cond, r_1, \ldots, r_n), [\ell_1; \ell_2]) \qquad \overline{cond}(R(r_1), \ldots, R(r_n)) = 0}{P, f \vdash (\ell, R, M) \rightarrow (\ell_2, R, M)}$$

Sémantique opérationnelle de RTL: appels de fonctions

Un appel de fonction (`call`) se présente comme une seule transition faisant appel au prédicat auxiliaire

$$P, f \vdash (\ell, R, M) \stackrel{*}{\longrightarrow} (v_{res}, M')$$

qui représente l'exécution du corps de la fonction jusqu'à la première instruction `return`.

$$\frac{\begin{array}{c} f.code(\ell) = (r = \texttt{call}(addr, r_1, \ldots, r_n), [\ell']) \\ P(addr) = f' \qquad R_0 = (f'.params \mapsto R(r_1), \ldots, R(r_n)) \\ P, f' \vdash (f'.start, R_0, M) \stackrel{*}{\longrightarrow} (v, M') \end{array}}{P, f \vdash (\ell, R, M) \longrightarrow (\ell', R\{r \leftarrow v\}, M')}$$

Sémantique opérationnelle de RTL: appels de fonctions

$$\frac{f.code(\ell) = (\texttt{return}\ r, [\,]) \qquad v = R(r)}{P, f \vdash (\ell, R, M) \xrightarrow{*} (v, M)}$$

$$\frac{P, f \vdash (\ell, R, M) \rightarrow (\ell_1, R_1, M_1) \qquad P, f \vdash (\ell_1, R_1, M_1) \xrightarrow{*} (v, M')}{P, f \vdash (\ell, R, M) \xrightarrow{*} (v, M')}$$

Optimisations classiques sur RTL

De nombreuses optimisations classiques s'effectuent sur le RTL.

- Constant propagation

```
ra = 1                          ra = 1
rb = 2            -->           rb = 2
rc = ra + rb                    rc = 3
rd = rx - ra                    rd = rx + (-1)
```

- Dead code elimination

```
ra = 1                          nop
rb = 2            -->           rb = 2
rc = 3                          rc = 3
```

(si `ra` inutilisé par la suite)

- Common subexpression elimination

```
rc = ra                          rc = ra
rd = ra + rb          -->        rd = ra + rb
re = rc + rb                     re = rd
```

- Hoisting of loop-invariant computations

```
   I: rc = ra + rb                  rc = ra + rb
      ...                -->    I: ...
      ... -> I                      ... -> I
```

- Induction variable elimination

```
      ri = 0                        ri = 0
   I: ra = ri * 4        -->        rb = rp
      rb = rp + ra               I: ...
      ...                           rb = rb + 4
      ri = ri + 1 -> I              ri = ri + 1 -> I
```

- ... et bien d'autres encore.

Optimisation et analyse statique

Exemple de la propagation des constantes:

```
ra = 1                          ra = 1
rb = 2          -->             rb = 2
rc = ra + rb                    rc = 3
rd = rx - ra                    rd = rx + (-1)
```

Effectuer à la compilation les opérations arithmétiques dont les opérandes sont statiquement connues; spécialiser les opérations dont un des opérandes est statiquement connu.

Problème: comment déterminer statiquement que `ra` et `rb` sont constants, et prédire leur valeur?

Solution: analyse statique de type *dataflow* (voir plus loin).

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 = 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;
```

```
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;
}
```

Plan du cours

Peut-on faire confiance au compilateur?

Les transformations effectuées par un compilateur optimisant réaliste sont complexes; une erreur est vite arrivée...

Contexte: le logiciel critique (embarqué) devant passer un certain niveau de certification.

Certification par des tests systématiques:

- Ce qui est testé est le code exécutable produit par le compilateur.
- Les bugs du compilateur sont détectés en même temps que ceux du programme.

Certification par méthodes formelles (preuve, model-checking):

- Ce qui est prouvé est le code source, pas le code exécutable.
- Les bugs du compilateur peuvent invalider l'approche.
- Pratique actuelle: revues manuelles du code assembleur produit par le compilateur (sans optimisations).

Peut-on faire confiance au compilateur?

Les transformations effectuées par un compilateur optimisant réaliste sont complexes; une erreur est vite arrivée...

Contexte: le logiciel critique (embarqué) devant passer un certain niveau de certification.

Certification par des tests systématiques:

- Ce qui est testé est le code exécutable produit par le compilateur.
- Les bugs du compilateur sont détectés en même temps que ceux du programme.

Certification par méthodes formelles (preuve, model-checking):

- Ce qui est prouvé est le code source, pas le code exécutable.
- Les bugs du compilateur peuvent invalider l'approche.
- Pratique actuelle: revues manuelles du code assembleur produit par le compilateur (sans optimisations).

Vérifier le compilateur

Appliquer les méthodes formelles au compilateur lui-même.

Établir formellement une propriété de préservation $Prop(S, C)$ entre le code compilé C et le 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 spécification *Spec*, alors C satisfait *Spec*
4. si S est sûr vis-à-vis du typage et de la mémoire, C l'est aussi".

Compilateurs prouvés

Compilateur = une fonction totale $\mathit{Comp} : \mathit{Source} \to \texttt{option}\ \mathit{Code}$.

Comp est prouvé si

$$\forall S, C, \quad \mathit{Comp}(S) = \texttt{Some}(C) \Rightarrow \mathit{Prop}(S, C)$$

Code auto-certifié

Proof-carrying code (P. Lee, G. Necula, A. Appel)

Compilateur certifiant

$$\mathit{CComp} : \mathit{Source} \to \texttt{option}\ (\mathit{Code} \times \mathit{Proof})$$

Si $\mathit{CComp}(S) = \texttt{Some}(C, \pi)$ et $\pi \vdash \mathit{Prop}(S, C)$, la compilation est correcte.

Validation de traducteurs

Translation validation (A. Pnueli et al; G. Necula; X. Rival)

Compilateur + vérificateur indépendant:

$$\begin{aligned} \mathit{Comp} &: \mathit{Source} \to \texttt{option}\ \mathit{Code} \\ \mathit{Verif} &: \mathit{Source} \times \mathit{Code} \to \texttt{bool} \end{aligned}$$

Le vérificateur est supposé prouvé:

$$\forall S, C, \quad \mathit{Verif}(S, C) = \texttt{true} \Rightarrow \mathit{Prop}(S, C)$$

Si $\mathit{Comp}(S) = \texttt{Some}(C)$ et $\mathit{Verif}(S, C) = \texttt{true}$, la compilation est correcte.

Unifier code auto-certifié et validation de traducteurs

Compilateur certifiant + vérificateur indépendant:

$$\begin{aligned} \mathit{Comp} &: \mathit{Source} \to \texttt{option}\ (\mathit{Code} \times \mathit{Annot}) \\ \mathit{Verif} &: \mathit{Source} \times \mathit{Code} \times \mathit{Annot} \to \texttt{bool} \end{aligned}$$

Le vérificateur est supposé prouvé

$$\forall S, C, A, \quad \mathit{Verif}(S, C, A) = \texttt{true} \Rightarrow \mathit{Prop}(S, C)$$

Techno	Annotations	Vérifieur
Proof-carrying code	Terme de preuve	Proof-checking
Translation validation pure	Rien	Analyse statique
Translation validation pratique	Infos de debug, etc	Analyse statique
Type-preserving compilation, typed assembly language, bytecode verification	Annotations de types	Type-checking

De certifiant à prouvé

Si *CComp* et *Verif* forment un compilateur certifiant, la fonction ci-dessous est un compilateur prouvé:

$$
\begin{array}{l}
\mathit{Comp}(S) = \\
\quad \texttt{match } \mathit{CComp}(S) \texttt{ with} \\
\quad \mid \texttt{None} \rightarrow \texttt{None} \\
\quad \mid \texttt{Some}(C, A) \rightarrow \texttt{if } \mathit{Verif}(S, C, A) \texttt{ then Some}(C) \texttt{ else None}
\end{array}
$$

De prouvé à certifiant

Si *Comp* est un compilateur prouvé et Π le terme de preuve Coq pour $\forall S, C,\ \mathit{Comp}(S) = \mathtt{Some}(C) \Rightarrow \mathit{Prop}(S, C)$, la fonction ci-dessous est un compilateur certifiant:

$$
\begin{array}{l}
\mathit{CComp}(S) = \\
\quad \mathtt{match}\ \mathit{Comp}(S)\ \mathtt{with} \\
\quad \mid \mathtt{None} \rightarrow \mathtt{None} \\
\quad \mid \mathtt{Some}(C) \rightarrow \mathtt{Some}(C, \Pi\ S\ C\ \pi_{eq})
\end{array}
$$

avec $\pi_{eq} : \mathit{Comp}(S) = \mathtt{Some}(C)$.

Verif est le vérificateur de preuves Coq.

(En logique constructive, Π est une fonction qui prend S, C et une preuve de $\mathit{Comp}(S)$ et renvoie une preuve de $\mathit{Prop}(S, C)$.)

Découpage en passes

Si $Comp_1 : L_1 \to \texttt{option}\ L_2$ et $Comp_2 : L_2 \to \texttt{option}\ L_3$ sont des compilateurs prouvés, leur composition monadique

$$\begin{array}{l} Comp(S) = \\ \quad \texttt{match}\ Comp_1(S)\ \texttt{with} \\ \quad \mid \texttt{None} \to \texttt{None} \\ \quad \mid \texttt{Some}(I) \to Comp_2(I) \end{array}$$

est un compilateur prouvé de L_1 vers L_3 à condition que $Prop$ soit transitive:

$$Prop(S, I) \wedge Prop(I, C) \Rightarrow Prop(S, C)$$

Découpage en passes

Si $Comp_1$ et $Comp_2$ sont deux (passes de) compilateurs certifiants, et $Verif_1$, $Verif_2$ les vérifieurs correspondants,

$$\begin{array}{rcl} Comp(S) & = & \texttt{let } (I, A_1) = Comp_1(S) \texttt{ in} \\ & & \texttt{let } (C, A_2) = Comp_2(I) \texttt{ in} \\ & & (C, (A_1, I, A_2)) \end{array}$$

$$Verif(C, S, (A_1, I, A_2)) \quad = \quad Verif_1(I, S, A_1) \texttt{ and } Verif_2(C, I, A_2)$$

forment un compilateur certifiant.

Méthodologie

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

Application: le projet Compcert

(INRIA Rocquencourt, INRIA Sophia, CNAM, PPS.)

Une expérience de développement d'un compilateur prouvé réaliste, utilisable pour le logiciel embarqué critique.

- Langage source = un sous-ensemble de C.
- Language cible = l'assembleur du processeur PowerPC.
- Produit du code raisonnablement compact et rapide
  $\Rightarrow$ quelques optimisations parmi les plus rentables.

(*Formal certification of a compiler back-end, or: programming a compiler with a proof assistant*, X. Leroy, *Principles Of Progr. Lang.*, 2006.)

(*Formal verification of a C compiler front-end*, S. Blazy, Z. Dargaye, X. Leroy, *Int. Symp. Formal Methods*, 2006.)

Prouvé en Coq

La preuve de correction du *back-end* 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 2 hommes-années et 40000 lignes de Coq.

Programmé en Coq

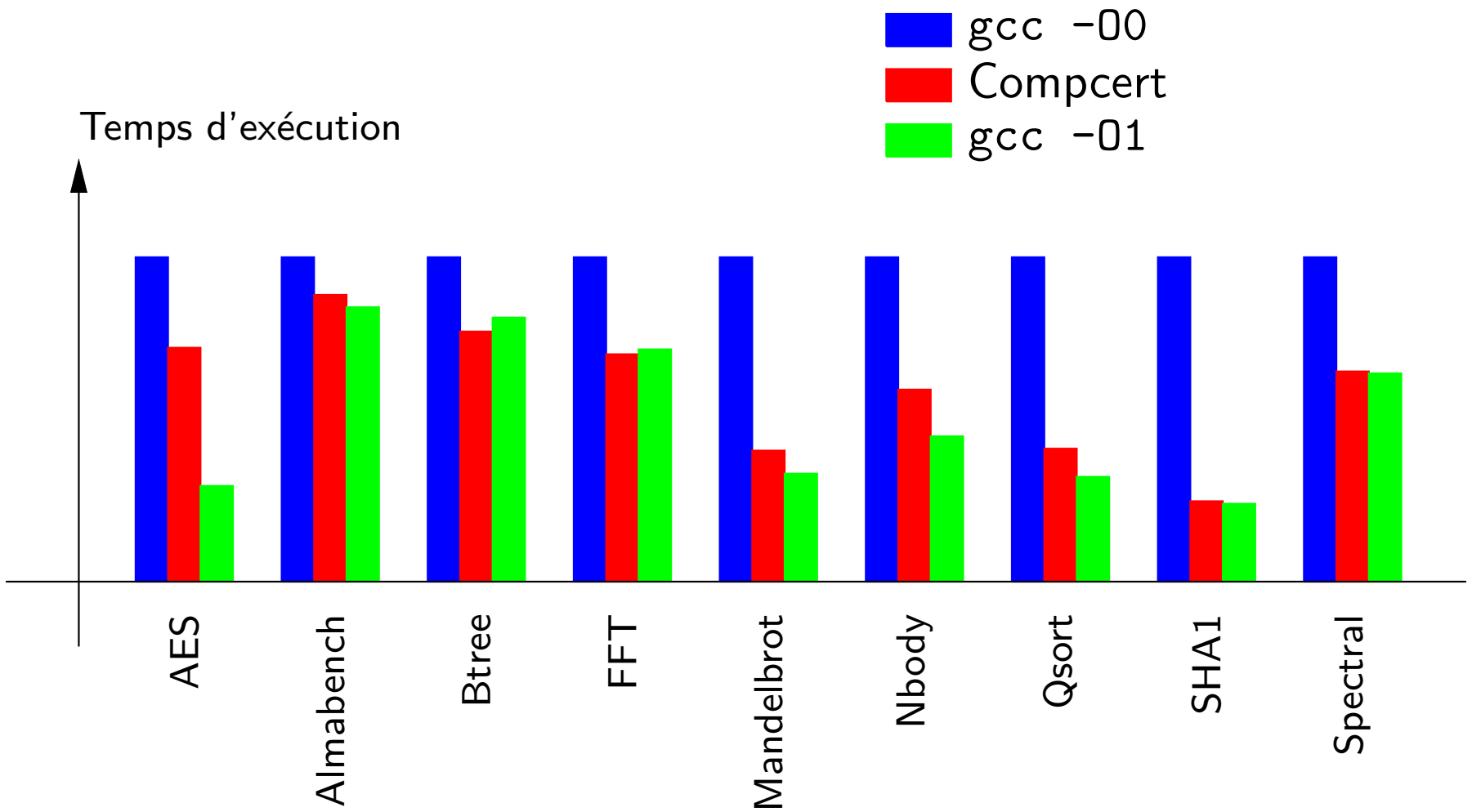
Toutes les parties prouvées du compilateur sont écrites directement dans le langage de spécification de Coq, sous forme de fonctions pures.

- Style monadique pour traiter les erreurs et les états globaux.
- Structures de données purement fonctionnelles (persistantes).

4500 lignes de Coq + 1500 lignes de Caml non certifié, notamment pour le coloriage de graphe dans l'allocation de registres.

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

Performances du code produit



Plan du cours

Les inéquations dataflow avant

À chaque point ℓ on associe une approximation $X(\ell)$ de l'état des registres en ce point.

Les approximations sont prises dans un demi-treillis L.
($a_1 \geq a_2$ signifie que l'approximation a_1 est plus grossière que a_2.)

Dataflow avant: trouver une solution X aux inéquations

$$\begin{array}{rcll} X(s) & \geq & T(\ell, X(\ell)) & \text{si } s \text{ est un successeur de } \ell \\ X(\ell) & \geq & a_\ell & \text{si } (\ell, a_\ell) \text{ est un point d'entrée} \end{array}$$

Ici, T est une fonction de transfer $T : \mathit{point} \times L \to L$ qui relie l'approximation avant (a) et après ($T(\ell, a)$) l'exécution de l'instruction au point ℓ.

Les points d'entrée sont une liste de couples (ℓ, a_ℓ) qui permet d'imposer des contraintes en des points particulier de la fonction.

Exemple: la propagation des constantes

Le demi-treillis des valeurs abstraites: les fonctions

$$reg \to (\top \mid value \mid \bot)$$

À chaque registre, on associe l'info valeur inconnue / valeur statiquement connue / pas de valeur (code inatteignable).

Idée de l'analyse: $\ell : r = op(r_1, \ldots, r_n)$.

Si $r_1, \ldots, r_n$ ont des valeurs connues au point ℓ, alors la valeur de r est connue aux successeurs de ℓ.

Sinon, la valeur de r est inconnue ($\top$) aux successeurs de ℓ.

$\to$ Une analyse dataflow avant.

La fonction de transfer et les points d'entrée

$a' = T(\ell, a)$ est défini par cas sur l'instruction au point ℓ:

- Cas $r = op(r_1, \ldots, r_n)$:
  - si $a(r_1) = v_1, \ldots, a(r_n) = v_n$ (valeurs connues),
    $a' = a\{r \leftarrow v\}$ avec $v = \overline{op}(v_1, \ldots, v_n)$.
  - sinon, $a' = a\{r \leftarrow \top\}$ (résultat non prévisible).
- Cas $r =$ `load(...)` ou $r =$ `call(...)` ou $r =$ `callind(...)`:
  $a' = a\{r \leftarrow \top\}$ (résultat non prévisible).
- Cas `store`, `if`, `return`:
  $a' = a$ (pas de modification des registres).

Contraintes de points d'entrée: $X(f.start) \geq \top$
(Les valeurs des registres à l'entrée de la fonction ne sont pas prévisibles.)

Résolution des inéquations dataflow

L'algorithme de la *worklist* de Kildall:

Initialisation: pour tout point ℓ,

$$\begin{aligned} \mathit{in}(\ell) &= a \text{ si } (\ell, a) \text{ est un point d'entrée, } \bot \text{ sinon} \\ \mathit{out}(\ell) &= \bot \\ W &= \text{l'ensemble de tous les points de la fonction} \end{aligned}$$

Résolution des inéquations dataflow

Itération de point fixe:

Tant que W n'est pas vide:
  Choisir ℓ dans W
  $W = W \setminus \{\ell\}$
  $out(\ell) = T(\ell, in(\ell))$
  Pour chaque successeur s de ℓ:
    $t = lub(in(s), out(\ell))$
    Si $t \neq in(s)$:
      $in(s) = t$
      $W = W \cup \{s\}$
    Fin Si
  Fin Pour tout
Fin Tant que.

À la sortie, *in* est une solution des inéquations dataflow.

Correction de l'algorithme de Kildall

Terminaison: à chaque étape, ou bien l'un des $in(\ell)$ augmente strictement, ou bien la taille de W diminue.
$\rightarrow$ Termine si le treillis L est bien fondé (pas de suites infinies strictement croissantes).

Deux invariants:

$$\begin{array}{lcll} in(s) & \geq & T(\ell, in(\ell)) & \text{si } \ell \notin W \text{ et } s \text{ est un successeur de } \ell \\ in(\ell) & \geq & in_0(\ell) & \text{pour tout } \ell \end{array}$$

Digression: optimalité vs. correction

L'algorithme de Kildall calcule en fait une solution minimale aux inéquations dataflow avant. Cette solution vérifie:

$$X(\ell) = \mathit{lub}\ \{\, T(p, X(p)) \mid p \text{ prédecesseur de } \ell \}$$

C'est cette forme qui est utilisée dans la littérature sous le nom "équations dataflow".

Cependant, l'égalité $X(\ell) = \ldots$ n'est pas nécessaire pour prouver la correction de l'analyse: l'inégalité $X(\ell) \geq \ldots$ suffit.

Les inéquations dataflow arrière

Pour certaines analyses, la fonction de transfer T calcule l'approximation avant ($T(\ell, a)$) à partir de l'approximation après (a) l'exécution de l'instruction au point ℓ. D'où:

Dataflow arrière: trouver une solution X aux inéquations

$$\begin{array}{lcll} X(\ell) & \geq & T(s, X(s)) & \text{si } s \text{ est un successeur de } \ell \\ X(\ell) & \geq & a_\ell & \text{si } (\ell, a_\ell) \text{ est un point d'entrée} \end{array}$$

Un problème dataflow arrière est équivalent à un problème dataflow avant sur le dual du graphe de flot de contrôle (renversement des arcs).

Propagation des constantes: correction de l'analyse

Un jeu de registres correspond à un résultat d'analyse si tous les registres dont la valeur est statiquement connue ont bien cette valeur.

$R : a$ si pour tout registre r,

- ou bien $a(r) = \top$
- ou bien $a(r) = v$ et $R(r) = v$ pour une certaine valeur v.

Remarque: si $a' \geq a$ et $R : a$, alors $R : a'$.

Propagation des constantes: correction de l'analyse

Si les registres avant l'exécution d'une instruction correspondent aux résultats de l'analyse en ce point, alors les registres après l'exécution correspondent aux résultats de l'analyse au point successeur.

Notant A le résultat de l'analyse statique de la fonction f:

$$\textit{Si } P, f \vdash (\ell, R, S) \rightarrow (\ell', R', S') \textit{ et } R : A(\ell), \textit{ alors } R' : A(\ell').$$

Propagation des constantes: transformation du code

On réécrit les instructions en fonction des résultats de l'analyse:

- Instruction arithmétique dont tous les arguments sont statiquement connus: remplacée par $r = v$.
- Opération arithmétique dont un des arguments est connu: remplacée par sa spécialisation (p.ex. $r = r_1 + r_2$ devient $r = r_1 + v$).
- Branchement conditionnel dont tous les arguments sont statiquement connus: remplacé par `nop` vers le bon successeur.

On note $\texttt{transf}(f)$ et $\texttt{transf}(P)$ le résultat de cette transformation.

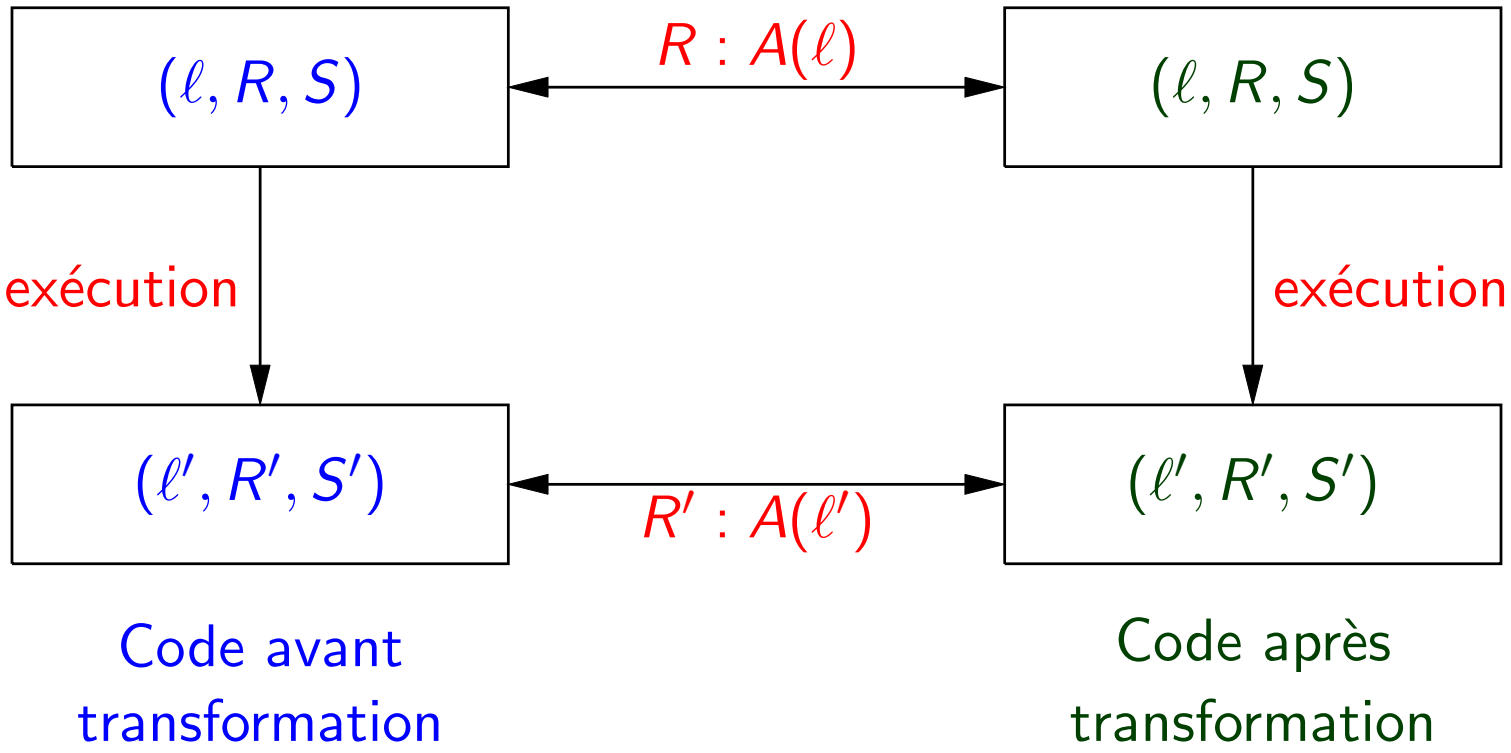
Propagation des constantes: correction de la transformation

Un résultat de simulation:

Si $P, f \vdash (\ell, R, S) \to (\ell', R', S')$ *et* $R : A(\ell)$, *alors* $\texttt{transf}(P), \texttt{transf}(f) \vdash (\ell, R, S) \to (\ell', R', S')$.

Si $P, f \vdash (\ell, R, S) \xrightarrow{*} (v, S')$ *et* $R : A(\ell)$, *alors* $\texttt{transf}(P), \texttt{transf}(f) \vdash (\ell, R, S) \xrightarrow{*} (v, S')$.

Le diagramme de simulation



Second exemple: analyse de durée de vie et applications

Un registre r est vivant en un point p si une instruction atteignable depuis p utilise r, et r n'est pas redéfini entre temps.

L'analyse de durée de vie (liveness analysis) détermine en chaque point l'ensemble des registres vivants en ce point.

Application 1: élimination de code mort.

$$\ell : r = op(r_1, \ldots, r_n) \quad \text{ou} \quad \ell : r = \texttt{load}(mode, r_1, \ldots, r_n)$$

Si r n'est pas vivant en ℓ, on peut supprimer cette instruction.

Application 2: allocation de registres (voir plus loin).

Analyse de durée de vie

Le demi-treillis des valeurs abstraites: les ensembles de registres (les registres vivants en ce point). Ordonnés par $\subseteq$, l.u.b = union.

Idée de l'analyse: $p : r = op(r_1, \ldots, r_n)$.

Si r est vivant en p, aux prédécesseurs de p il est mort, et $r_1, \ldots, r_n$ sont vivants.

$\rightarrow$ Une analyse dataflow arrière.

La fonction de transfer

$a' = T(\ell, a)$ est défini par cas sur l'instruction au point ℓ:

- Cas $r = op(r_1, \ldots, r_n)$ ou $r = \texttt{load}(mode, r_1, \ldots, r_n)$:
  Si $r \notin a$, $a' = a$ (instruction inutile).
  Si $r \in a$, $a' = (a \setminus \{r\}) \cup \{r_1, \ldots, r_n\}$.
- Cas $r = \texttt{call}(r_1, \ldots, r_n)$:
  $a' = (a \setminus \{r\}) \cup \{r_1, \ldots, r_n\}$.
- Cas $\texttt{store}(mode, r_1, \ldots, r_n)$ ou $\texttt{if}(cond, r_1, \ldots, r_n)$ ou $\texttt{return}(r_1)$:
  $a' = a \cup \{r_1, \ldots, r_n\}$.

Contraintes de points d'entrée: aucune.

Exemple de durées de vie

```
average(tbl: int, size: int): float
  var fs, fc, fd, f1: float
  var ri, ra, rb: int

  I1:  fs = 0.0;                    --> I2       [tbl,size,fs]
  I2:  ri = 0;                      --> I3       [tbl,size,ri,fs]
  I3:  if (ri >= size)              --> I9/I4    [tbl,size,ri,fs]
  I4:  ra = ri * 4                  --> I5       [tbl,size,ri,fs,ra]
  I5:  rb = load(tbl + ra)          --> I6       [tbl,size,ri,fs,rb]
  I6:  fc = float_of_int(rb)        --> I7       [tbl,size,ri,fs,fc]
  I7:  fs = fs +f fc                --> I8       [tbl,size,ri,fs]
  I8:  ri = ri + 1                  --> I3       [tbl,size,ri,fs]
  I9:  fd = float_of_int(size)      --> I10      [fs,fd]
  I10: f1 = fs /f fd                --> I11      [f1]
  I11: return f1                                 []
```

Application à l'allocation de registres

On dit que deux registres r_1 et r_2 interfèrent s'ils sont simultanément vivants en un point du programme.

(Exception: si r_1 et r_2 ont toujours la même valeur, p.ex. si la seule définition de r_2 est $r_2 = r_1$, ils n'interfèrent pas.)

Si r_1 et r_2 n'interfèrent pas, on peut les remplacer par un unique registre r.

$\rightarrow$ Permet de minimiser le nombre de registres nécessaires par coloriage du graphe représentant la relation d'interférence.

$\rightarrow$ Si ce nombre est $\leq$ au nombre de registres physiques, on a réussi une allocation de registres.

$\rightarrow$ Sinon, c'est une bon point de départ pour déterminer quels pseudo-registres vont dans un registre physiques et quels pseudo-registres vont dans des emplacements de pile.

Construction du graphe d'interférences

Graphe non orienté. Noeuds: pseudo-registres ou registres physiques. Arc entre 2 registres si et seulement si ils interfèrent.

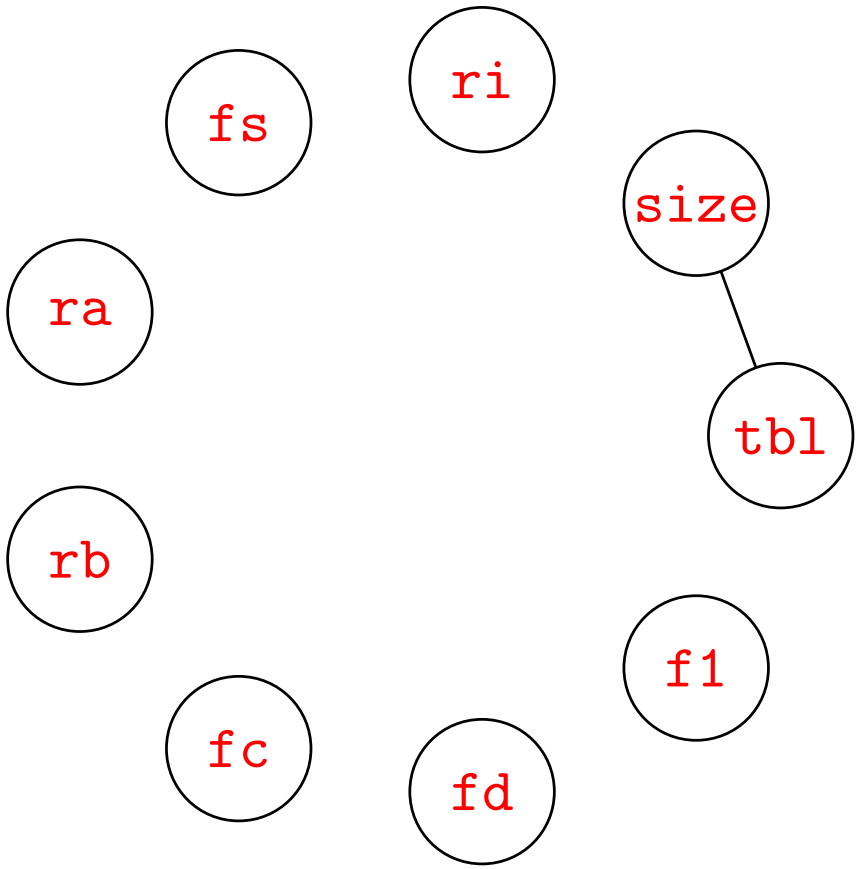
Construction efficace du graphe d'interférences:

Pour chaque instruction ℓ de la fonction:

- Si $\ell : r_d = r_s$ (affectation simple), ajouter des arcs entre r_d et tous les registres autres que r_s et r_d vivants en ℓ.
- Sinon, si ℓ est une instruction définissant le registre r: ajouter des arcs entre r et tous les registres autres que r vivants en ℓ.
- Si ℓ est un appel de procédure: ajouter en plus des arcs entre les registres vivants en ℓ et tous les registres réels non préservés pendant l'appel.

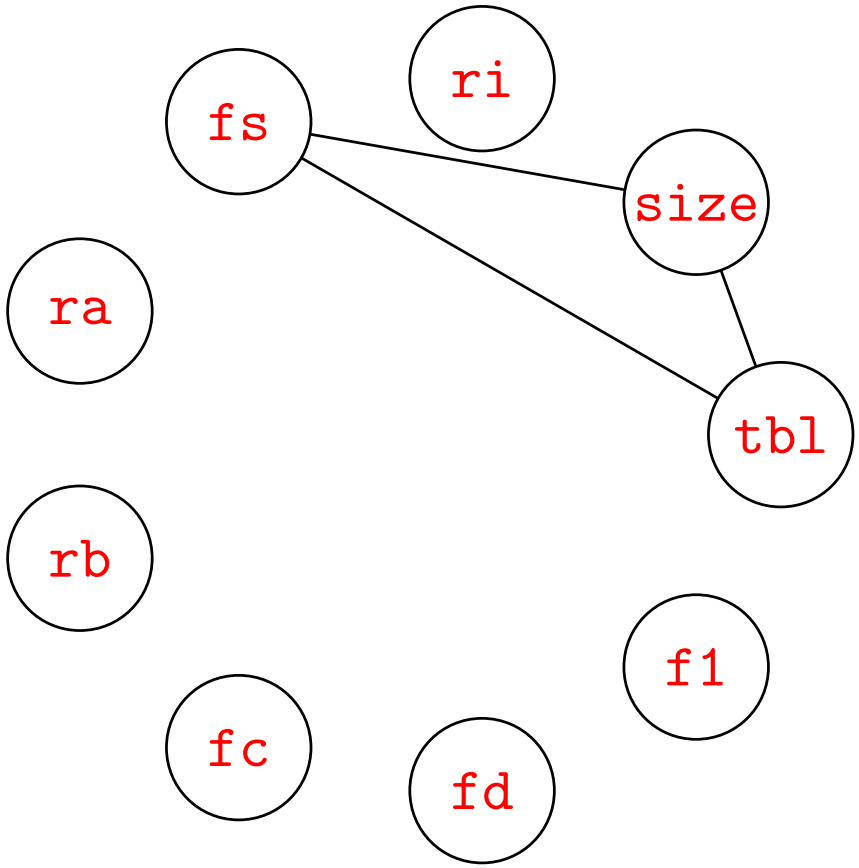
Exemple de graphe d'interférences

```
  fs = ...   [tbl,size,fs]
ri = ...   [tbl,size,ri,fs]
if (ri >= size) ...
ra = ...   [tbl,size,ri,fs,ra]
rb = ...   [tbl,size,ri,fs,rb]
fc = ...   [tbl,size,ri,fs,fc]
fs = ...   [tbl,size,ri,fs]
ri = ...   [tbl,size,ri,fs]
fd = ...   [fs,fd]
f1 = ...   [f1]
return f1
```



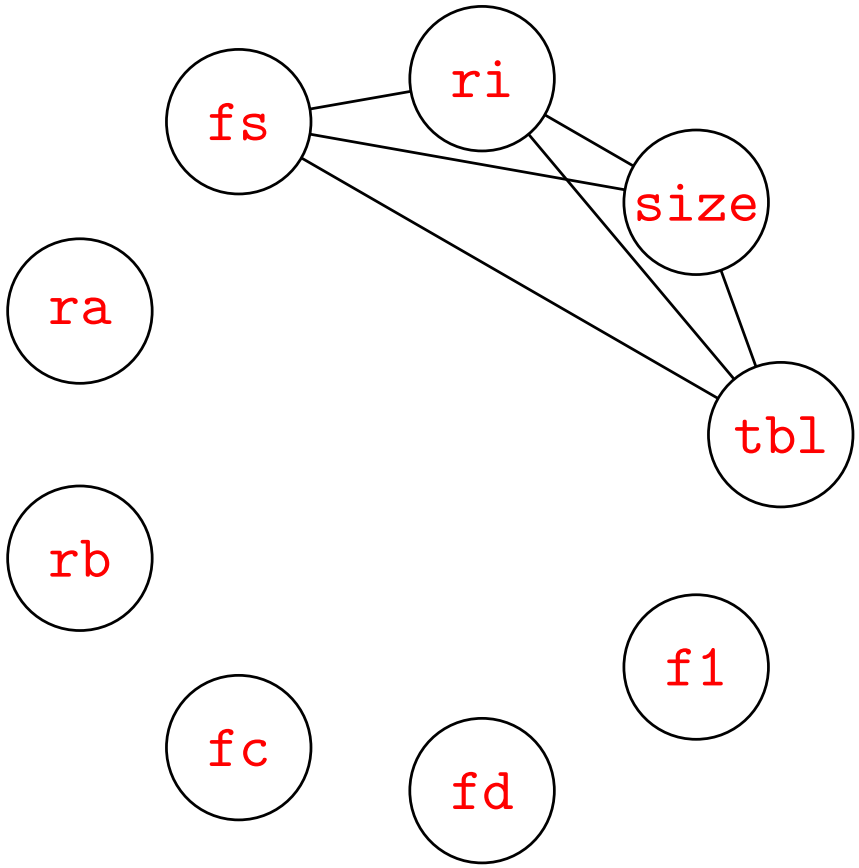
Exemple de graphe d'interférences

```
fs = ...   [tbl,size,fs]
ri = ...   [tbl,size,ri,fs]
if (ri >= size) ...
ra = ...   [tbl,size,ri,fs,ra]
rb = ...   [tbl,size,ri,fs,rb]
fc = ...   [tbl,size,ri,fs,fc]
fs = ...   [tbl,size,ri,fs]
ri = ...   [tbl,size,ri,fs]
fd = ...   [fs,fd]
f1 = ...   [f1]
return f1
```



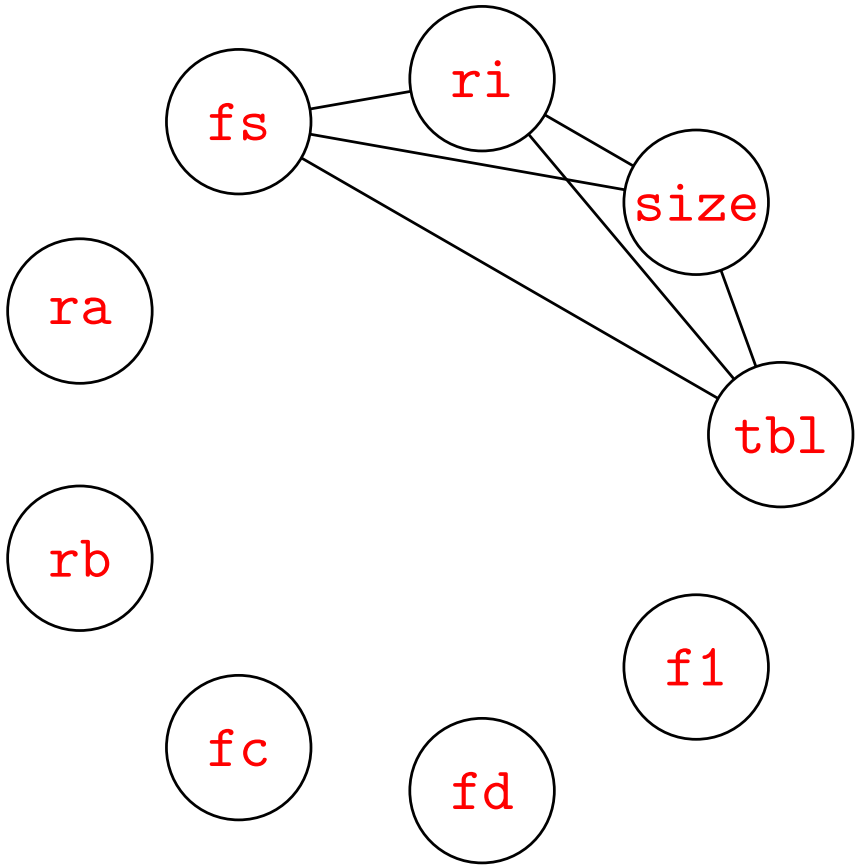
Exemple de graphe d'interférences

```
fs = ...   [tbl,size,fs]
ri = ...   [tbl,size,ri,fs]
if (ri >= size) ...
ra = ...   [tbl,size,ri,fs,ra]
rb = ...   [tbl,size,ri,fs,rb]
fc = ...   [tbl,size,ri,fs,fc]
fs = ...   [tbl,size,ri,fs]
ri = ...   [tbl,size,ri,fs]
fd = ...   [fs,fd]
f1 = ...   [f1]
return f1
```



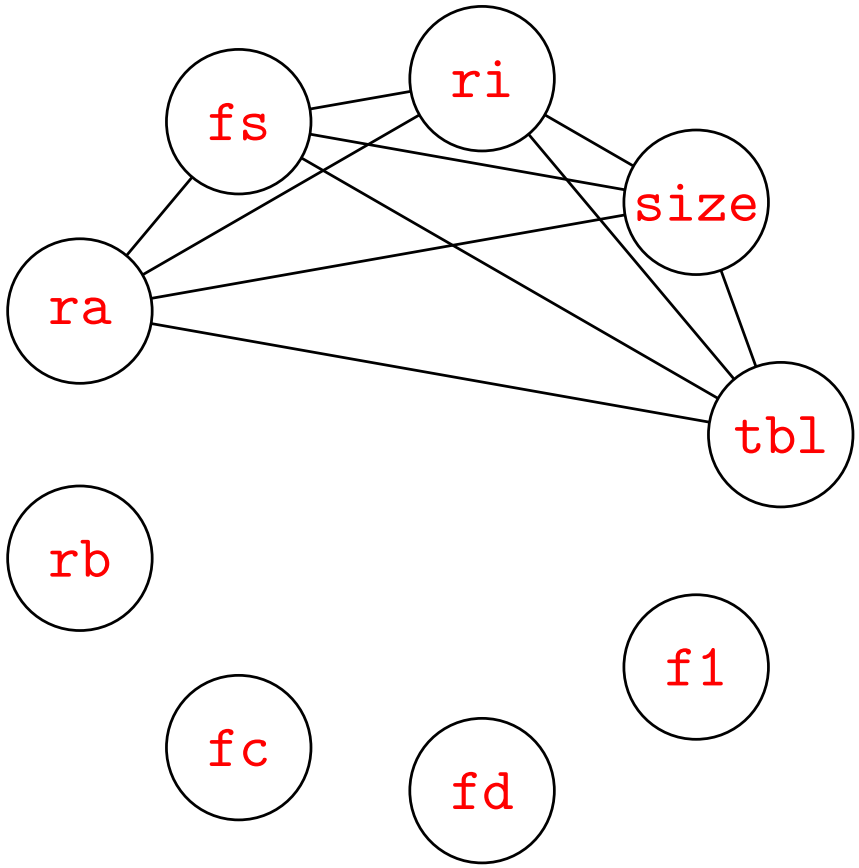
Exemple de graphe d'interférences

```
fs = ...   [tbl,size,fs]
ri = ...   [tbl,size,ri,fs]
if (ri >= size) ...
ra = ...   [tbl,size,ri,fs,ra]
rb = ...   [tbl,size,ri,fs,rb]
fc = ...   [tbl,size,ri,fs,fc]
fs = ...   [tbl,size,ri,fs]
ri = ...   [tbl,size,ri,fs]
fd = ...   [fs,fd]
f1 = ...   [f1]
return f1
```



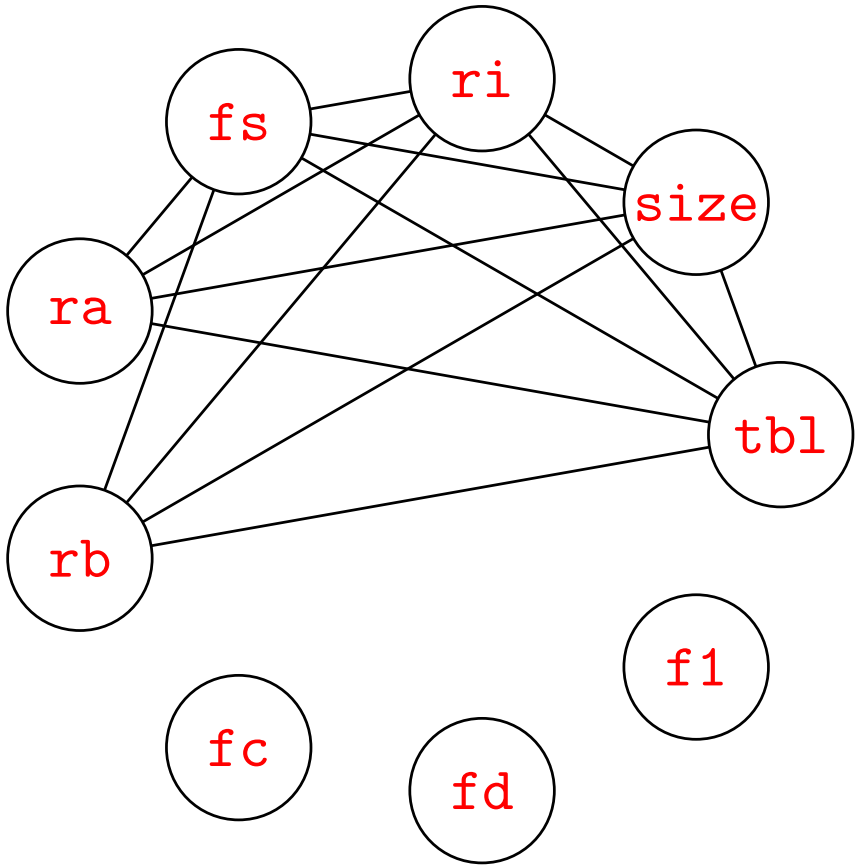
Exemple de graphe d'interférences

```
fs = ...   [tbl,size,fs]
ri = ...   [tbl,size,ri,fs]
if (ri >= size) ...
ra = ...   [tbl,size,ri,fs,ra]
rb = ...   [tbl,size,ri,fs,rb]
fc = ...   [tbl,size,ri,fs,fc]
fs = ...   [tbl,size,ri,fs]
ri = ...   [tbl,size,ri,fs]
fd = ...   [fs,fd]
f1 = ...   [f1]
return f1
```



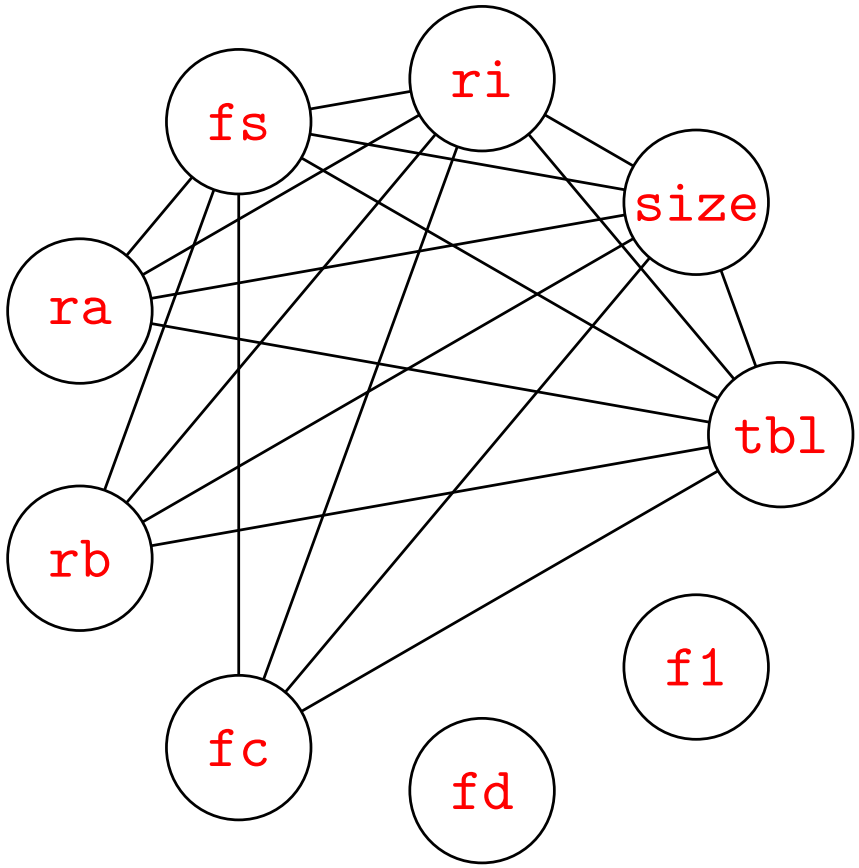
Exemple de graphe d'interférences

```
fs = ...   [tbl,size,fs]
ri = ...   [tbl,size,ri,fs]
if (ri >= size) ...
ra = ...   [tbl,size,ri,fs,ra]
rb = ...   [tbl,size,ri,fs,rb]
fc = ...   [tbl,size,ri,fs,fc]
fs = ...   [tbl,size,ri,fs]
ri = ...   [tbl,size,ri,fs]
fd = ...   [fs,fd]
f1 = ...   [f1]
return f1
```



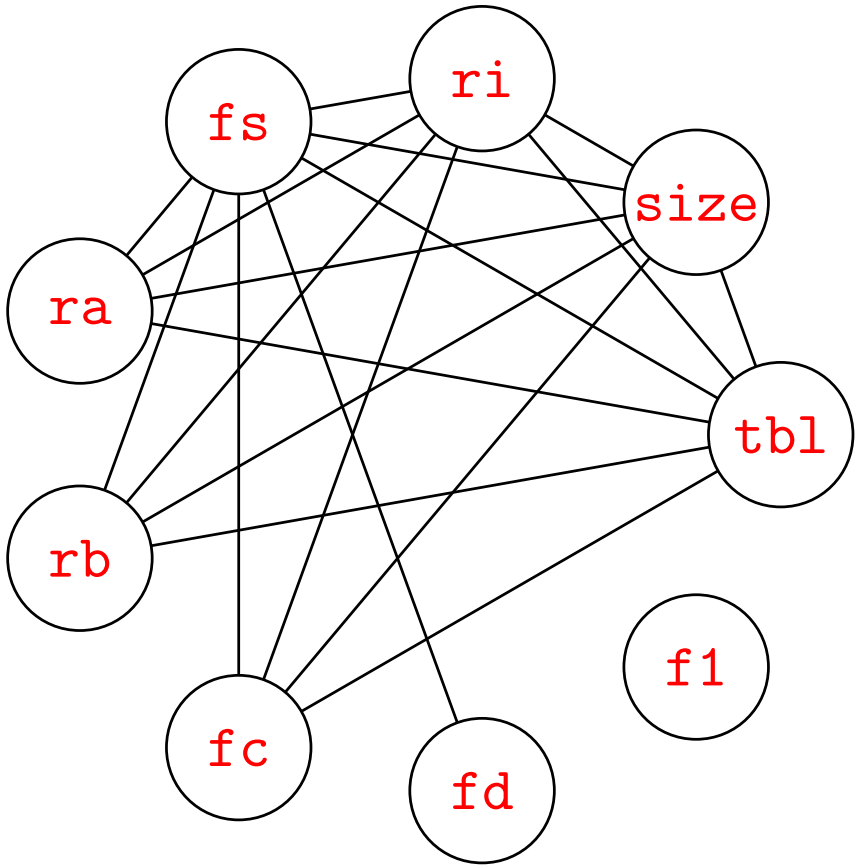
Exemple de graphe d'interférences

```
fs = ...   [tbl,size,fs]
ri = ...   [tbl,size,ri,fs]
if (ri >= size) ...
ra = ...   [tbl,size,ri,fs,ra]
rb = ...   [tbl,size,ri,fs,rb]
fc = ...   [tbl,size,ri,fs,fc]
fs = ...   [tbl,size,ri,fs]
ri = ...   [tbl,size,ri,fs]
fd = ...   [fs,fd]
f1 = ...   [f1]
return f1
```



Exemple de graphe d'interférences

```
fs = ...   [tbl,size,fs]
ri = ...   [tbl,size,ri,fs]
if (ri >= size) ...
ra = ...   [tbl,size,ri,fs,ra]
rb = ...   [tbl,size,ri,fs,rb]
fc = ...   [tbl,size,ri,fs,fc]
fs = ...   [tbl,size,ri,fs]
ri = ...   [tbl,size,ri,fs]
fd = ...   [fs,fd]
f1 = ...   [f1]
return f1
```



Heuristique de coloriage du graphe

Il faut attribuer à chaque noeud un registre réel (ou si pas possible un emplacement de pile) distinct de ceux attribués à ses voisins.

Lemme de Kempe: si un noeud n d'un graphe G est de degré $< k$, et si $G \setminus n$ est k-colorable, alors G est k-colorable.

Soit N le nombre de registres réels disponibles. On peut donc enlever du graphe d'interférence tous les noeuds de degré $< N$, et les colorier en dernier: il sera toujours possible de leur attribuer un registre.

Coloriage optimiste (Briggs et al)

S: une pile de noeuds, initialement vide.

Tant que le graphe G contient des pseudo-registres:
  Choisir un pseudo-registre $r \in G$,
    si possible de degré $< N$, sinon de degré maximal.
  Empiler r sur S.
  Enlever r de G.
Fin Tant que

Tant que la pile S n'est pas vide:
  Dépiler un pseudo-registre r depuis S.
  Remettre r dans le graphe G.
  Attribuer à r un registre physique différent de tous ceux
    attribués à ses voisins dans G.
  Si impossible, attribuer à r un emplacement de pile différent
    de tous ceux attribués à ses voisins dans G.
Fin Tant que

Transformation du code

Analyse de durée de vie; construction du graphe d'interférence; coloriage de ce dernier
$\longrightarrow$ une affectation de registres $\sigma : \texttt{reg} \longrightarrow \texttt{reg}$.

Transformation du code:

- On renomme les registres comme indiqué par σ.
- On remplace par `nop` les instructions pures (`op`, `load`) dont le registre résultat n'est pas vivant.
- On remplace par `nop` les instructions $r_d = r_s$ telles que $\sigma(r_s) = \sigma(r_d)$.

Correction de la transformation

Le registre r du code initial est renommé en $\sigma(r)$ dans le code optimisé.

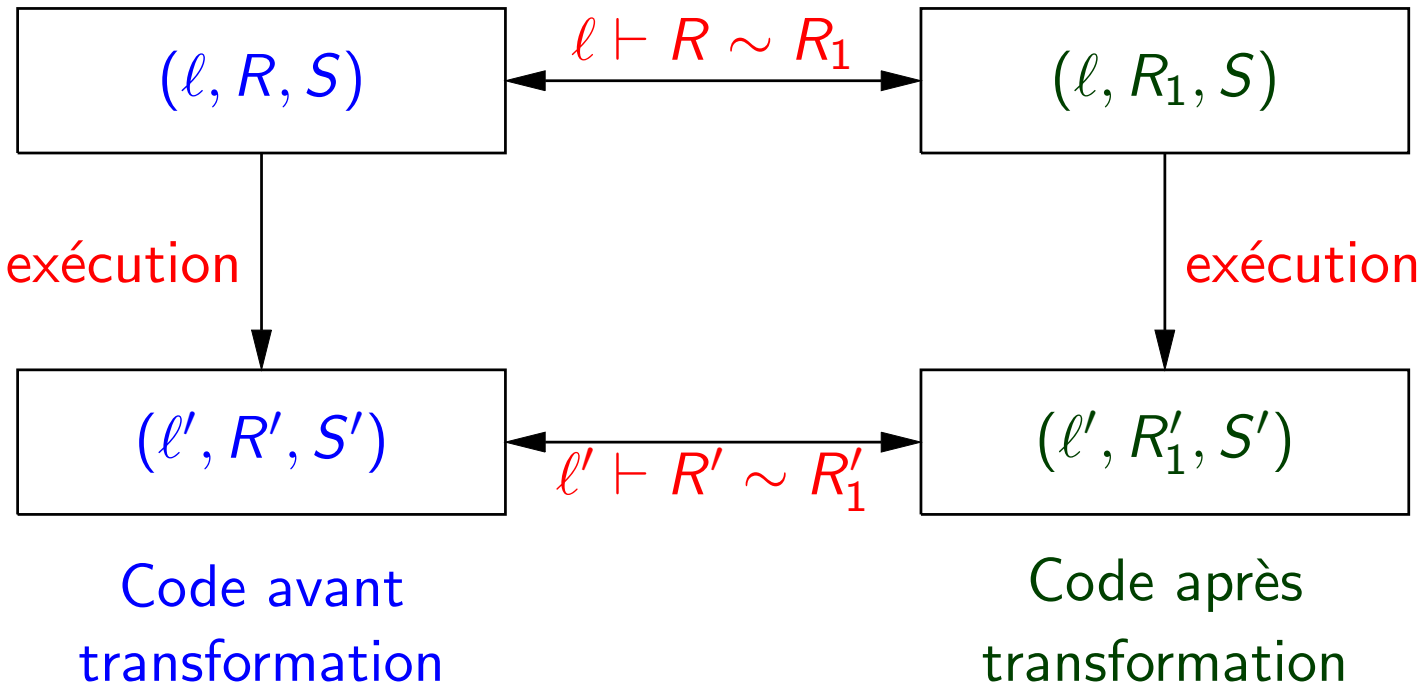
Pour que ce soit correct, il faut que la valeur de r dans le code initial soit égale à la valeur de $\sigma(r)$ dans le code transformé à chaque point où r est vivant.

Formellement, si A est le résultat de l'analyse de durée de vie:

$$\ell \vdash R_1 \sim R_2 \textit{ ssi } R_1(r) = R_2(\sigma(r)) \textit{ pour tout } r \in A(\ell)$$

Le résultat de simulation

Si $P, f \vdash (\ell, R, S) \rightarrow (\ell', R', S')$ et $\ell \vdash R \sim R_1$, alors il existe R'_1 tel que $\texttt{transf}(P), \texttt{transf}(f) \vdash (\ell, R_1, S) \rightarrow (\ell', R'_1, S')$ et $\ell' \vdash R' \sim R'_1$.



Plan du cours

Compiler pour calculer plus vite

La recherche traditionnelle en compilation (augmenter la vitesse d'exécution de Fortran et C) s'essoufle:

- Énormément de travaux antérieurs.
- Les gains sont de plus en plus faibles pour des efforts de plus en plus considérables.
- Le modèle de performances des processeurs modernes est extrêmement complexe.

Il reste quelques niches:

- Langages généralistes de haut niveau (Java, Caml, Haskell, ...): reste à gagner le dernier facteur 2.
- Langages exotiques (Esterel, hardware description languages), langages dédiés (domain specific languages): se prêtent à des optimisations spectaculaires.

Compiler pour calculer plus vite

La recherche traditionnelle en compilation (augmenter la vitesse d'exécution de Fortran et C) s'essoufle:

- Énormément de travaux antérieurs.
- Les gains sont de plus en plus faibles pour des efforts de plus en plus considérables.
- Le modèle de performances des processeurs modernes est extrêmement complexe.

Il reste quelques niches:

- Langages généralistes de haut niveau (Java, Caml, Haskell, ...): reste à gagner le dernier facteur 2.
- Langages exotiques (Esterel, hardware description languages), langages dédiés (domain specific languages): se prêtent à des optimisations spectaculaires.

Un renouveau de l'analyse statique

Analyser statiquement les programmes non pas pour trouver des inefficacités à réduire, mais pour trouver des bugs potentiels ou établir automatiquement des propriétés de sûreté ou de sécurité.

Souvent combiné avec des techniques de vérification automatique (software model checking).

Quelques beaux exemples: Astrée (ENS), SLAM/SDV (MSR), FiSC (Stanford).

Vérifier formellement le compilateur ou l'analyseur statique

Un besoin naissant dans le monde des méthodes formelles.

Hier: preuves de compilateurs de bytecode (non optimisants), de machines abstraites, de vérificateurs de bytecode.

Aujourd'hui: beaucoup d'efforts en cours sur la preuve d'analyses statiques et de transformations de programmes.

Demain: vérification intégrale de compilateurs natifs optimisants?

Après-demain: vers la vérification de tous les outils qui interviennent dans la production de logiciels critiques?
(analyseurs statiques, model checkers, prouveurs de programmes, générateurs de code, ...)