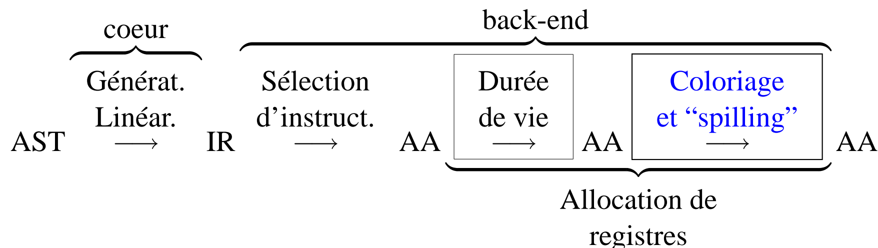
Remerciements: une partie de ces notes est basée sur le cours de Didier Remy

Allocation des registres par coloriage de graphe

Les grandes lignes:

- 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 qui n'interfèrent pas peuvent *partager un même registre*
- 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* toute l'allocation

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

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.

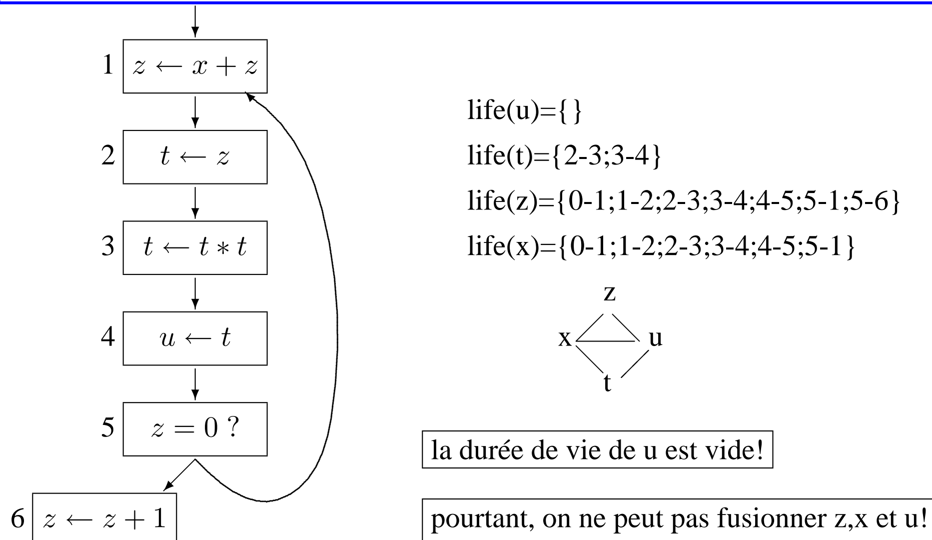
Mais qu'est-ce que cela signifie exactement *interférer*?

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 interfére avec t' s'il est *défini (écri)* pendant la durée de vie de t' ou viceversa.

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 nécéssaire d'introduire des conflits entre t et t' juste a cause d'une instruction MOVE:

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

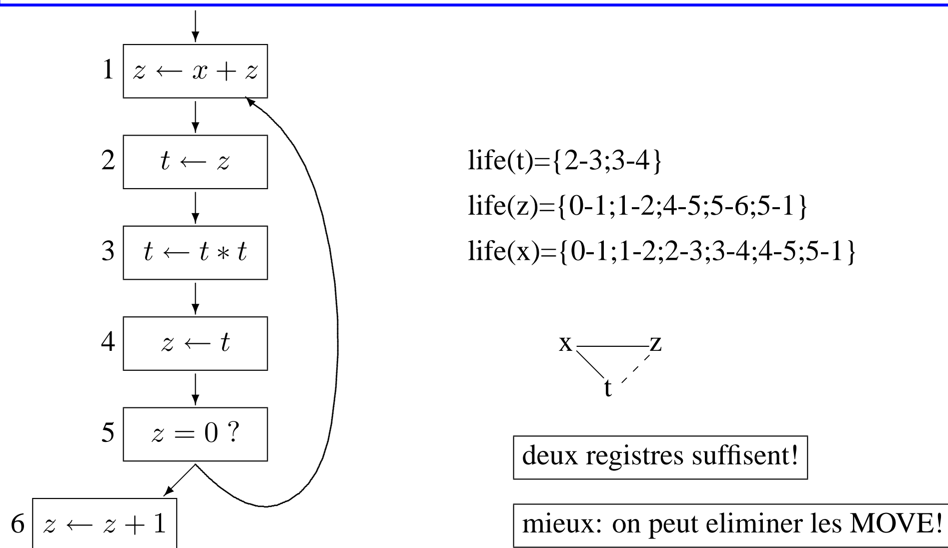
On modifie alors la construction du *graphe d'interférence* $GI = (N, A)$ comme suit

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<-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



Coloriage de graphe

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

Complexité Le problème est NP-complet.

Une solution approchée du coloriage

Principe : si un nœud n est de degré[1] *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.

[1] degré = nombre de voisins

Procédure récursive retirer les nœuds de faible degré (plus petit que K). Cela diminue le degré des nœuds 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*[2] ne pas être coloriable.

Stratégie optimiste

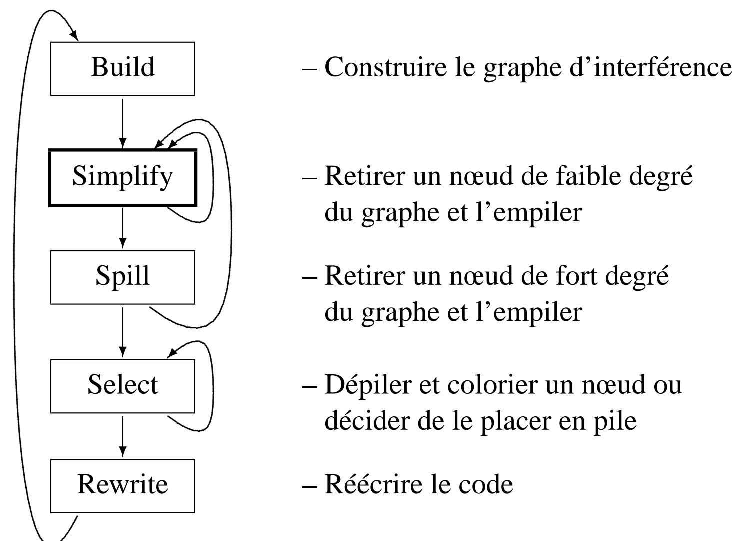
La procédure récursive peut être améliorée avec une simple euristique optimiste:

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

Au retour On essaye de colorier ce nœud: 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 nœud 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 nœud de fort degré est placé en pile, le choix de ce nœud parmi les candidats possibles est trés important. On veut:

- que ce choix ait un effet maximal, débloquant d'autres nœuds: il faut choisir un nœud de degré élevé;

[2] La solution est *approchée!*

- que la sauvegarde en pile ne soit pas trop coûteuse: choisir un nœud (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[3] 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 nœud 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 lequel 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.

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.

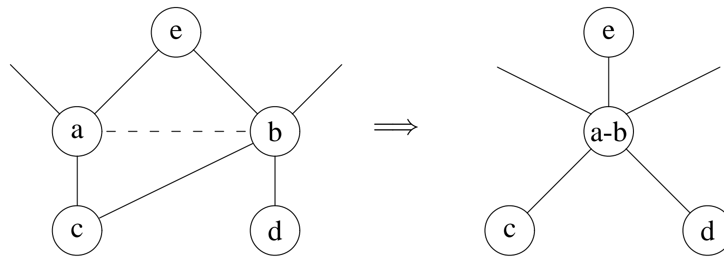
On peut modifier la procedure pour essayer de favoriser ces éliminations:

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

Fusion (coalescence) et son effet

On représente les MOVE par des arcs en pointillés.

[3] sous forme d'une fonction de priorité



Le degré des nœuds c et e diminue.
Le degré des nœuds a et b augmente[4].
Le degré des autre nœuds est inchangé.

Une stratégie de fusion sure

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

Deux critères surs (mais approchés)

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

Preuve : après avoir éliminé tous les nœuds de faible degré dans le graphe résultant

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

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

Allocation combinée

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

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

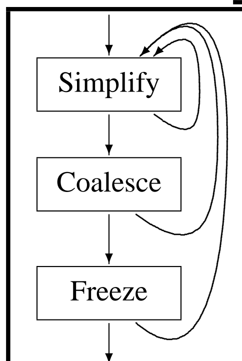
Donc, on fait une des actions suivantes, par ordre de priorité:

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

[4] Danger: on peut perdre la k-coloriabilité!

Allocation combinée, graphiquement

Remplacer la boîte Simplify de l'algorithme précédent par:



- Retirer du graphe un nœud simplifiable de faible degré
- Effectuer un fusion sure
- 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.

Mise en œuvre

Au cours du calcul, on maintient des ensembles de nœuds (et d'arcs) qui sont dans différents états (simple/complexe, faible/fort degré, etc.).
A chaque étape on choisit un nœud 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 nœuds (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 nœuds et des arcs

```
type node_info = {
    temp : temp;                    (* temporaire associ\'e *)
    mutable moves : move list;      (* arcs moves *)
    mutable adj : node list;        (* arcs d'interference *)
    mutable degree : int;
    mutable alias : node option;    (* noeud fusionn\'e *)
    mutable color : int;
    mutable occurs : int;           (* nombre d'utilisations *)
  }
and move_info =
    { instr : Ass.instr; left : node; right : node; }
and node = node_info elem
```

```
and move = move_info elem;;
```

Les partitions de nœuds

On distingue les nœuds:

- pré-coloriés: ne changeront pas d'état
- initiaux: état temporaire de tous les nœuds 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;
- fusionés: leur champ alias contient le nœud résultant;
- empilés: en attente d'être coloriés (l'ordre est essentiel);
- coloriés: état final de tous les nœuds 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.
- gélés: 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 sure. 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 nœud ou un arc à traiter, le traiter, puis mettre le graphe (i.e. l'état des nœuds 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 resoudre, on peut obtenir automatiquement pratiquement tous les traitements des registres spéciaux:

traitement des caller/callee save

- en indiquant dans les DEF du CALL les registres ecrasés, mais pas les callee save (s0, s1 ...), on dit à l'allocateur qu'il peut disposer des registres callee save pendant toute la procedure. Mais en même temps, il faut sauvegarder les registre que nous utilisons!
  Pour cela il suffit d'ajouter au prologue une suite
  (avec des t' nouveaux)

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

  et à l'épilogue une suite

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

- au contraire, les caller save (t0, t1, ...) sont ecrasés par un CALL, et l'allocatuer 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 optimisation utilisables seulement en respectant une certaine discipline d'usage des registres; cette discipline peut souvent se coder dans la notion d'interférence