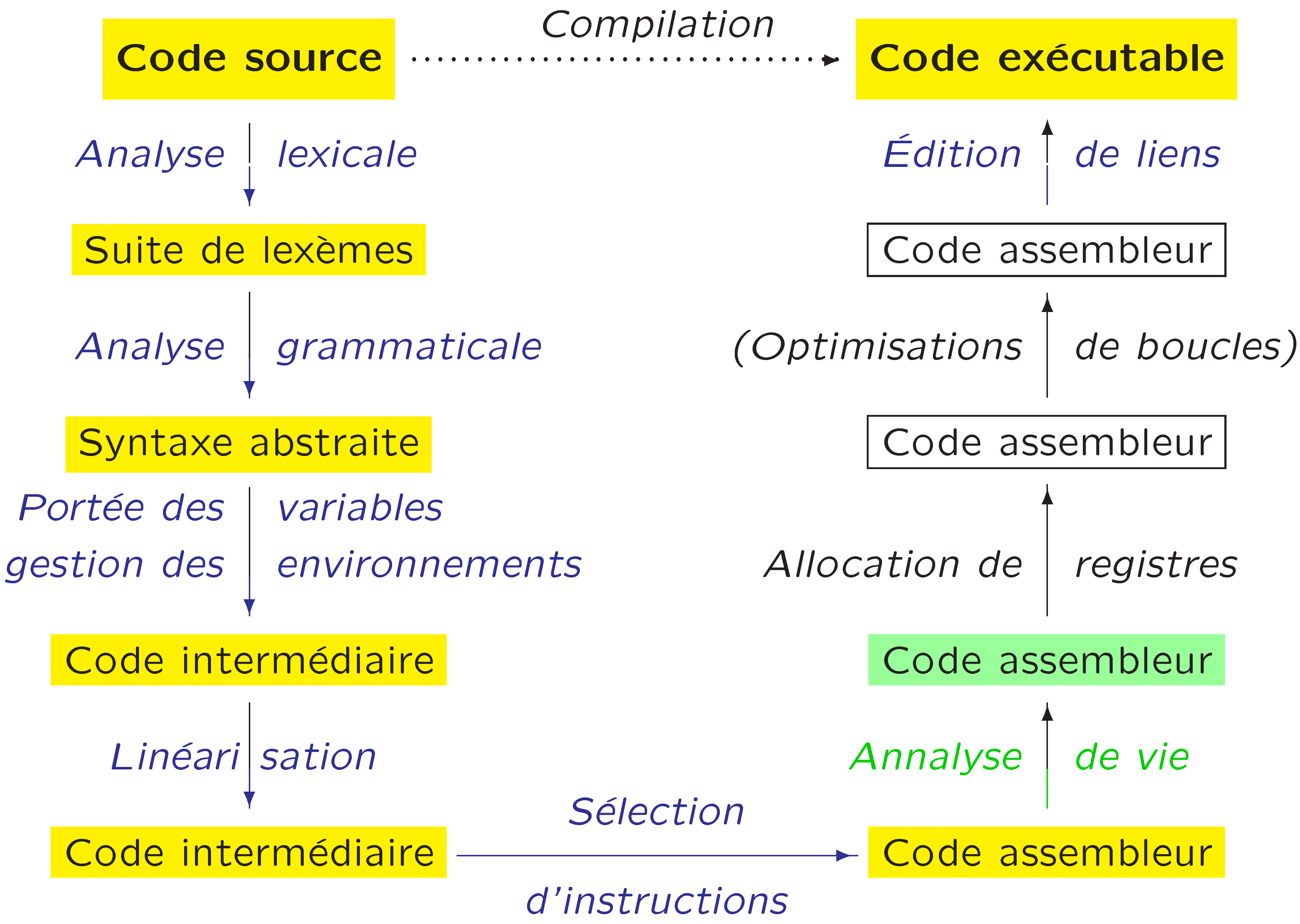
Analyse de durée de vie

Didier Rémy

Octobre 2000

http://cristal.inria.fr/~remy/poly/compil/5/

http://w3.edu.polytechnique.fr/profs/informatique/Didier.Remy/compil/5/


Code source
Compilation
Code exécutable
Analyse lexicale
Suite de lexèmes
Analyse grammaticale
Syntaxe abstraite
Portée des variables
gestion des environnements
Code intermédiaire
Linéarisation
Code intermédiaire
Sélection d'instructions
Code assembleur
Annalyse de vie
Code assembleur
Allocation de registres
Code assembleur
(Optimisations de boucles)
Code assembleur
Édition de liens

Sa place dans la chaîne de compilation

$$\text{SA} \overbrace{\xrightarrow{\text{Générat. Canon.}}}^{(1)} \text{CI} \overbrace{\xrightarrow{\text{Sélection d'instruct.}} \text{CA} \underbrace{\xrightarrow{\text{Durée de vie}} \text{CA} \boxed{\xrightarrow{\text{Coloriage}}}}_{\text{Allocation de registres}}}^{(2)} \text{CA}$$

Calculer pour chaque instruction la liste des temporaires vivants à la fin de l'instruction :

$\Rightarrow$ Les temporaires vivant simultanément ne peuvent pas être identifiés. En particulier, ils ne peuvent pas utiliser le même registre.

$\Rightarrow$ Les autres peuvent partager le même registre.

L'analyse de la durée se retrouve sous d'autres formes (analyse de flow) en aval pour faire des optimisations plus poussées.

Le problème

Chaque instruction
- **utilise** (lit) un ensemble de temporaires, calcule et
- **définit** (écrit) un ensemble de temporaires.

Les deux ensembles ne sont pas nécessairement disjoints.

Un temporaire est **vivant** à la sortie d'une instruction i si sa valeur est utilisée dans une instruction suivante (au sens large) avant d'être redéfinie. La **durée de vie** d'un temporaire est l'ensemble des instructions où il est vivant.

Lorsqu'un temporaire est mort, sa valeur n'a pas d'importance. Deux temporaires qui ont des durées de vie disjointes peuvent être superposés et utiliser le même registre.

Le calcul de la durée de vie est un préliminaire à l'allocation de registres.

Exemples

$\boxed{z \leftarrow x + y}$ utilise les variables x, y et définit z.

Analogie temporaire $\longleftrightarrow$ variable globale

Groupements On peut traiter un groupe d'instructions linéaires (sans saut ni entrée), en particulier un bloc élémentaire, comme une "macro-instuction" :

$$\boxed{\begin{array}{l} z \leftarrow x + y \\ t \leftarrow z \end{array}}$$

utilise x, y

définit z, t

Important : la valeur locale de z est utilisée dans le bloc, mais pas sa valeur à l'entrée du bloc ! Donc z n'est pas lu par le bloc.

L'instruction call se comporte comme une instruction qui

– lit $a0$, $a1$, etc. (selon le nombre d'arguments).
– écrit ra, les registres spéciaux, les registres t, v, et a.

Le graphe d'un programme

Un programme peut être vu comme un graphe dont les nœuds sont les instructions et les arcs décrivent la possibilité de passer d'une instruction à une autre et sont étiquetés par des conditions (mutuellement exclusives —déterminées à l'évaluation).

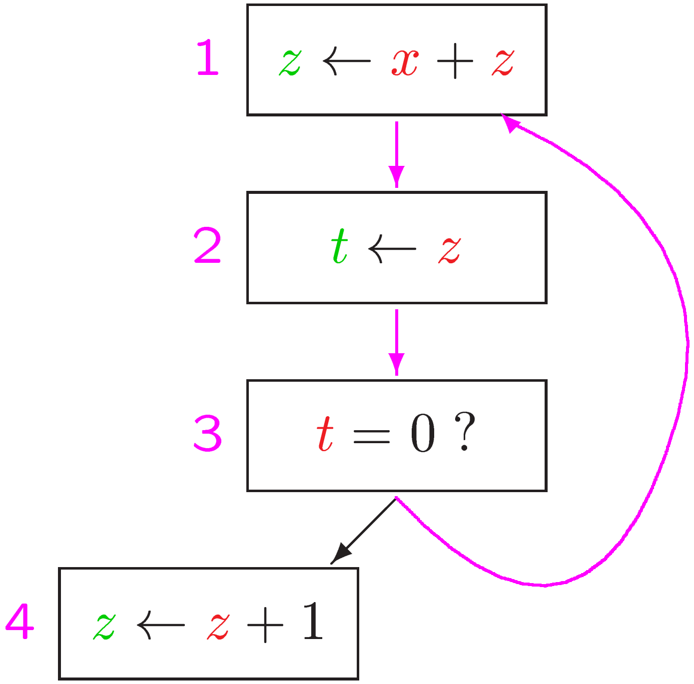
L'exécution du programme évalue une instruction puis passe à l'instruction suivante autorisée (satisfaisant la condition de saut).

Les conditions de branchement sont des conditions dynamiques. Elles peuvent dépendent d'un calcul arbitrairement complexe. Elles ne sont donc pas décidables.

Approximation On ne peut calculer qu'une approximation de la durée de vie. (On la choisira par excès pour être correct.)

On majore grossièrement les conditions de saut en les supposant toujours vraies (elles ne sont plus disjointes).

Un (sous) programme



	Def	Use	Succ	$\frac{\text{live in}}{\text{live out}}$
1	z	x, z	2	$\frac{x, z}{x, z}$
2	t	z	3	$\frac{x, z}{x, z, t}$
3	.	t	4, 1	$\frac{x, z, t}{x, z}$
4	z	z	.	$\frac{z}{.}$

Définition de la durée de vie

Pour chaque instruction i, on définit

- $\mathtt{Use}\,(i)$ l'ensemble des temporaires utilisés par i
- $\mathtt{Def}\,(i)$ l'ensemble des temporaires définis par i
- $\mathtt{Out}\,(i)$ l'ensemble des temporaires vivants à la sortie de i
- $\mathtt{Succ}\,(i)$ l'ensemble des successeurs immédiats de i

Par définition, $t \in \mathtt{Out}\,(i)$ si et seulement si il existe un chemin à partir de i pour lequel t sera lu avant d'être écrit :

$$\exists i_1 \in \mathtt{Succ}\,(i), \ldots i_{n+1} \in \mathtt{Succ}\,(i_n), \bigwedge \left\{ \begin{array}{l} t \in \mathtt{Use}\,(i_{n+1}) \\ \forall k \in [1, n], t \notin \mathtt{Def}\,(i_k) \end{array} \right.$$

Calcul de la durée de vie

On définit

- $\texttt{Succ}^k(i)$ comme l'ensemble des séquences de k-instructions consécutives à i.
- $\overleftarrow{p}$ la séquence p privée de la dernière instruction,
- $\hat{p}$ la dernière instruction de p.

On peut réécrire $\texttt{Out}(i)$ comme

$$\bigcup_{n \in I\!N^*} \bigcup_{p \in \texttt{Succ}^n(i)} \bigwedge \begin{cases} t \in \texttt{Use}(\hat{p}) \\ \forall j \in \overleftarrow{p}, t \notin \texttt{Def}(j) \end{cases}$$

(En fait on peut ajouter le cas $n = 0$ en considérant que $\texttt{Succ}^0(i)$ est vide.)

Temporaires vivant en entrée

$$\mathtt{Out}^{\,n}(i) \triangleq \bigcup_{k=1}^{n} \bigcup_{p \in \mathtt{Succ}^{\,k}(i)} \bigwedge \left\{ \begin{array}{l} t \in \mathtt{Use}\,(\hat{p}) \\ \forall j \in \overleftarrow{p}, t \notin \mathtt{Def}\,(j) \end{array} \right.$$

peut s'écrire

$$\bigcup_{i' \in \mathtt{Succ}\ i} \underbrace{\left(\bigcup_{k=0}^{n-1} \bigcup_{p \in \mathtt{Succ}^{\,k}(i')} \bigwedge \left\{ \begin{array}{l} t \in \mathtt{Use}\,(\hat{p}) \\ \forall j \in \overleftarrow{i'p}, t \notin \mathtt{Def}\,(j) \end{array} \right. \right)}_{\triangleq\ \mathtt{In}^{\,n-1}(i')}$$

Soit

$$\mathtt{In}^{\,n}(i) = \underbrace{\mathtt{Use}\,(i)}_{k\,=\,0} \cup \underbrace{\bigcup_{k=1}^{n} \bigcup_{p \in \mathtt{Succ}^{\,k}(i)} \bigwedge \left\{ \begin{array}{l} t \in \mathtt{Use}\,(\hat{p}) \\ \forall j \in \overleftarrow{p}, t \notin \mathtt{Def}\,(j) \\ t \notin \mathtt{Def}\,(i) \end{array} \right.}_{\mathtt{Out}^{\,n}(i)\ \setminus\ \mathtt{Def}\,(i)}$$

Algorithme

On a $\texttt{Out}\,(i) = \bigcup_{n \in I\!N} \texttt{Out}\,^n(i)$ où

$$\left\{ \begin{array}{l} \texttt{Out}\,^n(i) = \displaystyle\bigcup_{i' \in \texttt{Succ}\,(i)} \texttt{In}\,^{n-1}(i) \\ \texttt{In}\,^n(i) = \texttt{Use}\,(i) \cup (\texttt{Out}\,^n(i) \setminus \texttt{Def}\,(i)) \end{array} \right. \qquad \left\{ \begin{array}{l} \texttt{In}\,^0(i) = \emptyset \\ \texttt{Out}\,^0(i) = \emptyset \end{array} \right.$$

Les suites $\texttt{In}\,^k$ et $\texttt{Out}\,^k$ sont croissantes et bornées (par l'ensemble de tous les temporaires), donc elles convergent. Ainsi, elles sont constantes à partir d'un certain rang.

Algorithme On calcule les fonctions $(\texttt{Out}\,^n, \texttt{In}\,^n)$ pour n croissant tant que $\texttt{Out}\,^n$ est différent de $\texttt{Out}\,^{n-1}$ (i.e. tant qu'il existe i tel que $\texttt{Out}\,^n(i)$ est différent de $\texttt{Out}\,^{n-1}(i)$)

Complexité en $O(n^4)$: au plus n^2 itérations sur $O(n)$ instructions coûtant $O(n)$ par instruction.

Équations au point fixe

Lorsque le point fixe est atteint :

$$\left\{ \begin{array}{l} \mathtt{Out}\,(i) = \bigcup\limits_{i' \in \mathtt{Succ}\ i} \mathtt{In}\,(i') \\ \mathtt{In}\,(i) = \mathtt{Use}\,(i) \cup (\mathtt{Out}\,(i) \setminus \mathtt{Def}\,(i)) \end{array} \right.$$

Résultat : Un temporaire est vivant à l'entrée de i s'il est lu par i ou s'il est vivant en sortie de i et n'est pas écrit par i.

L'algorithme précédent calcule le plus petit point fixe de ces équations.

Il existe aussi un plus grand point fixe, que l'on peut calculer en partant des ensembles pleins. Plus grand et plus petit point fixes ne sont pas égaux (considérer un temporaire jamais utilisé).

Le plus petit point fixe est bien celui que nous cherchions...

Accélération de la convergence

On peut remplacer $\mathtt{In}^{n-1}$ par $\mathtt{In}^{n}$ dans le calcul $\mathtt{Out}^{n}$, lorsque que $\mathtt{In}^{n}$ est déjà connu.

On a donc intérêt à calculer l'instruction $\mathtt{Succ}\ i$ avant l'instruction i.

Un tri topologique permettrait de traiter les composantes connexes les plus profondes en premier.

Comme la plupart des instructions sont simplement des sauts à l'instruction suivante, i.e. $\mathtt{Succ}(i) = \{i+1\}$, on obtient un bon comportement par un parcours du code en sens inverse.

En pratique, quelques itérations suffisent, et on obtient un comportement entre linéaire et quadratique en la taille du code.

On peut représenter les ensembles de temporaires par des listes ordonnées (union et différence en temps linéaire) ou par des vecteurs de bits.

Calcul sur les blocs de base

On peut faire une analyse en deux étapes :

1. On calcule $\mathtt{Def}\,(b)$ et $\mathtt{Use}\,(b)$ pour les blocs de base :

$$\begin{cases} \mathtt{Def}\,(b) = \bigcup_{i \in b} \mathtt{Def}\,(i) \\ \mathtt{Use}\,(b) = \bigcup_{i \in b} \left(\mathtt{Use}\,(i) \setminus \bigcup_{i' \in \mathtt{Pred}^{\,*}(i) \cap b} \mathtt{Def}\,(i') \right) \end{cases}$$

   Puis les variables vivantes $\mathtt{Out}\,(b)$ à la sortie du bloc b comme précédemment.

2. On retrouve alors les $\mathtt{Out}\,(i)$ pour les instructions du bloc b par une simple passe linéaire en sens inverse sur le bloc.

La convergence est aussi rapide (même nombre d'itérations) à condition de faire un parcours arrière dans les deux cas, mais à chaque fois on considère un plus petit nombre de nœuds (donc d'opérations).

Calcul sur l'exemple

	Données			Calcul normal						-accéléré	
i	Def	Use	$Succ$	$\frac{in_0}{out_0}$	$\frac{in_1}{out_1}$	$\frac{in_2}{out_2}$	$\frac{in_3}{out_3}$	$\frac{in_4}{out_4}$	$\frac{in_5}{out_5}$	$\frac{in'_0}{out'_0}$	$\frac{in'_1}{out'_1}$
1	z	x, z	2	$\frac{x, z}{.}$	$\frac{x, z}{z}$	$\frac{x, z}{z}$	$\frac{x, z}{z}$	$\frac{x, z}{z}$	$\frac{x, z}{x, z}$	$\frac{x, z}{z}$	$\frac{x, z}{x, z}$
2	t	z	3	$\frac{z}{.}$	$\frac{z}{t}$	$\frac{z}{t}$	$\frac{z}{x, z, t}$	$\frac{x, z}{x, z, t}$	$\frac{x, z}{x, z, t}$	$\frac{z}{z, t}$	$\frac{x, z}{x, z, t}$
3	.	t	4, 1	$\frac{t}{.}$	$\frac{t}{x, z}$	$\frac{x, z, t}{x, z}$	$\frac{x, z, t}{x, z}$	$\frac{x, z, t}{x, z}$	$\frac{x, z, t}{x, z}$	$\frac{z, t}{z}$	$\frac{x, z, t}{x, z}$
4	z	z	.	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$

$$\texttt{Out}\,(i) = \bigcup_{i' \in \texttt{Succ}\ i} \texttt{In}\,(i') \qquad et \qquad \texttt{In}\,(i) = \texttt{Use}\,(i) \cup (\texttt{Out}\,(i) \setminus \texttt{Def}\,(i))$$

Mise en œuvre

Un nœud est une instruction annotée par les informations `Def`, `Use` et `Succ` ainsi que les champs mutables `In` et `Out`.

```
type flowinfo = {
    instr  :  Ass.instr ;
    def :  temp set;   use :  temp set;
    mutable live_in  :  temp set; mutable live_out :  temp set;
    mutable succ : flowinfo  list  ;
  }
```

On construit le graphe :

```
flowgraph :  code -> flowinfo list  ;;
```

On itère le calcul jusqu'à ce que plus rien n'ai changé :

```
fixpoint  :  flowinfo  list  -> flowinfo  list  ;;
```

(On peut facilement afficher la liste des temporaires vivants dans le code, en commentaire de chaque instruction.)

L'exemple du cours

```
[ Label ("L1", l1);
  Oper ("add  ^d0, ^s0, ^s1", [ x;z], [ z],  None);
  Move ("move ^d0, ^s0", z, t);
  Oper ("beq  ^s0, $zero", [ t], [],    Some [l1; l4]);
  Label ("L4", l4);
  Oper ("add  ^d0, ^s0, 1", [ z], [ z],  None);
  Oper ("jal    $ra", [], [],     Some []); ] ;;
```

Résultat :

```
                             # Def <= Use        # Out
L1:
    add   $z, $x, $z         # z <= z x          # z x
    move $t, $z              # t <= z            # z x t
    beq   $t, $zero, L1      # <= t              # z x
L4:
    add   $z, $z, 1          # z <= z            # .
    jump  Exit
```

```
fact:
    sub   $sp, $sp, fact_f      # sp <= sp            # a0 s0 ra
    move  $112, $ra             # 112 <= ra           # a0 s0 112
    move  $113, $s0             # 113 <= s0           # a0 112 113
    move  $108, $a0             # 108 <= a0           # 108 112 113
    li    $114, 1               # 114 <=              # 108 112 113 114
    bgt   $108, $114, L9        # <= 108 114          # 108 112 113
    li    $115, 1               # 115 <=              # 112 113 115
    move  $107, $115            # 107 <= 115          # 107 112 113
L10:
    move  $v0, $107             # v0 <= 107           # v0 112 113
    move  $s0, $113             # s0 <= 113           # v0 s0 112
    move  $ra, $112             # ra <= 112           # v0 s0 ra
    add   $sp, $sp, fact_f      # sp <= sp            # v0 s0 ra
    j     $ra                   # <= v0 s0 ra         #
L9:
    sub   $116, $108, 1         # 116 <= 108          # 108 112 113 116
    move  $a0, $116             # a0 <= 116           # a0 108 112 113
    jal   fact                  # v0 a0 ra <= a0      # v0 108 112 113
    move  $109, $v0             # 109 <= v0           # 108 109 112 113
    mul   $117, $108, $109      # 117 <= 108 109      # 112 113 117
    move  $107, $117            # 107 <= 117          # 107 112 113
    j     L10                   # <=                  # 107 112 113
```

Importance des annotations `Use` et `Def`

Noter l'effet des `Use` et `Def` dans

- un appel de primitive,
- un appel de fonction/procédure
- au retour d'un programme.

En particulier, en rendant les registres spéciaux vivants à la fin d'une procédure, ils resteront vivants pendant toute la procédure puisqu'ils ne sont jamais écrits.

(Pour simplifier, les registres spéciaux n'ont pas été affichés dans l'exemple ci-dessus.)

Corriger, si nécessaire, ces annotations dans la génération de code

Autres analyses de flux

L'analyse de durée de vie est une analyse arrière : on a intérêt à aller en arrière (dans le sens du calcul) pour accélérer la convergence, parce que l'inforation se propage vers l'arrière.

La plupart des analyses de flux sont à l'inverse des analyses avant.

Analyse d'atteignabilité

On veut calculer l'ensemble des définitions qui sont vivantes à chaque point du programme (instruction ou bloc).

Application à la propagation de constantes

En amont (sur le code source) ou en aval (sur le code machine) :

Si l'affectation d'une constante c à une variable globale (amont) ou à un temporaire t (aval) atteint un point p, alors on sait qu'à ce point on peut remplacer t par la constante c.

Analyse avant

L'information est en général déterminée comme le plus petit point fixe d'une équation de la forme :

$$\begin{cases} \mathtt{In}\,(i) = \bigcup\limits_{i' \in \mathtt{Pred}\ i} \mathtt{Out}\,(i') \\ \mathtt{Out}\,(i) = \mathtt{Gen}\,(i) \cup (\mathtt{In}\,(i) \setminus \mathtt{Kill}\,(i)) \end{cases}$$

Les fonctions `Gen` et `Kill` sont analogues à `Use` et `Def` :

– $\mathtt{Gen}\,(i)$: l'ensemble des défintions générées par l'instruction i

– $\mathtt{Kill}\,(i)$: l'ensemble des définitions détruites par i

Noter aussi l'inversion des ensembles $\mathtt{In}\,(i)$ et $\mathtt{Out}\,(i)$ dans les équations.

Graphe d'interférence

Les nœuds sont les temporaires et les arcs non orientés relient deux temporaires qui ne peuvent pas être superposés.

Cas général : Lorsqu'une instruction écrit dans un temporaire t, il n'est pas possible d'identifier t avec un autre temporaire qui survit après l'instruction.

Formellememt, on ajoute, pour chaque instruction i les arcs $\texttt{Def}\,(i) \times (\texttt{Out}\,(i) \setminus \texttt{Def}\,(i))$.

Cas particulier des instructions Move : Comme à la sortie, la source et la destination ont le même contenu, il n'y a pas jamais interférence entre la source et la destination, même si la source survit à l'instruction. Au contraire, il est possible (et souhaitable) de pouvoir les identifier.

Formellement, on ajoute, pour chaque instruction move les arcs $\texttt{Def}\,(i) \times ((\texttt{Out}\,(i) \setminus \texttt{Def}\,(i)) \setminus \texttt{Use}\,(i))$.

Élimination des instructions Move

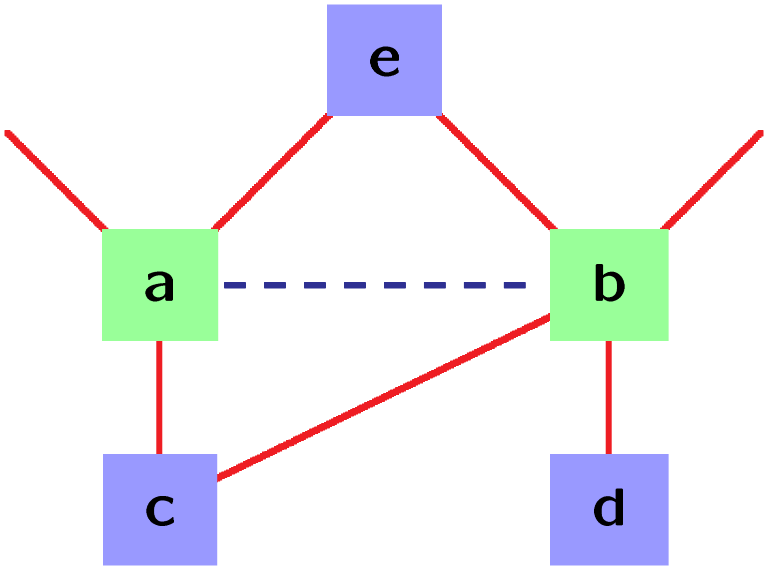
Ces instructions ne calculent pas, et pourront être éliminées, si les temporaires source et destination sont superposés, ce que l'on cherche à faire. Donc, ces deux temporaires n'interfèrent pas ! au contraire ils s'attirent. On les représente dans le graphe d'interférence par des arcs d'attraction.

Formellement, on ajoute pour chaque instruction move d s un arc d'attraction entre s et d.

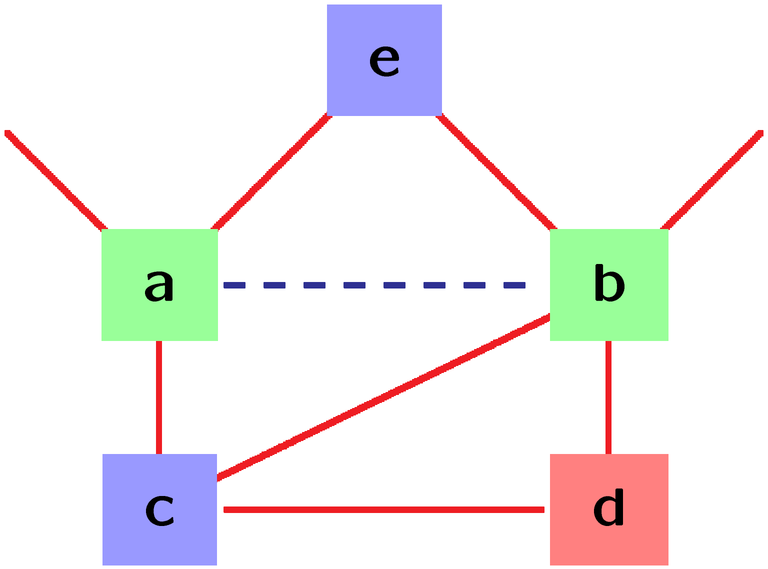
Exemple

Les arcs d'interférence sont en rouge.

Les arcs qui s'attirent (liés par une instruction move) en blue pointillés). Ce seront des candidats privilégiés pour la superposition de nœuds.



2 couleurs



3 couleurs

Mise en œuvre

Un nœud du graphe est un temporaire annoté par la liste des nœuds adjacents.

```
type interference =
  { temp : temp;
    mutable color : temp option;
    mutable adj : interference list ; }
type move =
  { move : Ass.instr ;
    left : interference ; right : interference ; }
```

Le graphe d'interférence est créé en parcourant le code annoté par l'ensemble des registres vivants après chaque instruction.

```
interference :
      temp list (* précoloriés *) ->
      flowinfo list -> interference list * move list
```

Allocation de registres triviale

Le graphe d'interférence permet d'effectuer l'allocation de registres. Nous proposons ici une méthode triviale (une méthode plus performante sera présentée dans le cours suivant).

Les temporaires représentant des registres machines sont pré-coloriés, tous de couleur différente (on pourrait identifier les couleurs avec les registres.) Le but est de colorier les temporaires de telle façon que deux temporaires qui interfèrent aient deux couleurs différentes.

1. On choisit un temporaire t non colorié au hasard.
2. Les couleurs interdites sont l'ensemble des couleurs des temporaires déjà coloriés qui interfèrent avec t.
3. On choisit une couleur non interdite au hasard.
4. En cas d'échec, on choisit de placer t en pile.

Allocation en pile

Lorsqu'un temporaire t doit être alloué en pile, il faut

- lui attribuer un emplacement en pile (position dans le bloc d'activation) à une distance n du sommet de pile.
- pour chaque utilisation de t
  - charger la valeur à la position n du sommet de pile dans un nouveau temporaire t' (ajouter une instruction load)
  - remplacer le temporaire t par t' dans l'instruction.
- pour chaque définition de t
  - remplacer le temporaire t par un nouveau temporaire t' dans l'instruction.
  - sauver la valeur de t' à la position n du sommet pile. (ajouter une instruction store)

Note d'implémentation : pour allouer un emplacement en pile, on utilise module frame (voir frame.mli).

Exemple d'allocation en pile de `s0` et `ra`

```
fact:
    sub  $sp,  $sp, fact_f
    move $117, $s0                 move $141, $s0
                                   sw   $141, 0($sp)
    move $125, $ra                 move $142, $ra
                                   sw   $142, 4($sp)
    move $112, $a0
    li   $126, 1
    bgt  $112, $126, L10
    li   $127, 1
    move $111, $127
L11:
    move $v0, $111
                                   lw   $143, 0($sp)
    move $s0, $117                 move $s0,  $143
                                   lw   $144, 4($sp)
    move $ra, $125                 move $ra,  $144
    add  $sp, $sp, fact_f
    j    $ra
```

Élimination triviale des moves inutiles

Après coup, on élimine les moves inutiles.

En fait, une amélioration triviale consiste à choisir parmi les couleurs possibles d'un temporaire t la couleur d'un temporaire t' attiré par t (directement ou indirectement).

Cela permet d'éliminer de nombreux moves. Une approche plus systématique sera vue dans le chapitre suivant.

Allocation de registres minimale

Au minimum, il s'agit de produire du code qui tourne... Si on sauve systématiquement les registres $s0, s1, ...$ en pile dans chaque procédure, on peut éviter le spilling dans les petits exemples. On peut donc se contenter d'un coloriage le plus simple possible.

Amélioration facile

Si le coloriage tire une couleur au hasard parmi les couleurs possibles, on trouvera peu de move inutiles.

Coloriage biaisé : on choisit, parmi les couleurs possibles, par ordre de préférence :

1. une couleur désirable :

   celle d'un temporaire déjà colorié relié par un move;

2. qui n'est pas déconseillée :

   celle qui est désirable pour un temporaire relié par un move;

Cela permet d'éliminer de nombreux moves, malgré une allocation des couleurs triviale (donc malgré un nombre de placements en pile relativement important).

Exercices

Ce td est très indépendant des précédents

Implémenter l'analyse de durée de vie

On fournit (dans `~remy/compil/td5/`)

– une librairie (smallset.ml : smallset.mli) pour calculer sur les ensembles de registres (représentés par des listes ordonnées).
– un squelette (dépouillé) liveness.ml et son interface liveness.mli.

Pour la mise au point, Afficher la dureé de vie en commentaire des instructions. (On pourra utiliser l'exemple du cours, lui ajouter un appel de fonction ou de primitive. Pour de gros exemples, insérer le module liveness à la fin de la chaîne de compilation.)

Voir `run.ml` et fact.mlx

Construire le graphe d'interférence.