# Analyse des durées de vie Interférences

Luc Maranget

2004

# Place dans la chaîne de compilation

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

**Rappel** le code assembleur est *presque* de l'assembleur : il contient des temporaires.

Le but de l'analyse des durées de vie est de déterminer l'usage des temporaires afin de les remplacer par des registres machine.

*Ce type d'analyse se retrouve sous d'autres formes (analyses de flot) pour faire des optimisations poussées (ou agressives)*

## 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** en en point d'un programme si son contenu *en cet endroit* est utilisé dans une instruction suivante (au sens large).

Ou encore, si le temporaire est lu par la suite, avant d'être écrit.

La **durée de vie** d'un temporaire est l'ensemble des points du programme où il est vivant.

**Application** Deux temporaires qui ont des durées de vie disjointes peuvent être superposés, c'est à dire représenter le même registre.

## Exemple

$$\begin{array}{l}
C_1 \\ \hline
t_1 \leftarrow 1 \\
\quad \{t_1\} \\
t_2 \leftarrow 2 \\
\quad \{t_1, t_2\} \\
t_2 \leftarrow t_1 + t_2 \\ \hline
\end{array}
\quad live1live2 \quad
\begin{array}{l}
C_2 \\ \hline
t_1 \leftarrow 1 \\
\quad \{t_1\} \\
t_2 \leftarrow t_1 + 2 \\
\quad \{t_2\} \\
t_1 \leftarrow 3 \\
\quad \{t_1, t_2\} \\
t_2 \leftarrow t_2 + t_1 \\ \hline
\end{array}$$

live3live4live5

**Remarquer** $t_1$ n'est pas vivant en sortie de la deuxième instruction de $C_2$.

# Définitions

Pour chaque instruction $i$, on définit :

- `Use` $(i)$ temporaires lus (ou encore utilisés) par $i$,
- `Def` $(i)$ temporaires écrits (ou encore définis) par $i$,
- `Succ` $(i)$ instructions qui peuvent suivre immédiatement $i$ dans une exécution.

`Use` $(i)$, `Def` $(i)$ et `Succ` $(i)$ sont mis en valeur par notre représentation des instructions de l'assembleur :

```
type instr =
  | Oper of string * temp list * temp list * label list option
(* Oper (mnémonique, sources, destinations, sauts) *)
  ...
```

# Définitions (suite)

Les informations de *liveness* :

- $\texttt{In}\,(i)$ temporaires vivants à l'entrée de $i$,
- $\texttt{Out}\,(i)$ temporaires vivants à la sortie de $i$.

Selon la définitions informelle « *temporaires lus avant d'être écrits* ». Les $\texttt{Out}\,(i)$ :

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

Et de même, les $\texttt{In}\,(i)$ :

$$\left\{ t \,\middle|\, \exists i_1 = i, i_2 \in \texttt{Succ}\,(i_1), \ldots, i_n \in \texttt{Succ}\,(i_{n-1}), \bigwedge \left\{ \begin{array}{l} t \in \texttt{Use}\,(i_n) \\ \forall k \in [1, n-1], t \notin \texttt{Def}\,(i_k) \end{array} \right. \right\}$$

(Même chose décalé d'un cran vers le haut.)

**Note** Il s'agit d'une approximation, pourquoi ?

# Équations corollaires

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

Ces deux égalités permettent de calculer `Out` et `In` par itération. Pour changer, nous allons le prouver.

Pour le moment, dans le cas d'un code en séquence, ces équations permettent de calculer les durées de vie *en remontant le sens de l'exécution*. Si $\texttt{Succ}\,(i_1) = \{i_2\}$, alors

$$\texttt{Out}\,(i_1) = \texttt{In}\,(i_2) = (\texttt{Out}\,(i_2) \setminus \texttt{Def}\,(i_2)) \cup \texttt{Use}\,(i_2)$$

# Exemple, code en séquence

Le calcul procède à l'inverse de l'ordre d'exécution.

| $C_1$ |
| --- |
| $\quad T \setminus \{t_1, t_2\}$ |
| $t_1 \leftarrow 1$ |
| $\quad (T \setminus \{t_2\}) \cup \{t_1\}$ |
| $t_2 \leftarrow 2$ |
| $\quad (T \setminus \{t_2\} \cup \{t_1, t_2\}$ |
| $t_2 \leftarrow t_1 + t_2$ |
| $\quad T$ |

| $C_2$ |
| --- |
| $\quad T \setminus \{t_1, t_2\}$ |
| $t_1 \leftarrow 1$ |
| $\quad (T \setminus \{t_1, t_2\}) \cup \{t_1\}$ |
| $t_2 \leftarrow t_1 + 2$ |
| $\quad (T \setminus \{t_1, t_2\}) \cup \{t_2\}$ |
| $t_1 \leftarrow 3$ |
| $\quad (T \setminus \{t_2\}) \cup \{t_1, t_2\}$ |
| $t_2 \leftarrow t_2 + t_1$ |
| $\quad T$ |

# Avec des sauts

On utilise donc le graphe de flot. Il met en valeur :

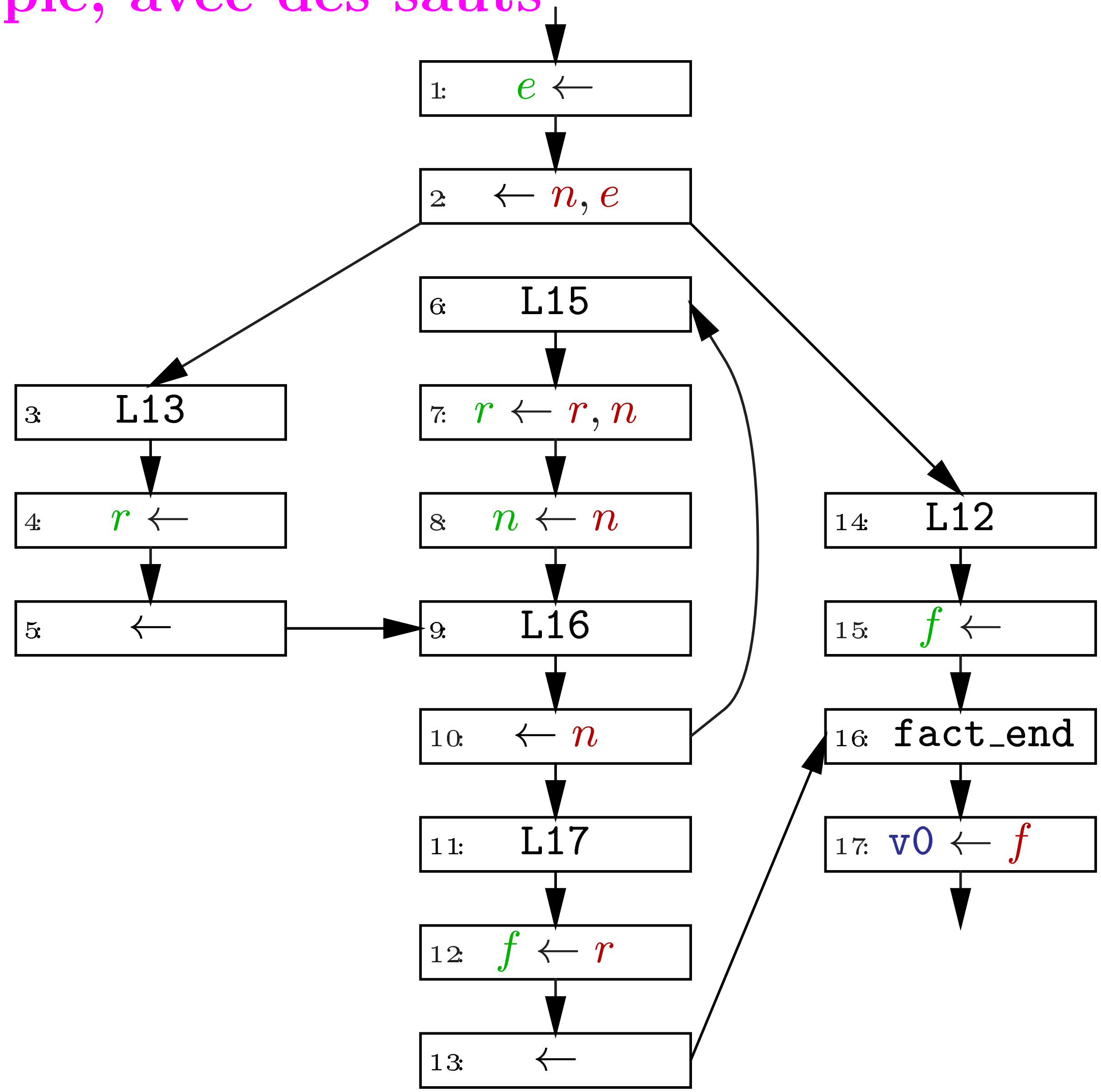
- Les `Succ` (les arcs du graphe),
- Les `Def` et les `Use`, peu importe ce que font exactement les instructions.

## Dans l'exemple

- Durée de vie de $e$ (entre 1 et 2).
- $n$ vivant de l'entrée de 1 à la sortie de 10.
- $f$ vivant en entrée de 16 pour deux raisons.

# Exemple, avec des sauts

```
      li    e, 1
      ble n, e, L12
L13:
      li    r, 1
      b L16
L15:
      mul r, r, n
      sub  n, n, 1
L16:
      bgt n, $zero, L15
L17:
      move f, r
      b fact_end
L12:
      li    f, 1
fact_end:
      move $v0, f
```



# Calcul des durées de vie

On définit précisément les séquences de $k$ instructions qui peuvent suivre une instruction $i$.

$$\texttt{Succ}^{0}(i) = \{[\,]\}$$

$$\texttt{Succ}^{k+1}(i) = \bigcup_{i_1 \in \texttt{Succ}(i)} \left( \bigcup_{c_k \in \texttt{Succ}^{k}(i_1)} \{\,[i_1; c_k]\,\} \right)$$

On restreint les définitions de $\texttt{Out}$ et $\texttt{In}$ en $\texttt{Out}^{n}$ et $\texttt{In}^{n}$

$$\texttt{Out}^{n}(i) =$$

$$\bigcup_{k=0}^{n} \left( \bigcup_{[c_{k-1}; i_k] \in \texttt{Succ}^{k}(i)} \left\{ t \,\middle|\, \bigwedge \left\{ \begin{array}{l} t \in \texttt{Use}(i_k) \\ \forall k' \in [1, k-1], t \notin \texttt{Def}(i_{k'}) \end{array} \right. \right\} \right)$$

# Calcul des durées de vie, suite

$$\texttt{In}^n(i) =$$

$$\bigcup_{k=0}^{n}\left(\bigcup_{c_{k-1}\in\texttt{Succ}^{k-1}(i)}\left\{t \,\middle|\, \bigwedge\left\{\begin{array}{l} t \in \texttt{Use}(j_k) \\ \forall k' \in [1, k-1], t \notin \texttt{Def}(j_{k'}) \end{array}\right.\right\}\right)$$

(avec $[j_1; \ldots; j_k] = [i; c_{k-1}]$).

Ces définitions entraînent :

- $\texttt{Out}^n(i)$ et les $\texttt{In}^n(i)$ sont des suites d'ensembles croissantes au sens large.
- Leurs limites sont $\texttt{Out}$ et $\texttt{In}$.

$$\texttt{Out}(i) = \bigcup_{n\in\mathbb{N}} \texttt{Out}^n(i) \qquad \texttt{In}(i) = \bigcup_{n\in\mathbb{N}} \texttt{In}^n(i)$$

# Calcul des durées de vie, méthode

Les suites $\texttt{Out}^n$ et $\texttt{In}^n$ ont une formulation par récurrence :

$$\texttt{Out}^0(i) = \emptyset \qquad\qquad \texttt{In}^0(i) = \emptyset$$

$$\texttt{In}^{n+1}(i) = \texttt{Use}(i) \cup (\texttt{Out}^n(i) \setminus \texttt{Def}(i))$$

$$\texttt{Out}^n(i) = \bigcup_{j \in \texttt{Succ}(i)} \texttt{In}^n(j)$$

Or ces suites sont croissantes et bornées (nombre fini de temporaires) : elles sont donc stationnaires à partir d'un certain rang, leurs limites sont $\texttt{Out}$ et $\texttt{In}$. Et on a évidemment :

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

## Durée de vie, définition par point fixe

On définit souvent $\texttt{Out}$ et $\texttt{In}$ plus directement.

La suite $\texttt{Out}$ est le *plus petit* point fixe de la fonction *monotone* :

$$O(i) \longrightarrow \bigcup_{j \in \texttt{Succ}(i)} F_j(O(j))$$

Avec $F_i(X) = \texttt{Use}(i) \cup (X \setminus \texttt{Def}(i))$.

Ou plus synthétiquement, $\texttt{Out}$ est le plus petit point fixe de la fonction $F_{\texttt{Out}}$ monotone.

$$F_{\texttt{Out}}(X) = \{i \mapsto \bigcup_{j \in \texttt{Succ}(i)} F_j(X(j))\}$$

Le $\texttt{Out}$ du transparent précédent (limite de la suite $\texttt{Out}^k$) est bien le plus petit des points fixes : $\{i \mapsto \emptyset\} \subseteq T$ et :

$$\texttt{Out}^{k+1} = F_{\texttt{Out}}(\texttt{Out}^k) \subseteq F_{\texttt{Out}}(T) = T$$

## Calcul des $\texttt{Out}^n$

À priori :

for i=1 to n do $\texttt{Out}\,(i)$ <- $\emptyset$ done ; $O(n)$
do
  for i=1 to n do $\texttt{Out}'\,(i)$ <- $\texttt{Out}\,(i)$ done ; $O(n)$
  for i=1 to n do
    $\texttt{Out}\,(i)$ <- $\left(\bigcup_{j\in\texttt{Succ}\,(i)}(\texttt{Out}'\,(j)\setminus\texttt{Def}\,(j))\cup\texttt{Use}\,(j)\right)$ $|\texttt{Succ}\,(i)|\times O(n)$
  done $\times n \rightarrow O(n^2)$
until $\forall\, i\in[1,n], \texttt{Out}\,(i)=\texttt{Out}\,'(i)$ $\times O(n^2)\rightarrow O(n^4)$

Coût ? Au plus $n^2$ itération (croissance) cout1cout2cout3cout4

### En pratique, le coût n'est pas si elevé

- Ensembles creux, (ou codage par champs de bits).
- Bon ordonancement des instructions $\rightarrow$ 2 ou 3 itérations en pratique.

## Une borne plus stricte

Reprenons la définition des $\texttt{Out}^k(i)$ :

$$\bigcup_{\ell=0}^{k}\left(\bigcup_{[c_{\ell-1};i_\ell]\in\texttt{Succ}^\ell(i)}\left\{t \,\middle|\, \bigwedge\left\{\begin{array}{l} t\in\texttt{Use}(i_\ell)\\ \forall k'\in[1,\ell-1], t\notin\texttt{Def}(i_{k'})\end{array}\right.\right\}\right)$$

On considère un code de $n$ instructions au total et on suppose $k\geq\ell>n$. La séquence $[i;i_1;\ldots;i_\ell]$ comprend alors des sous-sequences (strictes) de la forme $[i';\ldots;i']$. En les remplaçant par la séquence $[i']$, on obtient une nouvelle séquence de taille $\ell'+1\leq n+1$ dont toutes les instructions sont deux à deux distinctes, sauf éventuellement la première et la dernière. Cette séquence prouve $t\in\texttt{Out}^{\ell'}(i)$.

Autrement dit, on a prouvé : $k\geq n\Longrightarrow\texttt{Out}(k)\subseteq\texttt{Out}^n(i)$. Soit encore $\texttt{Out}^n(i)=\texttt{Out}^{n+1}(i)$. C'est à dire qu'il y a au plus $n+1$ itérations de la boucle externe. Le coût est donc en $O(n^3)$.

# Une envie logique

Calculer les $\texttt{Out}^{k}$ en place dans un tableau.

$$
\begin{array}{l}
\textbf{for}\ \mathrm{i}{=}1\ \textbf{to}\ \mathrm{n}\ \textbf{do}\ \texttt{Speed}\,(n)\ \texttt{<-}\ \emptyset\ \textbf{done}\ ;\\
\textbf{let}\ \mathrm{encore} = \mathrm{ref\ true}\ \textbf{in}\\
\textbf{while}\ !\mathrm{encore}\ \textbf{do}\\
\quad \mathrm{encore} := \mathrm{false}\ \ ;\\
\quad \textbf{for}\ \mathrm{i}{=}1\ \textbf{to}\ \mathrm{n}\ \textbf{do}\\
\quad\quad \textbf{let}\ \mathrm{prev} = \texttt{Speed}\,(i)\ \textbf{in}\\
\quad\quad \texttt{Speed}\,(i)\ \texttt{<-}\ \left(\bigcup_{j \in \texttt{Succ}\,(i)} (\texttt{Speed}\,(j) \setminus \texttt{Def}\,(j)) \cup \texttt{Use}\,(j)\right)\ ;\\
\quad\quad \mathrm{encore} := !\mathrm{encore}\ \ ||\ \ (\,\mathrm{prev}\ <>\ \texttt{Speed}\,(i))\\
\quad \textbf{done}\\
\textbf{done}
\end{array}
$$

# Justification de l'envie logique

**Intuition** On remplace certains $\texttt{Out}^n(j)$ par $\texttt{Out}^{n+1}(j)$ dans la formule

$$\texttt{Out}^{n+1}(i) = \Big(\bigcup_{j \in \texttt{Succ}(i)} \texttt{Use}(j) \cup (\texttt{Out}^n(j) \setminus \texttt{Def}(j))\Big)$$

**Plus précisément**

$$\texttt{Speed}^0(i) = \emptyset \qquad \texttt{Speed}^{k+1}(i) =$$

$$\left(\bigcup_{j \in \texttt{Succ}(i),\, j<i} F_j(\texttt{Speed}^{k+1}(j))\right) \bigcup \left(\bigcup_{j \in \texttt{Succ}(i),\, j>i} F_j(\texttt{Speed}^{k}(j))\right)$$

De façon synthétique on a :

$$\texttt{Speed}^0 = \{i \mapsto \emptyset\}, \qquad \texttt{Speed}^{k+1} = F_{\texttt{Speed}}(\texttt{Speed}^k)$$

## Suite de la justification de l'envie logique

Definissons $\texttt{Speed}$ comme la limite de $\texttt{Speed}^k$ ($F_{\texttt{Speed}}$ monotone).

Par monotonie des $F_j$ on a :

$$\texttt{Out}^k \subseteq \texttt{Speed}^k$$

C'est à dire que le calcul des $\texttt{Speed}^k$ est « *plus efficace* ».

On a donc aussi :

$$\texttt{Out} \subseteq \texttt{Speed}$$

Si on a peur du « passage à la limite », il suffit de considérer un $k_0$ pour lequel les limites sont atteintes.

Par ailleurs $F_{\texttt{Speed}}(\texttt{Out}) = \texttt{Out}$ et $\{i \mapsto \emptyset\} \subseteq \texttt{Out}$, on a donc :

$$\texttt{Speed}^k \subseteq \texttt{Out}$$

# Calcul en pratique, cas du code en séquence

Il faut ordonner les instructions à l'envers de l'ordre d'exécution :

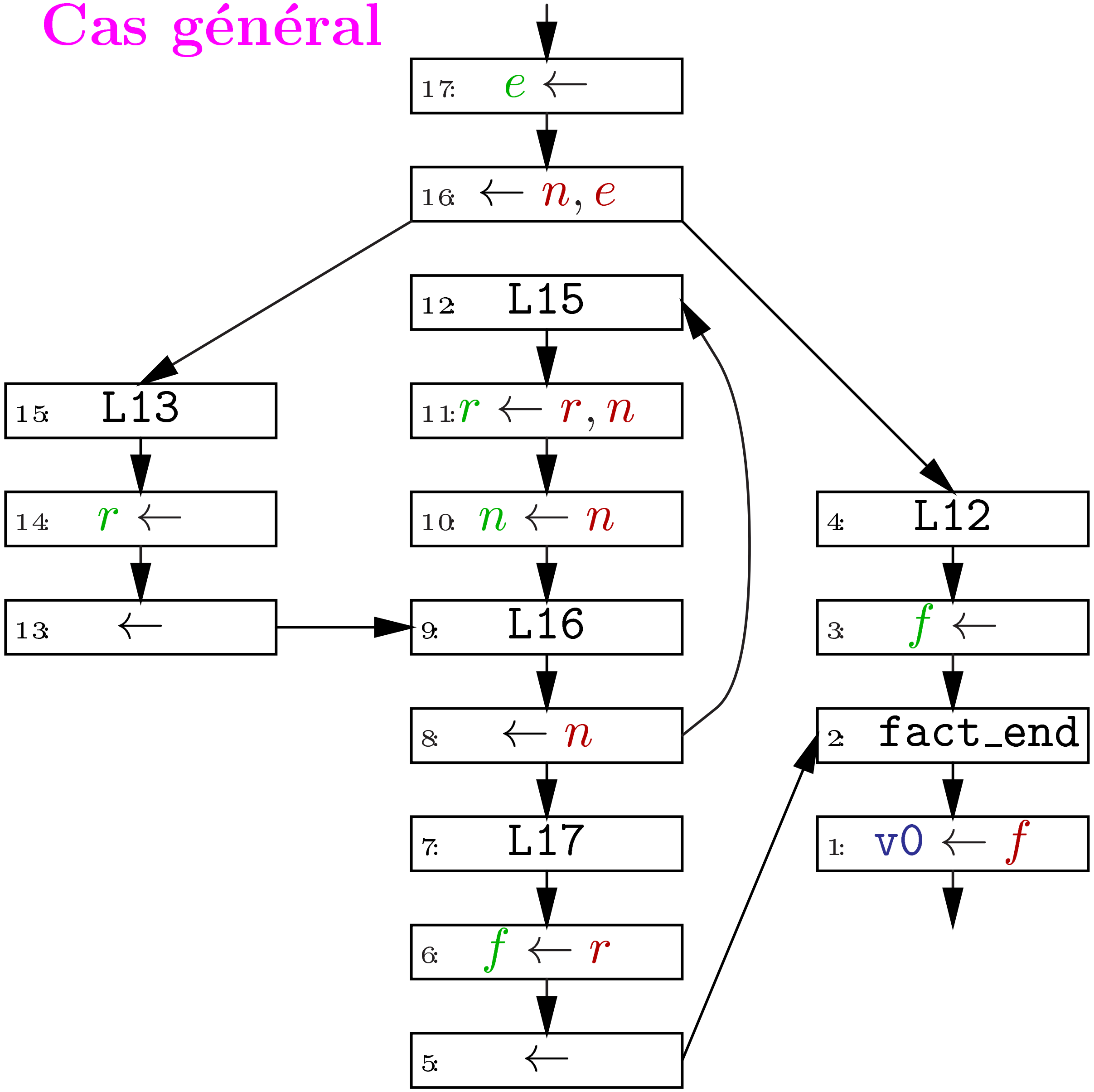
| $C$ | $\texttt{Out}^0$ | $\texttt{Out}^1$ | $\texttt{Out}^2$ | $\texttt{Out}^4$ | $\texttt{Out}^5$ |
|---|---|---|---|---|---|
| $t_1 \leftarrow 1$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\{t_1\}$ | $\{t_1\}$ |
| $t_2 \leftarrow 2$ | $\emptyset$ | $\emptyset$ | $\{t_1, t_2\}$ | $\{t_1, t_2\}$ | $\{t_1, t_2\}$ |
| $t_2 \leftarrow t_1 + t_2$ | $\emptyset$ | $\{t_1, t_2\}$ | $\{t_1, t_2\}$ | $\{t_1, t_2\}$ | $\{t_1, t_2\}$ |
| $t_3 \leftarrow t_2 * t_1$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

| $i$ | $C$ | $\texttt{Speed}^0$ | $\texttt{Speed}^1$ | $\texttt{Speed}^2$ |
|---|---|---|---|---|
| $4:$ | $t_1 \leftarrow 1$ | $\emptyset$ | $\{t_1\}$ | $\{t_1\}$ |
| $3:$ | $t_2 \leftarrow 2$ | $\emptyset$ | $\{t_1, t_2\}$ | $\{t_1, t_2\}$ |
| $2:$ | $t_2 \leftarrow t_1 + t_2$ | $\emptyset$ | $\{t_1, t_2\}$ | $\{t_1, t_2\}$ |
| $1:$ | $t_3 \leftarrow t_2 * t_1$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

## Cas général

Éviter le plus possible $j \in$ `Succ` $i$ et $j \geq i$.

Ici on a `Succ` $(8) = \{7, 12\}$.



# Calcul

| $i$ | $\texttt{Speed}^0$ | $\texttt{Speed}^1$ | $\texttt{Speed}^2$ | $\texttt{In}$ |
|---|---|---|---|---|
| 1 | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\{f\}$ |
| 2 | $\emptyset$ | $\{f\}$ | $\{f\}$ | $\{f\}$ |
| 3 | $\emptyset$ | $\{f\}$ | $\{f\}$ | $\emptyset$ |
| 4 | $\emptyset$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| 5 | $\emptyset$ | $\{f\}$ | $\{f\}$ | $\{f\}$ |
| 6 | $\emptyset$ | $\{f\}$ | $\{f\}$ | $\{r\}$ |
| 7 | $\emptyset$ | $\{r\}$ | $\{r\}$ | $\{r\}$ |
| 8 | $\emptyset$ | $\{r\}$ | $\{n, r\}$ | $\{n, r\}$ |

| $i$ | $\texttt{Speed}^0$ | $\texttt{Speed}^1$ | $\texttt{Speed}^2$ | $\texttt{In}$ |
|---|---|---|---|---|
| 9 | $\emptyset$ | $\{n, r\}$ | $\{n, r\}$ | $\{n, r\}$ |
| 10 | $\emptyset$ | $\{n, r\}$ | $\{n, r\}$ | $\{n, r\}$ |
| 11 | $\emptyset$ | $\{n, r\}$ | $\{n, r\}$ | $\{n, r\}$ |
| 12 | $\emptyset$ | $\{n, r\}$ | $\{n, r\}$ | $\{n, r\}$ |
| 13 | $\emptyset$ | $\{n, r\}$ | $\{n, r\}$ | $\{n, r\}$ |
| 14 | $\emptyset$ | $\{n, r\}$ | $\{n, r\}$ | $\{n\}$ |
| 15 | $\emptyset$ | $\{n\}$ | $\{n\}$ | $\{n\}$ |
| 16 | $\emptyset$ | $\{n\}$ | $\{n\}$ | $\{e, n\}$ |
| 17 | $\emptyset$ | $\{e, n\}$ | $\{e, n\}$ | $\{n\}$ |

$\texttt{Speed}^1\,(8) = \{r\} \cup \emptyset$ mais $\texttt{Speed}^2\,(8) = \{r\} \cup \{r, n\}$ (car $\texttt{Speed}^1\ 12 = \{r, n\}$).

# Pour aller encore plus vite

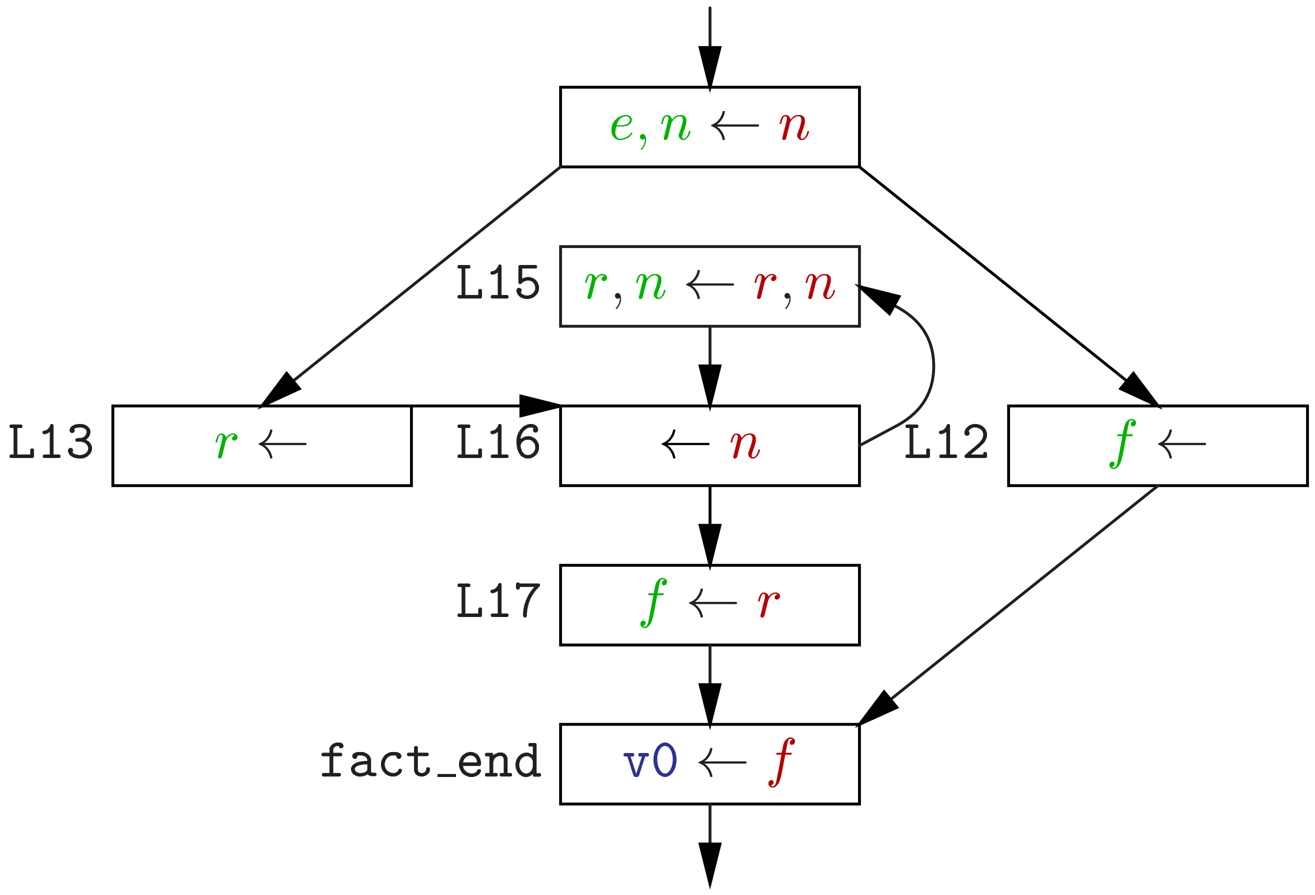
On passe au graphe des blocs de base. En effet il est inutile d'itérer à l'intérieur des blocs de base. Ils se comportent comme de grosses instructions :

$$\texttt{Def}\,(b) = \bigcup_{k=1}^{n} \texttt{Def}\,(i_k),$$

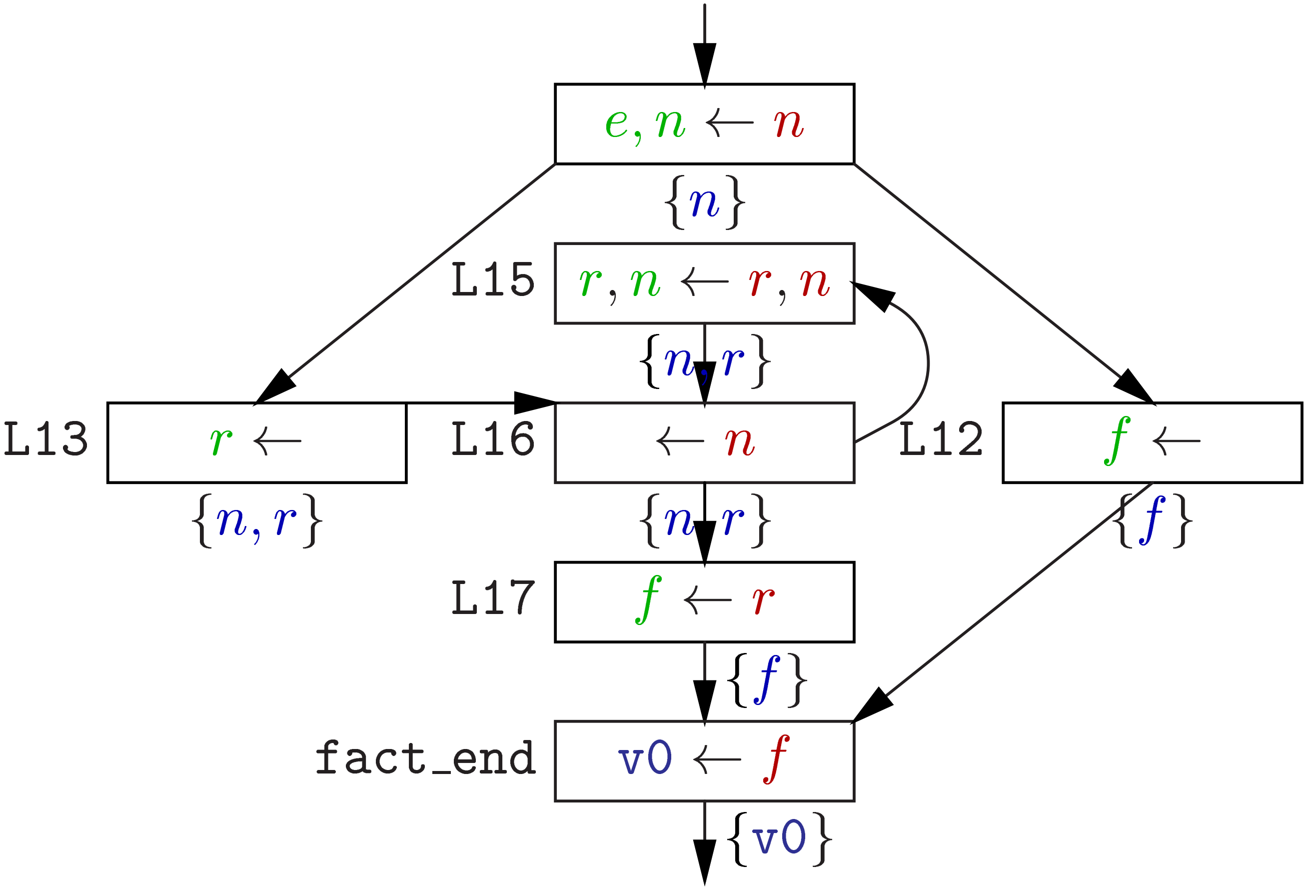
$$\texttt{Use}\,(b) = \left( \bigcup_{k=1}^{n} \texttt{Use}\,(i_k) \setminus \left( \bigcup_{j=1}^{k-1} \texttt{Def}\,(i_j) \right) \right)$$

Le graphe de flot des blocs de base a moins de nœuds que celui des instructions. On divise le nombre de calculs à itérer par la longueur moyenne des blocs.

## Graphe de flot des blocs de base



# Graphe de flot des blocs de base




Ensuite il faut calculer les `Out` , instruction par instruction dans les blocs et à partir des `Out` en fin de bloc.

# Interférence

Par *définition* deux temporaires interfèrent si ils ne peuvent pas partager le même registre. La relation d'interférence est non-réflexive, symétrique et non-nécessairement transitive.

Dans le cas du MIPS, il suffit de parcourir les instructions (*i.e.* le graphe de flot).

- Si $i$ est une instruction qui n'est pas un transfert entre temporaires, alors les temporaires de `Def` $(i)$ et ceux de `Out` $(i) \setminus$ `Def` $(i)$ interfèrent.
- Si $i$ est un transfert du temporaire $s$ vers le temporaire $d$, alors $d$ et les temporaires de `Out` $(i) \setminus \{d, s\}$ interfèrent.

La seconde règle est une finesse, quel est son intérêt ?

## Graphe d'interférence

La relation d'interférence est représentée par un graphe (non-orienté)

- Les sommets sont les temporaires (et les registres).
- Les arcs d'interférence expriment la relation d'interférence.
- Les arcs `move` désignent les temporaires liés par un transfert.

Ce graphe se calcule facilement comme décrit au transparent précédent.

# Exemple du code en séquence

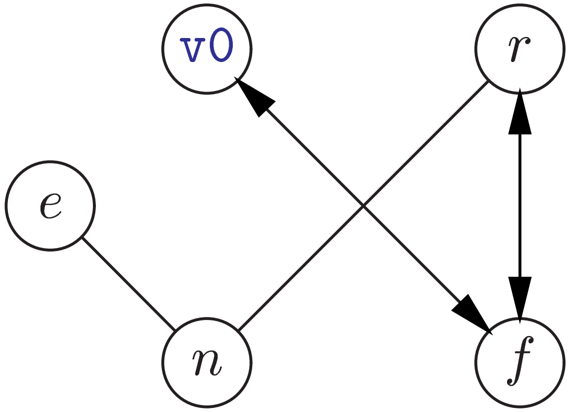
| $i$ | $C$ | `Def` | `Out` |
|---|---|---|---|
| 4 | $t_1 \leftarrow 1$ | $t_1$ | $t_1$ |
| 3 | $t_2 \leftarrow 2$ | $t_2$ | $t_1, t_2$ |
| 2 | $t_2 \leftarrow t_1 + t_2$ | $t_2$ | $t_1, t_2$ |
| 1 | $t_3 \leftarrow t_2 * t_1$ | $t_3$ | |

$t_1$ et $t_2$ interfèrent, en raison par exemple de `Def` $(2) = \{t_2\}$ et `Out` $(2) = \{t_1, t_2\}$.

# Exemple du code arbitraire

| $i$ | `move` | `Def` | `Out` |
|---|---|---|---|
| 1 | `v0` $\leftarrow f$ | `v0` | `v0` |
| 2 | | | $f$ |
| 3 | $f \leftarrow 1$ | $f$ | $f$ |
| 4 | | | |
| 5 | | | $f$ |
| 6 | $f \leftarrow r$ | $f$ | $f$ |
| 7 | | | $r$ |
| 8 | | | $n, r$ |

| $i$ | `move` | `Def` | `Out` |
|---|---|---|---|
| 9 | | | $n, r$ |
| 10 | | $n$ | $n, r$ |
| 11 | | $r$ | $n, r$ |
| 12 | | | $n, r$ |
| 13 | | | $n, r$ |
| 14 | $r \leftarrow 1$ | $r$ | $n, r$ |
| 15 | | | $n$ |
| 16 | | | $n$ |
| 17 | $e \leftarrow 1$ | $e$ | $n$ |




Allocation : $e$, $r$ et $f$ dans $r_1$, $n$ dans $r_2$. Les arcs `move` indiquent de choisir $r_1 =$ `v0`.

## Exemple dans le style des TP

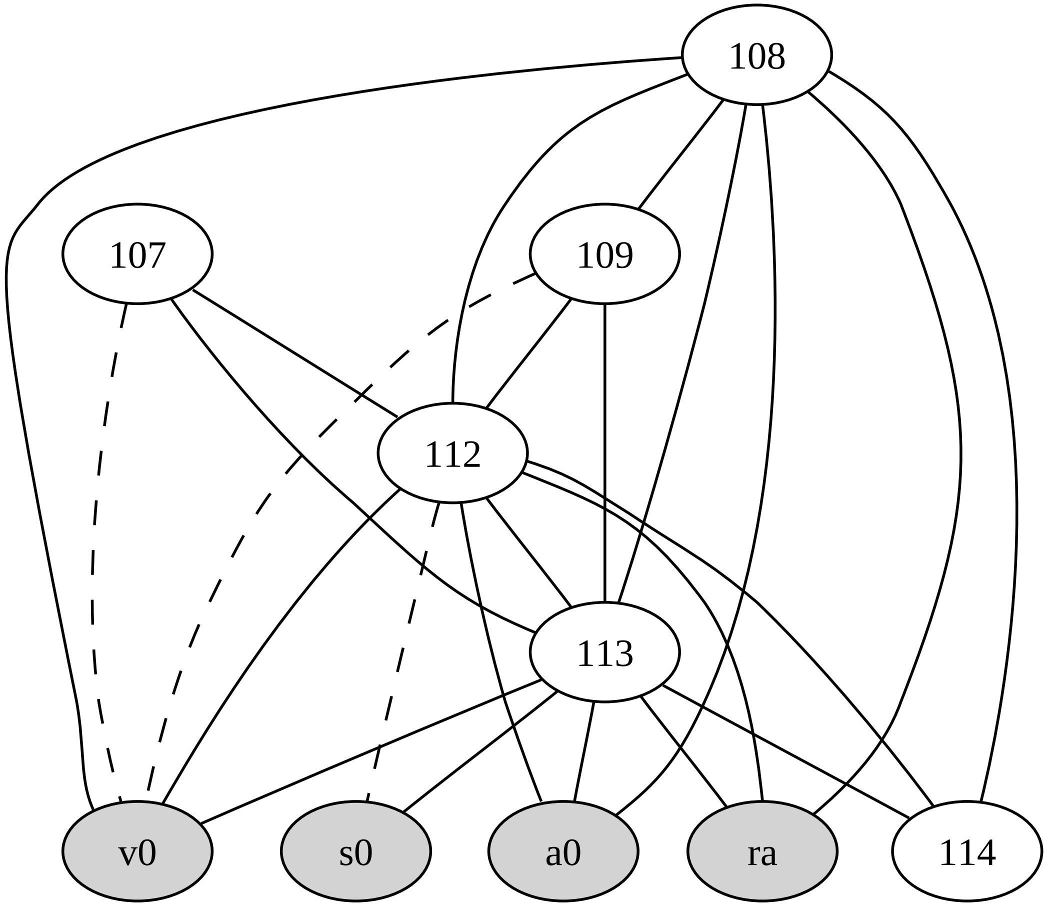
Source :

```
function fact ( n : integer ) : integer ;
begin
  if n <= 1 then
    fact := 1
  else
    fact := n * fact (n-1)
end;
```

# Liveness

```
fact:                          #      ⇐                    # $a0 $s0 $ra
    subu $sp, $sp, fact_f      #      ⇐                    # $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
    ble  $108, $114, L12       #      ⇐ $108 $114          # $108 $112 $113
L13:                           #      ⇐                    # $108 $112 $113
    sub  $a0, $108, 1          # $a0 ⇐ $108                # $a0 $108 $112 $113
    jal  fact                  # $v0 $a0 $ra ⇐ $a0         # $v0 $108 $112 $113
    move $109, $v0             # $109 ⇐ $v0                # $108 $109 $112 $113
    mul  $107, $108, $109      # $107 ⇐ $108 $109          # $107 $112 $113
    b    fact_end              #      ⇐                    # $107 $112 $113
L12:                           #      ⇐                    # $112 $113
    li   $107, 1               # $107 ⇐                    # $107 $112 $113
fact_end:                      #      ⇐                    # $107 $112 $113
    move $v0, $107             # $v0 ⇐ $107                # $v0 $112 $113
    move $ra, $112             # $ra ⇐ $112                # $v0 $113 $115
    move $s0, $113             # $s0 ⇐ $113                # $v0 $s0 $115
    addu $sp, $sp, fact_f      #      ⇐                    # $v0 $s0 $115
    j    $ra                   #      ⇐ $v0 $s0 $ra        #
```

**Interférence**



# Realisation des graphes orientés

```
(* Types de sommets et des graphes *)
type 'a node
and 'a t

val create : 'a -> 'a t
(* Créer un nouveau graphe, initialement vide *)

val new_node : 'a t -> 'a -> 'a node
(* ajoute un noeud  par effet de bord *)

val new_edge : 'a t -> 'a node -> 'a node -> unit
(* ajoute un arc de n1 vers n2, (si il  n'existe  pas déjà) *)
```

```
val nodes : 'a t -> 'a node list
(* Tous les noeuds, dans l'ordre de leur création *)

val info : 'a t -> 'a node -> 'a
(* Le contenu d'un noeud *)

val succ : 'a t -> 'a node -> 'a node list
(* Les successeurs d'un noeud *)

val iter : 'a t -> ('a node -> unit) -> unit
(* itère un fonction sur les noeuds (ordre de création) *)

val debug : out_channel -> (out_channel -> 'a node -> unit) -> 'a t -> unit
(* Affichage pour le debug *)
```

## Les graphes de flot

```
type flowinfo = {
    instr  :  Ass.instr   ;                  (* instruction *)
    def  :  temp set;  use  :  temp set;  (* détruits et lus *)
    mutable live_in : temp set;          (* sans commentaire *)
    mutable live_out : temp set;
  }

type flowgraph = flowinfo Graph.t
(* Type des graphes de flots décorés des live−in/live−out *)

val flow  :  Ass.instr  list  -> flowgraph
(* Fabrication du graphe de flot, décoré par les durées de vie *)
```

## Structure du calcul des graphe de flot

En deux temps : création du graphe, puis calcul des durées de vie.

```
let flow code =
  let g = mk_graph code in
  fixpoint g ;
  g
```

# Les graphes non-orientés

```
type ('a, 'b) t
type 'a node
val create : 'a -> ('a, 'b) t
val new_node : ('a, 'b) t -> 'a -> 'a node

val new_edge : ('a, 'b) t -> 'a node -> 'a node -> 'b ->  unit
(*
   « new_edge g n1 n2 l » , ajoute un arc étiqueté par l entre n1 et n2.
   L'arc n'est pas crée lorsque :
   — Il existe déjà,
   — ou n1=n2
*)
```

# Les graphes d'interférence

```
(* Sortes d'arcs du graphe d'interférence *)
type ilab = Inter | Move_related

(* Contenu des noeud du graphe d'interférence *)
type interference = {
    temp : temp  ;                    (* un temporaire *)
(* Les champs suivants sont utiles pour l'allocation de registres *)
    mutable color : temp option ;
    ...
  }

type igraph = (interference, ilab) Sgraph.t

val interference :  flowgraph  -> igraph
```