

## Chapitre 8

# Allocation des registres par coloriage de graphe

### Sommaire

<b>8.1</b>	<b>Analyse de la durée de vie des variables</b>	<b>1</b>
8.1.1	Introduction	1
8.1.2	Temporaires <i>définis</i> , temporaires <i>lus</i>	2
8.1.3	Graphe de flot et durée de vie des variables	3
8.1.4	Calcul de la durée de vie des temporaires	4
8.1.5	Complexité de l'analyse	5
8.1.6	Mise en œuvre	6
<b>8.2</b>	<b>Allocation des registres par coloriage de graphe</b>	<b>6</b>
8.2.1	Introduction	6
8.2.2	Interférence	7
8.2.3	Coloriage de graphe	9
8.2.4	Élimination des instructions MOVE	10
8.2.5	Mise en œuvre	12

## 8.1 Analyse de la durée de vie des variables

### 8.1.1 Introduction

#### Allocation des registres par coloriage de graphe

Nous avons vu comment, de façon extrêmement naïve, il est possible de produire du code assembleur exécutable à partir du code assembleur abstrait, en plaçant chaque temporaire en pile, et en réservant trois registres pour charger et décharger les temporaires à chaque instruction. Cela produit un trafic mémoire élevé et inutile, allonge le programme, et diminue énormément les performances. Il existe des approches plus sophistiquées qui permettent de réduire significativement l'allocation en pile :

- allocation de registres par coloriage de graphes [CAC<sup>+</sup>81]
- allocation linéaire [PS99]

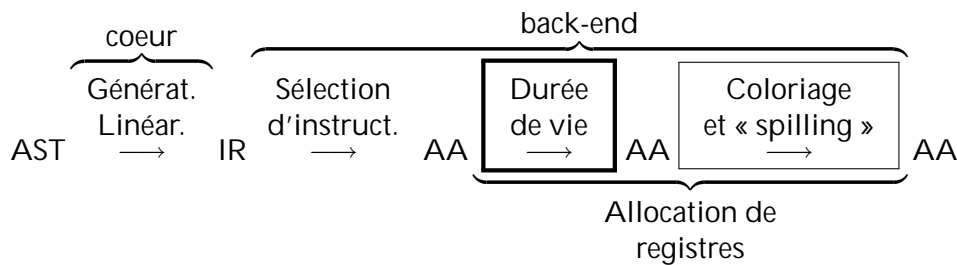
Nous allons maintenant présenter la première de ces approches.

**Remerciements** : une partie de ces notes est basée sur le cours de Didier Rémy

Les grandes lignes de l'allocation par coloriage de graphe :

- on construit à partir de l'assembleur abstrait le *graphe de flot* du programme
- en analysant ce graphe, on calcule la *durée de vie* de chaque temporaire
- on remarque que deux temporaires ayant durées de vie *disjointes* peuvent *partager un même registre*, sinon, ils *interfèrent*
- on construit le *graphe d'interférence* : les noeuds sont les temporaires, les arêtes relient deux noeuds qui interfèrent
- si on arrive à *k-colorier* le graphe, on sait placer tous les temporaires dans *k* registres
- sinon, on met en pile un temporaire (« spill »), et on recommence

### Analyse de la durée de vie des temporaires



Calculer *pour chaque instruction* la liste des *temporaires vivants*.

#### 8.1.2 Temporaires définis, temporaires lus

Une instruction

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

Les deux ensembles ne sont pas nécessairement disjoints.

#### Exemple

L'instruction

OP (PLUS, 124, 124, 126)

peut s'écrire, avec la notation du « code à trois adresses »

$t124 \leftarrow t124 + t126$

Cette instruction *utilise* les variables *t124*, *t126* et *définit* *t124*.

#### Le cas de l'appel de fonction

L'instruction *jal* (qui implemente le CALL) se traite de façon spéciale :

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

## Assembleur abstrait et temporaires

Pour chaque instruction de l'assembleur abstrait il faut connaître l'ensemble des temporaires lus et définis par cette instruction. On pourra par exemple écrire une fonction qui prend une instruction et renvoie ces deux ensembles.

### 8.1.3 Graphe de flot et durée de vie des variables

#### Le graphe d'un programme

Un programme peut être vu comme un graphe :

- les noeuds sont les instructions
- les arcs décrivent la possibilité de passer d'une instruction à une autre et sont étiquetés par des *conditions* (mutuellement exclusives)

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

#### La durée de vie d'une variable

Un temporaire est dit

**vivant** sur un arc du graphe si sa valeur est utilisée dans une instruction **suivante** (dans le flot du contrôle) avant d'être redéfinie.

**vivant à la sortie** (live-out) d'une instruction  $i$  si il est vivant sur un des arcs sortant de  $i$ .

**vivant à l'entrée** (live-in) d'une instruction  $i$  si il est vivant sur un des arcs entrant de  $i$ .

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

Lorsqu'un temporaire est mort, sa valeur, quoique bien définie, n'a pas d'importance.

#### Approximation dans le calcul de la durée de vie d'une variable

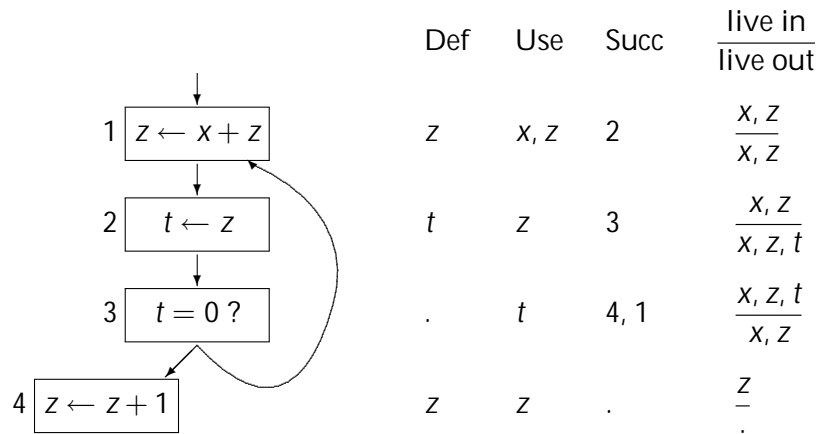
Les conditions de branchement sont des conditions dynamiques : elles peuvent dépendre d'un calcul arbitrairement complexe.

Elles ne sont donc pas décidables, donc...

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

On majore grossièrement les conditions de saut : on fait comme si on pouvait sauter toujours à n'importe laquelle des étiquettes du saut.

#### Un (sous) programme



### 8.1.4 Calcul de la durée de vie des temporaires

#### Comment *calculer* la durée de vie d'un temporaire ?

Pour chaque instruction  $i$ , on définit

- $\text{Use}(i)$  l'ensemble des temporaires utilisés (lus) par  $i$
- $\text{Def}(i)$  l'ensemble des temporaires définis (écrits) par  $i$
- $\text{In}(i)$  l'ensemble des temporaires vivants à l'entrée de  $i$
- $\text{Out}(i)$  l'ensemble des temporaires vivants à la sortie de  $i$
- $\text{Succ}(i)$  l'ensemble des successeurs immédiats de  $i$

On remarque la propriété fondamentale de  $\text{Out}(i)$  et  $\text{In}(i)$  :

*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  $i$ .*

Il s'agit alors de calculer une *solution* des équations suivantes :

$$\begin{cases} \text{Out}(i) = \bigcup_{i' \in \text{Succ } i} \text{In}(i') \\ \text{In}(i) = \text{Use}(i) \cup (\text{Out}(i) \setminus \text{Def}(i)) \end{cases}$$

Toute solution est un *point fixe* des équations.

Par le théorème de Knaster-Tarski, nous savons que la famille des solutions est un treillis, il y a donc en particulier

- un plus petit point fixe (ce que nous cherchons)
- un plus grand point fixe

Plus grand et plus petit point fixes ne sont en général pas égaux (considérer un temporaire jamais utilisé).

#### Algorithme

Les équations définissent des fonctions continues sur le treillis des ensembles de temporaires, donc on peut calculer le plus petit point fixe comme

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

où

$$\begin{cases} \text{Out}^n(i) = \bigcup_{i' \in \text{Succ}(i)} \text{In}^{n-1}(i') \\ \text{In}^n(i) = \text{Use}(i) \cup (\text{Out}^n(i) \setminus \text{Def}(i)) \end{cases} \quad \begin{cases} \text{In}^0(i) = \emptyset \\ \text{Out}^0(i) = \emptyset \end{cases}$$

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

### 8.1.5 Complexité de l'analyse

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

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

#### Accélération de la convergence

Lorsque  $\text{In}^n$  est déjà connu, on peut remplacer  $\text{In}^{n-1}$  par  $\text{In}^n$  dans le calcul de  $\text{Out}^n$  : il est donc avantageux de calculer l'instruction  $\text{Succ}(i)$  avant l'instruction  $i$ .

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

Mais comme la plupart des instructions sont simplement des sauts à l'instruction suivante, i.e.  $\text{Succ}(i) = \{i+1\}$ , 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 l'exemple

Données				Calcul normal				accéléré	
i	Def	Use	Succ	$\frac{in_1}{out_1}$	$\frac{in_2}{out_2}$	$\frac{in_3}{out_3}$	$\frac{in_4}{out_4}$	$\frac{in'_1}{out'_1}$	$\frac{in'_2}{out'_2}$
1	z	x, z	2	$\frac{x, z}{.}$	$\frac{x, z}{z}$	$\frac{x, z}{z}$	$\frac{x, z}{x, z}$	$\frac{x, z}{z}$	$\frac{x, z}{x, z}$
2	t	z	3	$\frac{z}{.}$	$\frac{z}{t}$	$\frac{x, z}{x, z, t}$	$\frac{x, z}{x, z, t}$	$\frac{z}{z, t}$	$\frac{x, z}{x, z, t}$
3	.	t	4, 1	$\frac{t}{.}$	$\frac{x, z, t}{x, z}$	$\frac{x, z, t}{x, z}$	$\frac{x, z, t}{x, z}$	$\frac{z, t}{z}$	$\frac{x, z, t}{x, z}$
4	z	z	.	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$	$\frac{z}{.}$

#### Optimisation de l'analyse : blocs de base de AA

Sur l'assembleur abstrait aussi on peut définir des « bloc de base » : une suite d'instructions ne contenant aucun saut ni point d'entrée, autre que l'entrée à la première instruction et la sortie à la fin<sup>1</sup>.

<sup>1</sup>Ce sont les mêmes propriétés que pour les blocs de base de l'IR.

On peut traiter un « bloc de base » comme une « macro-instruction » :

$\begin{array}{l} z \leftarrow x + y \\ t \leftarrow z \end{array}$	utilise $x, y$ définit $z, t$	<b>Important</b> : la valeur locale de $z$ est utilisée dans le bloc, mais pas sa valeur à l'entrée du bloc !
---	----------------------------------	---

### Calcul sur les blocs de base

Sur les blocs de base, fait l'analyse en deux étapes :

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

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

On définit les variables vivantes  $\text{Out}(b)$  à la sortie du bloc  $b$  comme précédemment.

2. On calcule les  $\text{Out}(i)$  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 noeuds (donc d'opérations).

#### 8.1.6 Mise en œuvre

Un noeud est une instruction annotée par les informations  $\text{Def}$ ,  $\text{Use}$  et  $\text{Succ}$  ainsi que les champs mutables  $\text{In}$  et  $\text{Out}$ .

```
type flowinfo = {
  instr : Assem.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 : Assem.instr list -> flowinfo list;;
```

On itère le calcul jusqu'à ce que plus rien ne change :

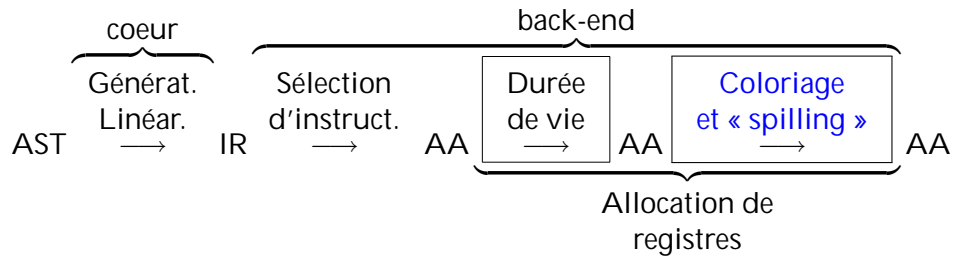
```
fixpoint : flowinfo list -> flowinfo list;;
```

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

## 8.2 Allocation des registres par coloriage de graphe

### 8.2.1 Introduction

## Coloriage



- Assigner à *chaque* groupe de temporaires qui n'interfèrent pas un registre différent.
- En cas d'échec, choisir un temporaire pour l'allouer en pile (spill), et recommencer

### 8.2.2 Interférence

#### La bonne notion d'interférence

Pour construire le graphe d'interférence  $GI = (N, A)$ , on veut procéder comme suit

**Noeuds** il y a un noeud pour chaque temporaire  $t$

**Arcs** on ajoute un arc entre  $t$  et  $t'$  si et seulement si  $t$  et  $t'$  interfèrent.

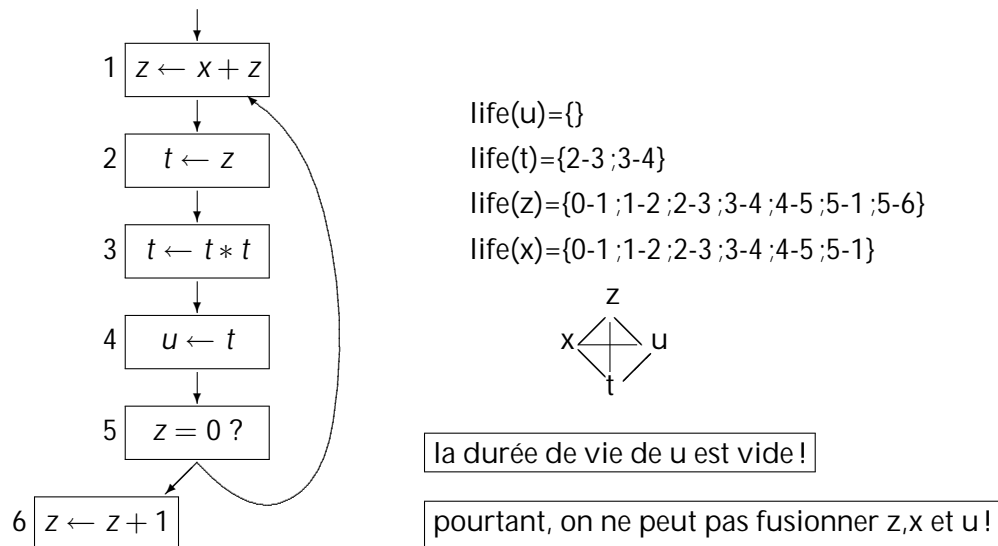
Mais qu'est-ce que cela signifie exactement *interférer*?

#### Interférence

On peut essayer de dire que  $t$  et  $t'$  interfèrent si leur durées de vie ne sont pas disjointes...

Ceci est presque bon, à cela près que dans certain cas on peut avoir des temporaires de durée de vie vide, mais qui ne peuvent être partagés avec certains autres temporaires ...

#### Exemple



### Construction du graphe d'interférence

Donc, on doit dire que un temporaire  $t$  interfère avec  $t'$  s'il est *défini (écrit)* pendant la durée de vie de  $t'$  ou vice-versa.

On construit alors le *graphe d'interférence*  $GI = (N, A)$  comme suit

**Noeuds** il y a un noeud pour chaque temporaire  $t$

**Arcs** on parcourt la liste d'instructions, et pour chaque instruction  $I$  qui définit un temporaire  $t$  on ajoute un arc entre  $t$  et  $t'$  pour tout  $t'$  qui est vivant en sortie de  $I$

### Construction du graphe d'interférence avec traitement des MOVE

Il n'est pas nécessaire d'introduire des conflits entre  $t$  et  $t'$  juste à cause d'une instruction MOVE :

```
t <- t'
x <- t+3
y <- t+t'
```

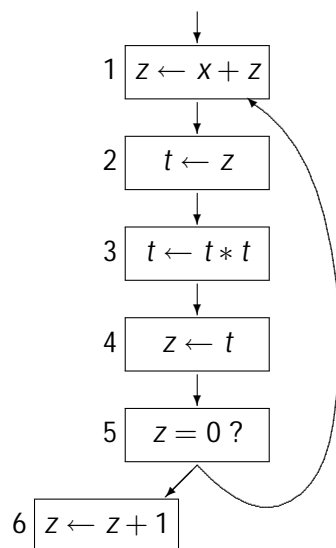
On modifie alors la construction du *graphe d'interférence*  $GI = (N, A)$  comme suit

**Noeuds** il y a un noeud pour chaque temporaire  $t$

**Arcs** on parcourt la liste d'instructions, et :

- pour chaque instruction  $I$  *différent de MOVE*, et qui définit un temporaire  $t$  on ajoute un arc entre  $t$  et  $t'$  pour tout  $t'$  qui est vivant en sortie de  $I$
- pour chaque instruction MOVE  $t \leftarrow u$ , et qui définit un temporaire  $t$  on ajoute un arc entre  $t$  et  $t'$  pour tout  $t'$  *différent de  $u$*  qui est vivant en sortie de  $I$  ; on ajoute un arc spécial MOVE entre  $t$  et  $u$

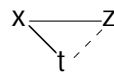
### Exemple



life(t)={2-3;3-4}

life(z)={0-1;1-2;4-5;5-6;5-1}

life(x)={0-1;1-2;2-3;3-4;4-5;5-1}



deux registres suffisent !

mieux : on peut éliminer les MOVE !



### 8.2.3 Coloriage de graphe

**Problème** Étant donné un graphe et un ensemble de  $K$  couleurs, il s'agit d'attribuer une couleur à chaque noeud du graphe de telle façon qu'un arc relie toujours des noeuds de couleurs différentes.

**Complexité** Le problème est NP-complet.

#### Une solution approchée du coloriage

**Principe** si un noeud  $n$  est de degré (degré = nombre de voisins) *strictement plus petit* que  $K$  et si le graphe  $G - \{n\}$  est  $K$ -coloriable, alors le graphe  $G$  est  $K$ -coloriable. En effet, une fois  $G - \{n\}$   $K$ -colorié il reste au moins une couleur qui ne soit pas celle d'un voisin de  $n$ .

**Procédure récursive** retirer les noeuds de faible degré (plus petit que  $K$ ). Cela diminue le degré des noeuds restant et permet de continuer au mieux jusqu'à ce que le graphe soit vide.

Dans ce cas le graphe est coloriable, et on est certain de pouvoir attribuer correctement les couleurs au retour de la procédure récursive.

Sinon, le graphe *peut* (La solution est *approchée* !) ne pas être coloriable.

#### Stratégie optimiste

La procédure récursive peut être améliorée avec une simple heuristique optimiste :

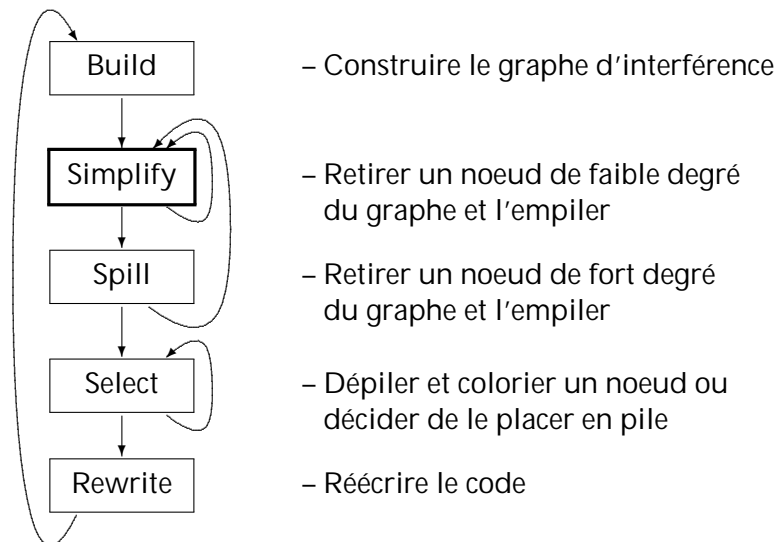
**À l'aller** : Lorsque tous les noeuds sont de fort degré le graphe peut ne pas être coloriable. On retire un noeud qui sera *éventuellement* placé en pile, et on poursuit quand même la procédure.

**Au retour** : On essaye de colorier ce noeud : en effet la solution précédente étant approchée, il se peut que le graphe soit malgré tout coloriable.

Si ce n'est pas possible, alors on décide de placer *définitivement* le noeud en pile, sans lui attribuer de couleur, et on poursuit.

**À la fin** : S'il y a eu des placements en pile, il faut réécrire le code, et recommencer : en effet, le placement en pile introduit de nouveaux temporaires...

#### Vue d'ensemble de la procédure



## Le rôle des heuristiques

Lorsqu'un noeud de fort degré est placé en pile, le choix de ce noeud parmi les candidats possibles est très important. On veut :

- que ce choix ait un effet maximal, débloquent d'autres noeuds : il faut choisir un noeud **de degré élevé** ;
- que la sauvegarde en pile ne soit pas trop coûteuse : choisir un noeud (temporaire) **peu utilisé** (dont le nombre d'occurrences dans *Def* et *Use* est faible).

Il faut trouver un compromis entre l'efficacité du spill et son coût.

On utilise une heuristique (sous forme d'une fonction de priorité)  $p$  pour choisir le noeud. Par exemple, on peut fixer  $p(n) = d(n)/u(n)$  où  $u(n)$  est le nombre d'utilisations du noeud  $n$  et  $d(n)$  est son degré. Il faudrait pondérer chaque utilisation par une estimation de sa fréquence d'utilisation : une instruction à l'intérieur d'une boucle interne sera exécutée plus souvent qu'une instruction plus externe.

## Itération et Terminaison

Lorsqu'il y a un placement en pile, la réécriture du code introduit de nouveaux temporaires (dits *auxiliaires*).

Ces temporaires auxiliaires ont une durée de vie très courte : ils trouveront donc souvent leur place dans des interstices.

Mais il peut arriver qu'il soit nécessaire d'allouer d'autres temporaires en pile pour faire de la place pour les temporaires auxiliaires : il faut itérer jusqu'à ce qu'il n'y ait plus de spill.

*En général*, une ou deux itérations suffisent.

### Terminaison

La procédure **peut** à priori boucler !

Cela se produit si un temporaire auxiliaire est à nouveau placé en pile, ce qui n'a pas de sens (son propre auxiliaire lui sera isomorphe).

On peut détecter le risque de non terminaison et lever une exception si il n'y a plus d'autres solutions que celle de placer un temporaire auxiliaire en pile. Il s'agit alors d'une erreur de conception ou de réglage (trop peu de registres  $t$  ou bien d'une mauvaise fonction de priorité).

Mais nous *savons* que avec seulement 2 registres libres on peut compiler n'importe quel code à 3 adresses parce-que nous avons fait ça dans l'allocation naïve des registres, donc sur une machine avec au moins 2 registres on ne risque de boucler que si la priorité est mal réglée.

## 8.2.4 Élimination des instructions MOVE

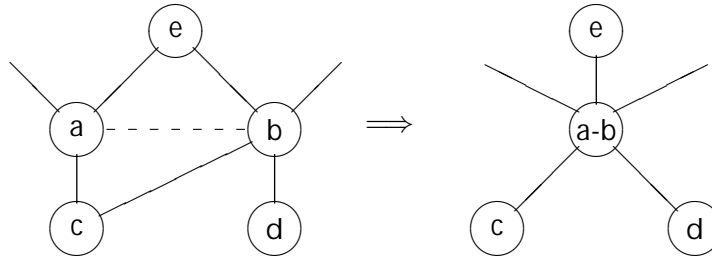
On a vu que si deux temporaires liés par une instruction **MOVE** reçoivent la même couleur, on peut alors supprimer cette instruction.

On peut modifier la procédure pour essayer de favoriser ces éliminations :

- on garde trace des instructions **MOVE** dans le graphe d'interférence (par des arcs spéciaux)
- on **fusionne** les noeuds reliés par ces arcs, **s'ils n'interfèrent pas**

## Fusion (coalescence) et son effet

On représente les **MOVE** par des arcs en pointillés.



Le degré des noeuds *c* et *e* **diminue**.

Le degré des noeuds *a* et *b* **augmente** (Danger : on peut perdre la *k*-coloriabilité!).

Le degré des autres noeuds est **inchangé**.

### Une stratégie de fusion sûre

Ne fusionner les noeuds (*a*) et (*b*) dans le graphe *G* que si cela préserve sa coloriabilité.

#### Deux critères sûrs (mais approchés)

1. Après fusion le noeud (*a-b*) a moins de *K* voisins de fort degré.
2. Le noeud (*a*) est tel que tous ses voisins de fort degré interfèrent avec (*b*).

**Preuve** après avoir éliminé tous les noeuds de faible degré dans le graphe résultant

1. le noeud (*a-b*) a moins de *K* voisins et peut aussi être retiré.
2. le noeud (*a-b*) peut être identifié avec le noeud (*b*) de *G*.

Dans les deux cas, il reste un sous-graphe de *G*, donc coloriable.

### Allocation combinée

Les noeuds qui n'ont pas d'arc **MOVE** sont dits *simplifiables*, les autres sont dits *complexes*.

La fusion augmente le degré des noeuds fusionnés : on ne peut donc pas simplifier un noeud tant qu'il est complexe.

Donc, on fait une des actions suivantes, par ordre de priorité :

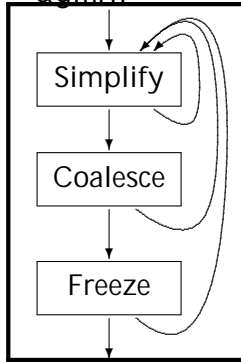
1. on retire du graphe un noeud simplifiable de faible degré (comme précédemment) ;
2. on effectue une fusion sûre (qui préserve la coloriabilité) ;
3. on retire tous les arcs **MOVE** d'un noeud complexe de faible degré (abandonnant tout espoir de le fusionner) ce qui le rend simplifiable ;
4. on retire un noeud de fort degré qui sera a priori placé en pile (spill).

### Allocation combinée, graphiquement

Remplacer la boîte **Simplify** de l'algorithme précédent par :

..(\* agmrrr

'ienc :)



– Retirer du graphe un noeud simplifiable de faible degré

– Effectuer une fusion sûre

– Retirer un arc **MOVE**

Au pire, aucune fusion n'est possible : les arcs **MOVE** sont retirés un à un et on retombe sur (une trace de) l'algorithme précédent.

### 8.2.5 Mise en œuvre

Au cours du calcul, on maintient des ensembles de noeuds (et d'arcs) qui sont dans différents états (simple/complex, faible/fort degré, etc.). À chaque étape on choisit un noeud qui est dans un certain état pour le placer dans un autre état (et ajuster le graphe).

Le module `partition` avec l'interface ci-dessous permet de maintenir à jour l'ensemble des noeuds (et d'arcs) dans chaque état de façon efficace (paresseuse).

```
type 'a partition      type 'a elem
val make : int -> 'a partition array
val create : 'a partition -> 'a -> 'a elem
val info : 'a elem -> 'a
val belong : 'a elem -> 'a partition -> bool
val move : 'a elem -> 'a partition -> unit
val pick : 'a partition -> 'a elem
```

### Représentation des noeuds et des arcs

```
type node_info = {
  temp : temp;                                (* temporaire associé *)
  mutable moves : move list;                   (* arcs moves *)
  mutable adj : node list;                     (* arcs d'interference *)
  mutable degree : int;
  mutable
```

## Les partitions de noeuds

On distingue les noeuds :

- pré-coloriés : (registres du processeur) ne changeront pas d'état
- initiaux : état temporaire de tous les noeuds non coloriés juste après la construction du graphe d'interférence ;
- simplifiables de faible degré : candidats à la simplification ;
- complexes de faible degré : candidats pour être gelés (i.e. pour geler leurs arcs **MOVE**) ;
- de fort degré : candidats pour être placés en pile ;
- spilled : définitivement programmés pour être placés en pile ;
- fusionnés : leur champ alias contient le noeud résultant ;
- empilés : en attente d'être coloriés (l'ordre est essentiel) ;
- coloriés : état final de tous les noeuds non pré-coloriés.

## Les partitions d'arcs **MOVE**

On distingue les arcs :

- fusionnés : ne seront plus considérés.
- contraints : ne peuvent être fusionnés car source et destination interfèrent. Ne seront plus considérés.
- gelés : ceux pour lesquels la fusion a été définitivement abandonnée ; ils ont été retirés du graphe, et ne seront plus considérés.
- candidats à la fusion.
- candidats à la fusion temporairement bloqués tant que leur fusion n'est pas sûre. Ils seront reconsidérés (candidats à la fusion) dès que le degré d'un de leurs voisins aura diminué.

## Implémentation

L'implémentation suit l'algorithme pas à pas : sélectionner un noeud ou un arc à traiter, le traiter, puis mettre le graphe (i.e. l'état des noeuds et des arcs voisins) à jour.

### Ré-écriture du code

Dans le cas où il y a des registres à placer en pile, il faut réécrire le code et recommencer.

On peut prendre en compte les fusions qui ont eu lieu avant le premier spill. En effet, celles-ci seront forcément reproductibles. Par contre, il faut ignorer toutes les fusions qui ont eu lieu après.

Pour cela, on sauvegarde l'état des temporaires et des **MOVE** déjà fusionnés au moment du premier spill.

## Flexibilité de l'approche

L'algorithme de coloriage de graphe joue le rôle d'un solveur de contraintes.

En jouant sur les contraintes qu'on lui demande de résoudre, on peut obtenir automatiquement pratiquement tous les traitements des registres spéciaux :

```
t' <- s0  
t'' <- s1  
...
```

et à l'épilogue une suite

```
...  
s1 <- t''  
s0 <- t'
```

- au contraire, les caller save (t0, t1,...) sont écrasés par un CALL, et l'allocateur va les sauver dans d'autres registres ou alors les mettre en pile si nécessaire.

**traitement automatique de la sauvegarde de FP et RA** en traitant ces deux registres comme des caller save

**traitement des particularité des processeurs** certains processeurs fournissent des optimisations utilisables seulement en respectant une certaine discipline d'usage des registres ; cette discipline peut souvent se coder dans la notion d'interférence

# Bibliographie

- [CAC<sup>+</sup>81] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register allocation via coloring. *Computer Languages*, 6 :45–57, January 1981.
- [PS99] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5) :895–913, 1999.