Formation accélérée en compilation

Xavier Leroy
INRIA Rocquencourt

Première partie

Une présentation simplifiée du PowerPC

Mémoire et registres

Mémoire: fonction adresses $\rightarrow$ octets $[0 \ldots 255]$

Peut être accédée par groupes de 1, 2, 4 ou 8 octets

Registres:

$r_0 \ldots r_{31}$ registres généraux de 32 bits

$f_0 \ldots f_{31}$ registres flottants de 64 bits IEEE 754

$cr_0 \ldots cr_7$ registres de conditions (=, <, etc)

lr adresse de retour d'une procédure

ctr pour les branchements calculés

(Plus quelques autres sans intérêt.)

Instructions

Transfers et arithmétique entre registres:

```
mr   rd, rs                    rd := rs
fmr  fd, fs                    fd := fs
mflr rd                        rd := lr
mtlr rs                        lr := rs
li   rd, N                     rd := N
add  rd, rs1, rs2              rd := rs1 + rs2
fmul fd, fs1, fs2              fd := fs1 *f fs2
fmadd fd, fs1, fs2, fs3        fd := fs1 *f fs2 +f fs3
addi rd, rs, N                 rd := rs + N
cmp  crd, rs1, rs2             crd := compare(rs1, rs2)
cmpi crd, rs, N                crd := compare(rs1, N)
```

`N` est un entier entre -32768 et 32767

Instructions, suite

Lecture et écriture en mémoire:

```
lwz  rd, ra, N          rd := 32 bits à l'adresse ra + N
stw  rs, ra, N          32 bits à l'adresse ra + N := rs
lbz  rd, ra1, ra2       rd := 8 bits à l'adresse ra1+ra2
lfd  fd, ra, N          fd := 64 bits à l'adresse ra + N
stfs fs, ra, N          32 bits à l'adresse ra + N := fs
```

Branchements:

```
b    étiquette                  branchement inconditionnel
bc   cond, cr, étiquette        branchement conditionnel
bl   étiquette                  branchement, retour dans lr
blr                             branchement calculé à lr
```

Conventions pour le code compilé

Pointeur de pile: dans `r1`.

La pile grandit vers les adresses décroissantes.

Entrée dans une procédure:

`lr` contient l'adresse de retour

`r3-r10` contiennent les arguments de type entier ou pointeur

`f1-f8` contiennent les arguments de type flottant

Sortie d'une procédure:

résultat dans `f1` si flottant, dans `r3` si entier ou pointeur.

Registres qui doivent être préservés par la procédure:

`r1` (pointeur de pile), `r2`, `r13-r31`, `f14-f31`

Les autres registres peuvent voir leur valeur modifiée par la procédure

Exemple: la fonction de Fibonacci

```
fibonacci:
        addi    r1, r1, -12       allouer 12 octets de pile
        mflr    r0
        stw     r0, 0(r1)         sauvegarder lr en pile
        cmpi    cr0, r3, 2        argument >= 2 ?
        bc      >=, cr0, L1       si oui, aller en L1
        li      r3, 1             si non, résultat = 1
        b       L2                aller en L2
L1:
        stw     r3, 4(r1)         sauvegarder l'argument
        addi    r3, r3, -1        appel récursif sur argument - 1
        bl      fibonacci
        lwz     r4, 4(r1)         recharger l'argument
        stw     r3, 8(r1)         sauver fibonacci(argument - 1)
        addi    r3, r4, -2        appel récursif sur argument - 2
        bl      fibonacci
        lwz     r4, 8(r1)         recharger fibonacci(argument - 1)
        add     r3, r3, r4        l'ajouter à fibonacci(argument - 2)
L2:
        lwz     r0, 0(r1)         recharger lr
        addi    r1, r1, 12        restaurer le pointeur de pile
        mtlr    r0
        blr                       retour à l'appelant
```

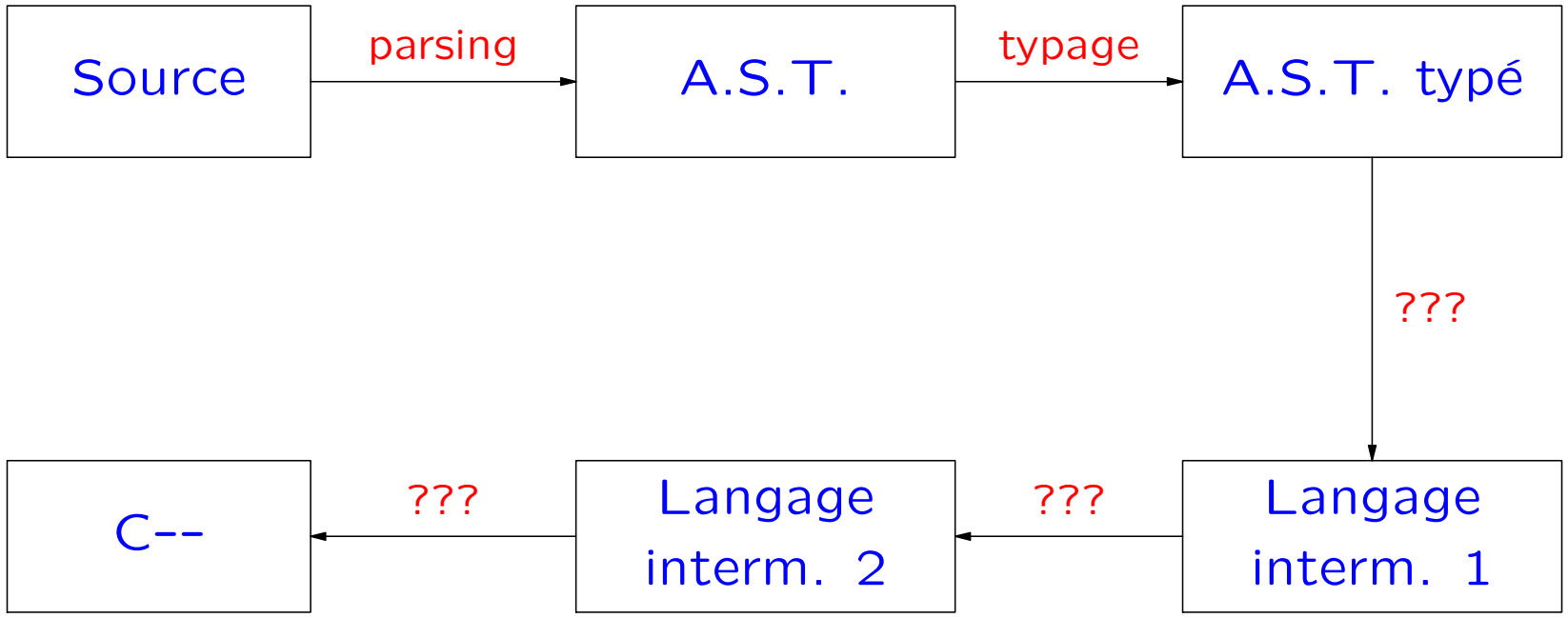
Même exemple avec des registres sauvegardés

```
fibonacci:
        addi    r1, r1, -12       allouer 12 octets de pile
        mflr    r0
        stw     r0, 0(r1)         sauvegarder lr en pile
        stw     r14, 4(r1)        sauvegarder r14
        stw     r15, 8(r1)        sauvegarder r15
        cmpi    cr0, r3, 2        argument >= 2 ?
        bc      >=, cr0, L1       si oui, aller en L1
        li      r3, 1             si non, résultat = 1
        b       L2                aller en L2
L1:     mov     r14, r3           sauvegarder l'argument dans r14
        addi    r3, r3, -1        appel récursif sur argument - 1
        bl      fibonacci
        mov     r15, r3           sauver fibonacci(argument - 1)
        addi    r3, r14, -2       appel récursif sur argument - 2
        bl      fibonacci
        add     r3, r3, r15       calcul du résultat
L2:     lwz     r0, 0(r1)         recharger lr
        lwz     r14, 4(r1)        recharger r14
        lwz     r15, 8(r1)        recharger r15
        addi    r1, r1, 12        restaurer le pointeur de pile
        mtlr    r0
        blr                       retour à l'appelant
```

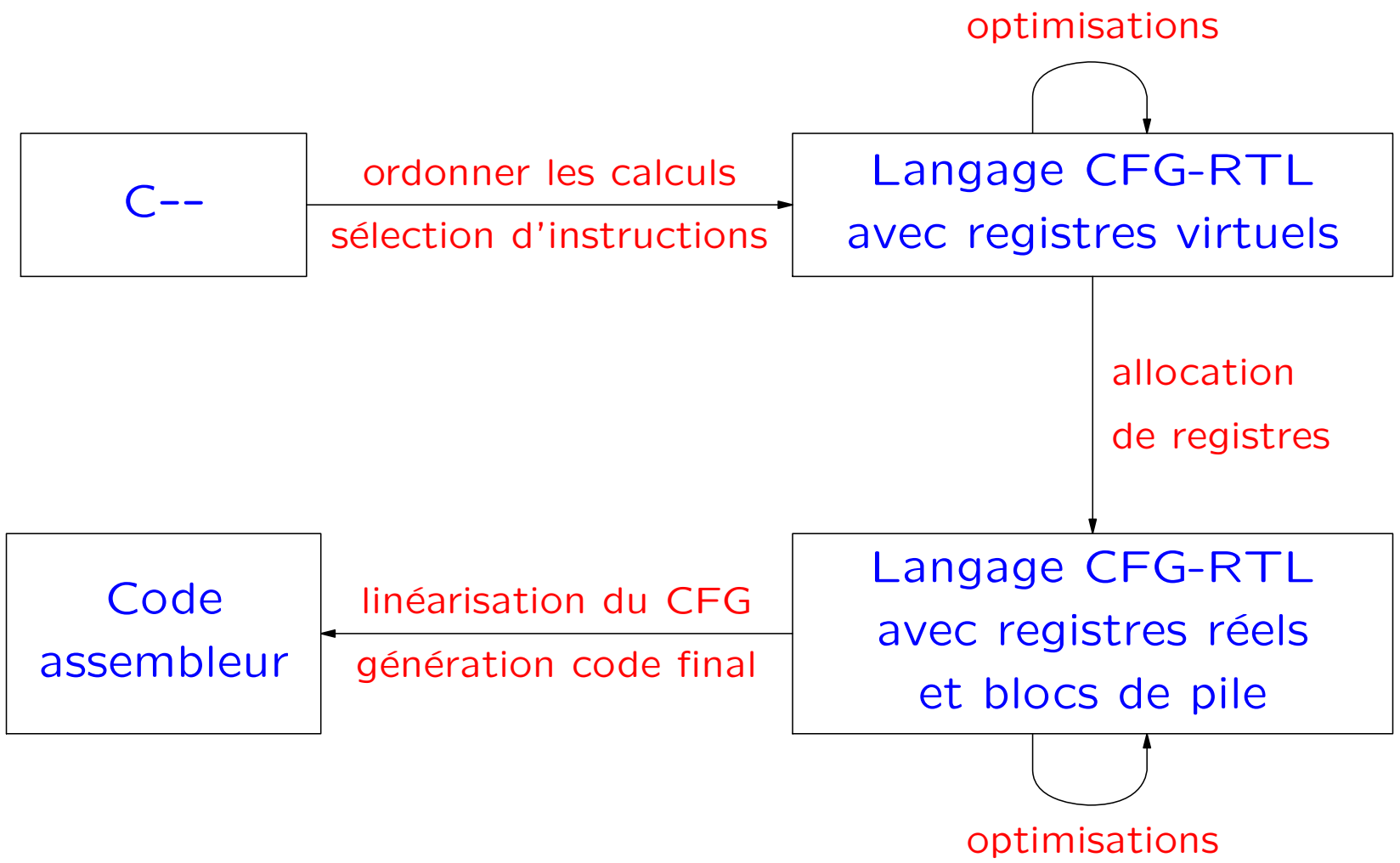
Deuxième partie

Vue aérienne d'un compilateur natif optimisant

Le front-end


Source
parsing
A.S.T.
typage
A.S.T. typé
???
C--
???
Langage
interm. 2
???
Langage
interm. 1

Le back-end


optimisations
C--
ordonner les calculs
sélection d'instructions
Langage CFG-RTL
avec registres virtuels
allocation
de registres
Code
assembleur
linéarisation du CFG
génération code final
Langage CFG-RTL
avec registres réels
et blocs de pile
optimisations

Le langage intermédiaire C--

Langage impératif, à base d'expressions et d'instructions (*statements*).

Très faiblement typé: entiers+pointeurs / flottants.

Conversions explicites; surcharge des opérateurs résolue.

Calculs d'adresses explicites (pour les tableaux, les records, etc).

Structures de contrôle simplifiées.

Le cas échéant: "support" pour le GC, les exceptions.

Exemple de C--

Code source en C:

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

Code C-- équivalent:

```
average(tbl: ptr, size: int) : double {
    s : double;
    i : int;
    s = 0.0;
    i = 0;
    loop {
        if (i >= size) break;
        s = s +f float_of_int(load_int32(tbl + i * 4))
        i = i + 1
    }
    return s /f float_of_int(size);
}
```

Le langage intermédiaire CFG-RTL

CFG = Control-Flow Graph
RTL = Register Transfer Language

Noeuds du graphe: opérations élémentaires

(correspondant à peu près à des instructions machines)

Arguments et résultats des opérations: dans des registres

registres réels: correspondant à ceux du processeur

registres virtuels: des variables qu'il faudra plus tard placer dans des registres réels ou des emplacements de pile.

Arcs du graphe: le(s) successeur(s) de l'opération dans le flot de contrôle

opérations arithmétiques: 1 successeur

branchements conditionnels: 2 sucesseurs

branchements à travers une table (`switch`): N successeurs

`return`: 0 successeurs

Exemple de CFG-RTL avec des registres abstraits

```
I1:  rtbl = r3;                  --> I2
I2:  rsize = r4;                 --> I3
I3:  fs = 0.0;                   --> I4
I4:  ri = 0;                     --> I5
I5:  if (ri >= rsize)            --> I11 si oui, I6 si non
I6:  ra = ri * 4                 --> I7
I7:  rb = load_int32(rtbl + ra) --> I8
I8:  fc = float_of_int(rb)       --> I9
I9:  fs = fs +f fc               --> I10
I10: ri = ri + 1                 --> I5
I11: fd = float_of_int(rsize)    --> I12
I12: f1 = fs /f fd               --> I13
I13: return f1
```

La sélection d'instructions

Passage de C-- à RTL-CFG-abstrait.

Traitement du contrôle: le transformer en arcs du CFG

Traitement des données:

- Décomposer les expressions en opérations élémentaires

```
s = s +f float_of_int(load_int32(ptr + i * 4))

-->  i * 4,   load_int32(ptr + ...),   float_of_int(...),   s = s +f ...
```

- Choisir l'ordre des calculs.
- Introduire des registres virtuels pour les résultats intermédiaires.
- Les connecter aux registres réels utilisés pour les appels de fonctions.

Optimisations du CFG-RTL abstrait

La plupart des optimisations s'effectuent sur le CFG-RTL abstrait.

- Constant propagation

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

- Dead code elimination

```
ra = 1
rb = 2                  -->     rb = 2
rc = 3                          rc = 3
// ra inutilisé ensuite
```

- 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.

Dataflow analysis

Ces transformations s'appuient en général sur une analyse préalable du code.

États abstraits: registre $\rightarrow$ élément d'un treillis.

Pour chaque instruction I: état abstrait avant exécution $in(I)$, après exécution $out(I)$.

Équations dataflow en avant:

$$\begin{aligned} out(I) &= T(I, in(I)) \\ in(I) &= lub\ \{out(J) \mid J \text{ prédecesseur de } I\} \\ in(I_0) &= \ldots \end{aligned}$$

Dataflow analysis

Équations dataflow en arrière:

$$\begin{aligned} in(I) &= T(I, out(I)) \\ out(I) &= lub\ \{in(J) \mid J \text{ successeur de } I\} \\ in(I_{\texttt{return}}) &= \ldots \end{aligned}$$

Remarque: dataflow arrière $\approx$ dataflow avant sur le dual du CFG.

Les équations dataflow se résolvent par itération de point fixe.

Exemple d'équations dataflow

La propagation des constantes:

Treillis $\mathbf{Z} + \{\top, \bot\}$.

$in(I_0) = \lambda r.\top$

Si $I : rc = ra + rb$, $\quad out(I) = in(I) + \{rc \mapsto v\}$ avec

$in(I)(ra)$	$in(I)(rb)$	v
$\bot$	$\bot$	$\bot$
$n_a \in \mathbf{Z}$	$n_b \in \mathbf{Z}$	$n_a + n_b$
$\top$	_	$\top$
_	$\top$	$\top$

Résolution des équations dataflow

Algorithme de la worklist (Kildall):

```
in(I_0) = ...
in(I) = λr.⊥  pour tout I ≠ I_0
out(I) = λr.⊥  pour tout I
W = { I_0 }

Tant que W n'est pas vide:
    Choisir I dans W
    W = W \ { I }
    out(I) = T(I, in(I))
    Pour tout successeur J de I:
        t = lub(in(J), out(I))
        Si t ≠ in(J):
            in(J) = t
            W = W ∪ { J }
        Fin Si
     Fin Pour tout
Fin Tant que
```

L'allocation de registres

Passage de CFG-RTL abstrait à CFG-RTL concret: attribuer des registres réels ou des emplacements de pile à tous les registres virtuels de la procédure.

Idée: deux registres virtuels peuvent partager le même registre réel s'ils n'interfèrent pas, c.à.d. ils ne sont pas vivants en un même point du programme.

Un registre est vivant en un point s'il a été défini auparavant et est utilisé plus tard.

Allocation de registres par coloriage de graphe:

- Analyse de vivacité (*liveness analysis*)
- Construction d'un graphe d'interférences
- Coloriage de ce graphe avec des registres réels et des emplacements de pile

Liveness analysis

Une analyse de type dataflow arrière.

Treillis: $\top$ (vivant), $\bot$ (mort).

Si $I : r_d = op(r_1, \ldots, r_n)$,

$$in(I) = (out(I) + \{r_d \mapsto \bot\}) + \{r_1, \ldots, r_n \mapsto \top\}$$

(L'instruction définit r_d, donc r_d n'est plus vivant avant.
L'instruction utilise $r_1, \ldots, r_n$, qui doivent donc être vivants avant.
Les autres registres sont inchangés.)

Points de sortie: si I : `return` r,
$out(I) = \{\_ \mapsto \bot\}$ et $in(I) = \{r \mapsto \top; \_ \mapsto \bot\}$.

(Traiter les appels de procédures comme des points de sortie.)

Exemple de liveness analysis

```
{ r3, r4 }
        I1:  rtbl = r3;                    --> I2
{ rtbl, r4 }
        I2:  rsize = r4;                   --> I3
{ rtbl, rsize }
        I3:  fs = 0.0;                     --> I4
{ rtbl, rsize, fs }
        I4:  ri = 0;                       --> I5
{ rtbl, rsize, fs, ri }
        I5:  if (ri >= rsize)              --> I11 si oui, I6 si non
{ rtbl, rsize, fs, ri }
        I6:  ra = ri * 4                   --> I7
{ rtbl, rsize, fs, ri, ra }
        I7:  rb = load_int32(rtbl + ra)    --> I8
{ rtbl, rsize, fs, ri, rb }
        I8:  fc = float_of_int(rb)         --> I9
{ rtbl, rsize, fs, ri, fc }
        I9:  fs = fs +f fc                 --> I10
{ rtbl, rsize, fs, ri }
        I10: ri = ri + 1                   --> I5
{ rtbl, rsize, fs, ri }
```

```
{ rsize, fs }
        I11: fd = float_of_int(rsize)   --> I12
{ fs, fd }
        I12: f1 = fs /f fd              --> I13
{ f1 }
        I13: return f1
{  }
```

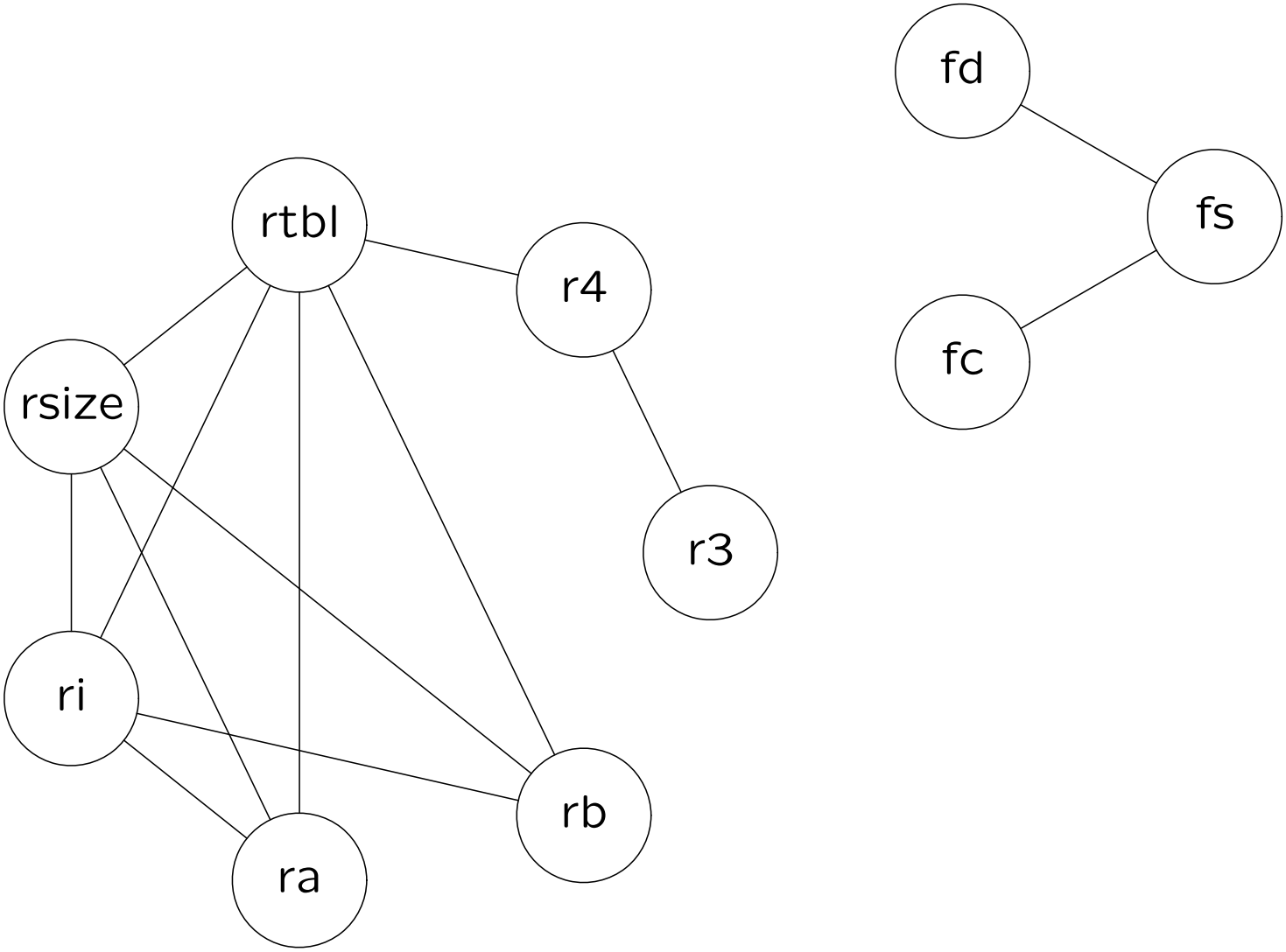
Construction du graphe d'interférence

Graphe non orienté. Noeuds: registres virtuels ou réels. Arc entre 2 registres ssi ils interfèrent.

Pour chaque instruction I du programme:

- Si $I : r_d = op(r_1, \ldots, r_n)$, ajouter des arcs entre r_d et tous les $r \in out(I)$, $r \neq r_d$.
- Si $I : r_d = r_s$ (affectation simple), ne pas ajouter d'arcs.
- Si I est un appel de procédure: ajouter en plus des arcs entre $out(I)$ et tous les registres réels non préservés pendant l'appel.

Exemple de graphe d'interférences



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 registres virtuels:
  Choisir r virtuel ∈ G de degré < N si possible, 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 r de S.
  Remettre r dans G.
  Attribuer à r un registre différent de tous ceux attribués à ses voisins
  dans G.
  Si pas possible, attribuer à r un emplacement de pile différent de
  tous ceux attribués à ses voisins dans G.
Fin Tant que
```

Spilling & reloading

Lorsqu'on place un registre virtuel dans un emplacement de pile, il faut ajuster les instructions qui définissent ou utilisent ce registre:

Supposons `rc` $\longrightarrow$ `stack(0)` and `ra` $\longrightarrow$ `stack(4)` and `rb` $\longrightarrow$ `r3`:

```
                                                          rtmp = reload stack(0)
rc = ra + rb  -->  stack(0) = stack(4) + r3 -->  rtmp = rtmp + r3
                                                          spill(rtmp, stack(4))
```

Les opérations `reload` et `spill` sont juste des lectures et écritures dans la pile.

Nécessite jusqu'à 2 registres temporaires. Où les trouver?

Solution simple: réserver 2 registres du processeur pour cet usage.

Solution fine: réécrire le code CFG-RTL abstrait avec les `load` et `store` supplémentaires, et re-lancer l'allocateur de registre. (Terminaison non garantie.)

Optimisations et transformations sur le CFG-RTL concret

Instruction scheduling:
réordonner les instructions pour mieux exploiter le parallélisme d'instructions.

Linéarisation du CFG:
placer les instructions en séquence, introduire des `goto` lorsque nécessaire pour respecter le flot de contrôle.

Émission du code assembleur:
ajouter un prélude et un épilogue à chaque procédure
(allocation/désallocation du bloc de pile; sauvegarde et restauration de l'adresse de retour et des registres préservés)
traduire les noeuds du RTL en syntaxe assembleur.

Troisième partie

Récapitulation sur les langages intermédiaires

Langages intermédiaires intéressants

- C--

  Point de passage obligé pour presque tous les langages source.

- CFG-RTL abstrait et concret.

  Pour toutes les optimisations classiques.

- Forme SSA.

  Voir transparents suivants.

- Forme ANF.

  Voir transparents suivants.

Autre langage intermédiaire possible: la forme SSA

Single Static Assignment: comme le CFG-RTL abstrait, mais chaque pseudo-registre est affecté une seule fois.

Introduction de "noeuds φ" aux points de jointure du graphe de contrôle.

Avantage: les analyses dataflow sont plus simples et plus rapides.

Inconvénient: la sémantique dynamique des noeuds φ est peu claire.

Exemple de forme SSA

```
I1:  rtbl = r3;                    --> I2
I2:  rsize = r4;                   --> I3
I3:  fs1 = 0.0;                    --> I4
I4:  ri1 = 0;                      --> I5
I5:  ri2 = φ(I4:ri1, I11:ri3)
     fs2 = φ(I4:fs1, I11:fs3)      --> I6
I6:  if (ri2 >= rsize)             --> I12 si oui, I7 si non
I7:  ra = ri2 * 4                  --> I8
I8:  rb = load_int32(rtbl + ra)    --> I9
I9:  fc = float_of_int(rb)         --> I10
I10: fs3 = fs2 +f fc               --> I11
I11: ri3 = ri2 + 1                 --> I5
I12: fd = float_of_int(rsize)      --> I13
I13: f1 = fs2 /f fd                --> I14
I14: return f1
```

Autre langage intermédiaire possible: la forme ANF

A-normal forms: un λ-calcul où tous les résultats intermédiaires sont nommés par `let` ou λ. Les expressions permises sont celles qui correspondent à des noeuds du graphe CFG-RTL. Utilisation de λ pour capturer les branchements et les points de jointure.

Folklore: équivalent à la forme SSA.

Encore mal compris.

Exemple de forme ANF

```
average (r3, r4) =
  let rtbl = r3 in let rsize = r4 in let fs = 0.0 in let ri = 0 in
  f rtbl rsize fs ri

f (rtbl, rsize, fs, ri) =
  if (ri >= rsize) then g (rtbl, rsize, fs, ri) else
    let ra = ri * 4 in
    let rb = load_int32(rtbl + ra) in
    let fc = float_of_int(rb) in
    let fs' = fs +f fc in
    let ri' = ri + 1 in
    f (rtbl, rsize, fs', ri')

g (rtbl, rsize, fs, ri) =
  let fd = float_of_int(rsize) in
  let f1 = fs /f fd in
  return f1
```

Pour aller plus loin

Alfred V. Aho and Ravi Sethi and Jeffrey D. Ullman, *Compilers: principles, techniques, and tools*, Addison-Wesley, 1986. Chapitres 8, 9, 10.

Andrew W. Appel, *Modern compiler implementation in C* ou in Java ou in ML, Cambridge University Press, 1998.

Cours de compilation de Didier Rémy,
`http://pauillac.inria.fr/~remy/poly/compil/`

Steven S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann, 1997.