Allocation des registres par coloriage de graphe

Nous avons vu comment, de façon extrémement naïve, il est possible de produire du code assembleur éxécutable 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 réduit énormement les performances. Il existent des approches plus sophistiquées qui permettent de réduire significativement l'allocation en pile:

- allocation de registres pas coloriage de graphes [CAC+81]
- allocation linéaire [PS99]

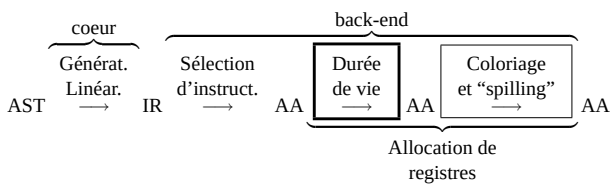
Nous allons maintenant présenter la première de ces approches.
Remerciements: une partie de ces notes est basée sur le cours de Didier Remy

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 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*.

Analyse de la durée de vie des temporaires

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

```
OPER {o_assem="add 'd0 's0 's1";
      o_dst=[Temp "t124"];
      o_src=[Temp "t124";Temp "t126"]; jump=None};;
```

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

$$t124 \leftarrow t124 + t126$$

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

On voit bien l'utilité de l'assembleur abstrait, maintenant!

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.

Le graphe d'un programme

Un programme peut être vu comme un graphe:

- les nœuds 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**[1] 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

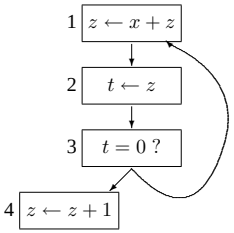
Les conditions de branchement sont des conditions dynamiques: elles peuvent dépendre d'un calcul arbitrairement complexe.

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



	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	.	$\frac{z}{.}$

[1] dans le flot du contrôle

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

Pour chaque instruction i, on définit

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

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

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:

$$\begin{cases} \texttt{Out}\,(i) = \bigcup\limits_{i' \in \texttt{Succ}\,i} \texttt{In}\,(i') \\ \texttt{In}\,(i) = \texttt{Use}\,(i) \cup (\texttt{Out}\,(i) \setminus \texttt{Def}\,(i)) \end{cases}$$

Toute solution est un *point fixe* des équations.

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

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

Plus grand et plus petit point fixes ne sont en géneral pas égaux[2].

Algorithme

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

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

où

$$\begin{cases} \texttt{Out}^{\,n}(i) = \bigcup\limits_{i' \in \texttt{Succ}\,(i)} \texttt{In}^{\,n-1}(i') \\ \texttt{In}^{\,n}(i) = \texttt{Use}\,(i) \cup (\texttt{Out}^{\,n}(i) \setminus \texttt{Def}\,(i)) \end{cases} \qquad \begin{cases} \texttt{In}^{\,0}(i) = \emptyset \\ \texttt{Out}^{\,0}(i) = \emptyset \end{cases}$$

En effet, 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.

Complexité de l'analyse

[2] considérer un temporaire jamais utilisé

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.

Accélération de la convergence

Lorsque que $\texttt{In}^n$ est déjà connu, on peut remplacer $\texttt{In}^{n-1}$ par $\texttt{In}^n$ dans le calcul de $\texttt{Out}^n$: il est donc avantageux de calculer l'instruction $\texttt{Succ}$ 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. $\texttt{Succ}\,(i) = \{i+1\}$, 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	$Succ$	$\frac{in_0}{out_0}$	$\frac{in_1}{out_1}$	$\frac{in_2}{out_2}$	$\frac{in_3}{out_3}$	$\frac{in'_0}{out'_0}$	$\frac{in'_1}{out'_1}$
1	z	x,z	1	$\frac{x,z}{.}$	$\frac{x,z}{z}$	$\frac{x,z}{z}$	$\frac{x,z}{x,z}$	$\frac{x,z}{z}$	$\frac{x,z}{x,z}$
2	t	z	2	$\frac{z}{.}$	$\frac{z}{t}$	$\frac{x,z}{x,z,t}$	$\frac{x,z}{x,z,t}$	$\frac{z}{z,t}$	$\frac{x,z}{x,z,t}$
3	.	t	$3,1$	$\frac{t}{.}$	$\frac{x,z,t}{x,z}$	$\frac{x,z,t}{x,z}$	$\frac{x,z,t}{x,z}$	$\frac{z,t}{z}$	$\frac{x,z,t}{x,z}$
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 definir 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[3].

On peut traiter un "bloc de base" 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!*

[3] Ce sont les mêmes propriétés que pour les blocs de base de l'IR.

Calcul sur les blocs de base

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

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

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

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

2. On calcule les $\texttt{Out}\,(i)$ 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).

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 : 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 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.

References

[CAC+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.