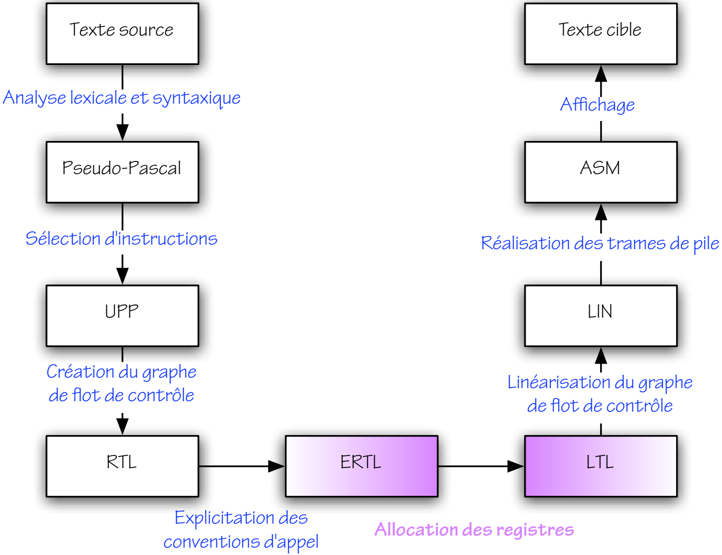
Compilation (INF 564)

Analyse de durée de vie
Construction du graphe d'interférences

François Pottier

27 janvier 2016


Texte source
Analyse lexicale et syntaxique
Pseudo-Pascal
Sélection d'instructions
UPP
Création du graphe de flot de contrôle
RTL
Explicitation des conventions d'appel
ERTL
Allocation des registres
LTL
Linéarisation du graphe de flot de contrôle
LIN
Réalisation des trames de pile
ASM
Affichage
Texte cible

La fonction factorielle en ERTL

Comment *réaliser* les pseudo-registres %3, %0, %4, et %1 ?

```
procedure f(1)
var %0, %1, %2, %3, %4, %5, %6
entry f11
f11: newframe              → f10
f10: move   %6, $ra        → f9
f9 : move   %5, $s1        → f8
f8 : move   %4, $s0        → f7
f7 : move   %0, $a0        → f6
f6 : li     %1, 0          → f5
f5 : blez   %0             → f4, f3
f3 : addiu  %3, %0, −1     → f2
f2 : j                     → f20
f20: move   $a0, %3        → f19
f19: call   f(1)           → f18
f18: move   %2, $v0        → f1
f1 : mul    %1, %0, %2     → f0
f0 : j                     → f17
f17: move   $v0, %1        → f16
f16: move   $ra, %6        → f15
f15: move   $s1, %5        → f14
f14: move   $s0, %4        → f13
f13: delframe              → f12
f12: jr     $ra              (xmits $v0)
f4 : li     %1, 1          → f0
```

Durée de vie et interférence

Cette réflexion doit avoir illustré trois points :

- il faut savoir en quels points du code chaque variable est *vivante* — contient une valeur susceptible d'être utilisée à l'avenir;
- deux variables peuvent être réalisées par le même emplacement *si elles n'interfèrent pas* — si l'on n'écrit pas dans l'une tandis que l'autre est vivante.
- les instructions **move** suggèrent des emplacements préférentiels.

La phase d'allocation de registres doit donc être précédée d'une *analyse de durée de vie*, d'où on déduit un *graphe d'interférences*.

La définition du mot *"variable"* sera précisée sous peu.

Analyse de la fonction factorielle

Voici les résultats de l'analyse de durée de vie. On a affiché l'ensemble des variables vivantes *à la sortie* de chaque instruction :

```
procedure f(1)
var %0, %1, %2, %3, %4, %5, %6
entry f11
f11: newframe          → f10       $a0, $s0, $s1, $ra
f10: move   %6, $ra    → f9        %6, $a0, $s0, $s1
f9 : move   %5, $s1    → f8        %5, %6, $a0, $s0
f8 : move   %4, $s0    → f7        %4, %5, %6, $a0
f7 : move   %0, $a0    → f6        %0, %4, %5, %6
f6 : li     %1, 0      → f5        %0, %4, %5, %6
f5 : blez   %0         → f4, f3    %0, %4, %5, %6
f3 : addiu  %3, %0, −1 → f2        %0, %3, %4, %5, %6
f2 : j                 → f20       %0, %3, %4, %5, %6
f20: move   $a0, %3    → f19       %0, %4, %5, %6, $a0
f19: call   f(1)       → f18       %0, %4, %5, %6, $v0
f18: move   %2, $v0    → f1        %0, %2, %4, %5, %6
f1 : mul    %1, %0, %2 → f0        %1, %4, %5, %6
f0 : j                 → f17       %1, %4, %5, %6
f17: move   $v0, %1    → f16       %4, %5, %6, $v0
f16: move   $ra, %6    → f15       %4, %5, $v0, $ra
f15: move   $s1, %5    → f14       %4, $v0, $s1, $ra
f14: move   $s0, %4    → f13       $v0, $s0, $s1, $ra
f13: delframe          → f12       $v0, $s0, $s1, $ra
f12: jr     $ra        (xmits $v0)
f4 : li     %1, 1      → f0        %1, %4, %5, %6
```

Utilisez les options -few -dertl -dlive pour reproduire cela.

Registres allouables et variables

On souhaite *réaliser* chaque pseudo-registre par un registre physique, ou, à défaut, un emplacement de pile.

On s'intéresse uniquement aux registres physiques dits *allouables*, c'est-à-dire *non réservés* pour un usage particulier, comme le sont par exemple \$gp ou \$sp.

Les registres exploités par la convention d'appel, à savoir \$v0, \$a0-\$a3, et \$ra *sont* allouables. Les registres \$t0-\$t9 et \$s0-\$s7 le sont également. La liste complète est donnée par *MIPS.allocatable*.

L'analyse de durée de vie concerne les pseudo-registres et les registres physiques allouables. Je les appelle collectivement *variables.* L'analyse ne concerne pas les emplacements mémoire.

Vivants et morts

Voici une définition :

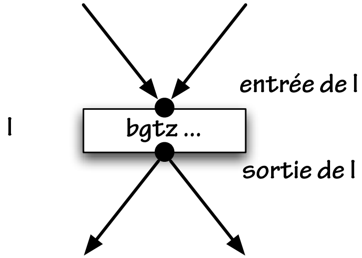
> *Une variable v est vivante ("live") au point p s'il existe un chemin menant de p à un point p′ où v est utilisée et si v n'est pas définie le long de ce chemin.*

Une variable est morte ("dead") lorsqu'elle n'est pas vivante.

Points de programme

Qu'appelle-t'on un "point de programme" p ?

On distinguera les points *"à l'entrée d'un sommet ℓ"* du graphe de flot de contrôle, et ceux *"à la sortie d'un sommet ℓ."*



Approximation

L'analyse de durée de vie est *approximative :* on vérifie s'il *existe* un chemin menant à un site d'utilisation, mais on ne se demande pas dans quelles conditions ce chemin est *effectivement emprunté.*

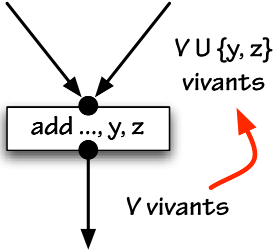
De ce fait, "vivante" signifie *"potentiellement vivante"* et "morte" signifie *"certainement morte"*.

Cette approximation est *sûre*. Au pire, si on suppose toutes les variables vivantes en tous points, on devra attribuer à chacune un emplacement physique distinct — un résultat inefficace mais correct.

"Naissance" d'une variable

Une variable *v* est *engendrée* par une instruction *i* si *i* *utilise* *v*, c'est-à-dire si *i* *lit* une valeur dans *v*.

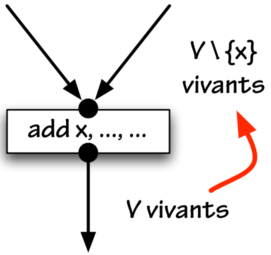
Dans ce cas, *v* est *vivante au point qui précède* immédiatement *i*.



"Mort" d'une variable

Une variable *v* est *tuée* par une instruction *i* si *i* *définit* *v*, c'est-à-dire si *i* *écrit* une valeur dans *v*.

Dans ce cas, *v* est *morte au point qui précède* immédiatement *i*.

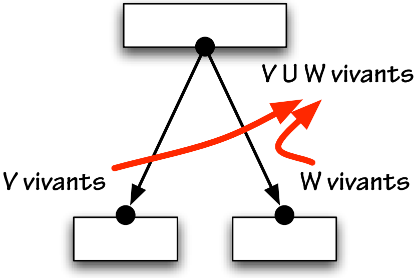


"Vie" d'une variable (I)

Si i n'engendre ni ne tue v, alors v est vivante immédiatement *avant* i si et seulement si elle est vivante immédiatement *après* i.

"Vie" d'une variable (II)

Une variable est vivante *après* i si et seulement si elle est vivante *avant* l'un quelconque des successeurs de i.



Mise en inéquations

Les assertions précédentes permettent d'exprimer le problème sous forme *d'inéquations ensemblistes*.

À chaque étiquette ℓ du graphe de flot de contrôle, on associe *deux* ensembles de variables :

- $\mathit{vivantes}_{\mathit{entrée}}(\ell)$ est l'ensemble des variables vivantes à l'entrée du sommet ℓ, c'est-à-dire juste *avant* l'instruction située en ℓ ;
- $\mathit{vivantes}_{\mathit{sortie}}(\ell)$ est l'ensemble des variables vivantes à la sortie du sommet ℓ, c'est-à-dire juste *après* l'instruction située en ℓ.

Inéquations

Les inéquations qui définissent l'analyse sont :

$$\begin{array}{rl} & ? \\ \subseteq & \text{vivantes}_{\text{sortie}}(\ell) \\ & ? \\ \subseteq & \text{vivantes}_{\text{entrée}}(\ell) \end{array}$$

Inéquations

Les inéquations qui définissent l'analyse sont :

$$\begin{array}{ll} & \text{vivantes}_{\text{entrée}}(\ell') \\ \subseteq & \text{vivantes}_{\text{sortie}}(\ell) \end{array} \qquad \textit{si } \ell \longrightarrow \ell'$$

$$\begin{array}{ll} & ? \\ \subseteq & \text{vivantes}_{\text{entrée}}(\ell) \end{array}$$

Inéquations

Les inéquations qui définissent l'analyse sont :

$$\begin{array}{ll} & \text{vivantes}_{\text{entrée}}(\ell') \\ \subseteq & \text{vivantes}_{\text{sortie}}(\ell) \qquad\qquad \textit{si } \ell \longrightarrow \ell' \end{array}$$

$$\begin{array}{ll} & (\text{vivantes}_{\text{sortie}}(\ell) \ldots ? \qquad) \ldots ? \\ \subseteq & \text{vivantes}_{\text{entrée}}(\ell) \end{array}$$

Inéquations

Les inéquations qui définissent l'analyse sont :

$$\begin{array}{ll} & \text{vivantes}_{\text{entrée}}(\ell') \\ \subseteq & \text{vivantes}_{\text{sortie}}(\ell) \end{array} \qquad \textit{si } \ell \longrightarrow \ell'$$

$$\begin{array}{ll} & (\text{vivantes}_{\text{sortie}}(\ell) \setminus \text{tuées}(\ell)) \cup \text{engendrées}(\ell) \\ \subseteq & \text{vivantes}_{\text{entrée}}(\ell) \end{array}$$

Pourquoi pas $(\text{vivantes}_{\text{sortie}}(\ell) \cup \text{engendrées}(\ell)) \setminus \text{tuées}(\ell)$?

Inéquations

Les inéquations qui définissent l'analyse sont :

$$\begin{array}{ll} & \text{vivantes}_{\text{entrée}}(\ell') \\ \subseteq & \text{vivantes}_{\text{sortie}}(\ell) \end{array} \qquad \textit{si } \ell \longrightarrow \ell'$$

$$\begin{array}{ll} & (\text{vivantes}_{\text{sortie}}(\ell) \setminus \text{tuées}(\ell)) \cup \text{engendrées}(\ell) \\ \subseteq & \text{vivantes}_{\text{entrée}}(\ell) \end{array}$$

Toute solution de ces inéquations est sûre. La *plus petite* solution donne les meilleurs résultats. (Pourquoi ?)

La recherche de la plus petite solution conduit à une analyse *en arrière :* la vivacité se propage *dans le sens inverse des arêtes* du graphe de flot de contrôle.

Communication entre procédures

La communication entre procédures se fait à travers des registres physiques qui doivent donc être considérés comme *vivants* :

- l'instruction ICall *engendre* les registres physiques dédiés au passage d'arguments — un préfixe de $a0-$a3 — et *tue* tous les registres "caller-save" — à savoir $a0-$a3, $v0, $ra, $t0-$t9;
- l'instruction IReturn *engendre* non seulement $ra, mais tous les registres "callee-save" — à savoir $s0-$s7 — et éventuellement le registre physique dédié au renvoi de résultat — à savoir $v0.

Notons que IReturn ne tue personne — de toute façon, elle n'a pas de successeur.

Communication entre procédures

L'instruction ITailCall combine certains aspects des instructions ICall et IReturn :

- comme ICall, elle *engendre* les registres physiques dédiés au passage d'arguments — un préfixe de \$a0-\$a3 ;
- comme IReturn, elle *engendre* \$ra ainsi que tous les registres "callee-save".

À titre d'épreuve, retrouvez ces règles pendant le TD sans consulter le cours !

Analyse de durée de vie

Calculs de point fixe

Graphe d'interférences

Élimination du code mort

Analyse de flot de données

L'analyse de durée de vie est membre de la famille des *analyses de flot de données*. Ces analyses associent une *propriété* à chaque point du code.

En général, les propriétés sont *ordonnées ;* un système *d'inéquations* définit un ensemble de solutions *sûres ;* parmi les solutions sûres, la *plus petite* est celle recherchée.

Cette théorie classique date des années 1970. C'est un cas particulier d'une théorie plus générale, datant de la même époque et beaucoup développée depuis, intitulée "interprétation abstraite" (Cousot, 2000).

Exemples

Voici un échantillon des propriétés étudiées dans la littérature :

- quelles variables seront *potentiellement utilisées ?*
- quelles variables ont *une valeur connue* et laquelle ?
- quelles variables ont une valeur *appartenant à un intervalle connu* et lequel ?
- quelles sont les *relations affines connues* entre variables ?
- quelles variables sont *certainement égales ?*
- quelles variables sont *potentiellement égales ?*
- quelles expressions *seront certainement évaluées ?*
- quels points du code *ont certainement été atteints* auparavant ?
- ...

Le treillis des propriétés

L'ensemble $\mathcal{P}$ des propriétés doit être un *sup-demi-treillis :* il doit jouir

- d'un *ordre* $\sqsubseteq$;
- d'un élément *minimum* $\perp$ ("bottom") ;
- d'une opération de *plus petite borne supérieure* ("join") $\sqcup$.

On exige de plus que *toute suite croissante soit ultimement stationnaire*, de façon à garantir la terminaison de l'analyse.

Ces conditions impliquent en fait que $\mathcal{P}$ est un *treillis complet*.

Pour l'analyse de durée de vie, $\mathcal{P} = (2^{\mathcal{V}}, \subseteq, \emptyset, \cup)$.

Inéquations

Les inéquations qui définissent l'analyse de durée de vie peuvent s'écrire :

$$\begin{array}{rcl} \bigsqcup_{\ell \to \ell'} \mathrm{vivantes}_{\mathrm{entrée}}(\ell') & \sqsubseteq & \mathrm{vivantes}_{\mathrm{sortie}}(\ell) \\ \mathrm{transfert}(\ell)(\mathrm{vivantes}_{\mathrm{sortie}}(\ell)) & \sqsubseteq & \mathrm{vivantes}_{\mathrm{entrée}}(\ell) \end{array}$$

où la *fonction de transfert* transfert(ℓ), une fonction de $\mathcal{P}$ dans $\mathcal{P}$, est donnée par :

$$\mathrm{transfert}(\ell)(p) = (p \setminus \mathrm{tuées}(\ell)) \cup \mathrm{engendrées}(\ell)$$

Fonctions de transfert

En général, toute fonction de transfert f doit être *monotone*, ce que l'on peut écrire de deux façons équivalentes :

$$p_1 \sqsubseteq p_2 \Rightarrow f(p_1) \sqsubseteq f(p_2)$$
$$f(p_1 \sqcup p_2) \sqsupseteq f(p_1) \sqcup f(p_2)$$

Cette condition signifie qu'une meilleure information à l'entrée d'une instruction doit donner une meilleure information à la sortie.

On n'exige pas en général la *distributivité :*

$$f(p_1 \sqcup p_2) = f(p_1) \sqcup f(p_2)$$

Dans le cas de l'analyse de durée de vie, la fonction de transfert *est* distributive.

Vers une inéquation unique

Soit $\mathcal{L}$ l'ensemble des étiquettes du graphe de flot de contrôle.

Le système d'inéquations peut s'écrire sous la forme :

$$F(X) \sqsubseteq X$$

où X est un vecteur de $2\mathcal{L}$ inconnues à valeurs dans le treillis $\mathcal{P}$, ou bien (c'est équivalent) une inconnue à valeurs dans le treillis $\mathcal{L} \longrightarrow \mathcal{P}^2$, et où la fonction F est monotone.

Vers une équation au point fixe

Théorème (Tarski). Soit F une fonction monotone d'un treillis complet vers lui-même. Alors l'équation $F(X) = X$ admet une plus petite solution, appelée *plus petit point fixe* de F et notée $\mathrm{lfp}\, F$. De plus, celle-ci coïncide avec la plus petite solution de l'inéquation $F(X) \sqsubseteq X$.

La démonstration, facile, est laissée en exercice.

Calcul par approximations successives

Le plus petit point fixe de F peut être calculé par *approximations inférieures successives* :

Théorème. Soit F une fonction monotone d'un treillis complet vers lui-même. Alors,

$$\mathrm{lfp}\, F = \lim_{n \to \infty} F^n(\bot)$$

La démonstration est de nouveau laissée en exercice.

Un algorithme moins naïf

En pratique, (pour l'analyse de durée de vie,) il n'est utile de réévaluer la propriété au point ℓ que lorsque la propriété associée à l'un des *successeurs* de ℓ a évolué.

Un algorithme moins naïf

D'où un meilleur algorithme :

> insérer tous les points ℓ dans un ensemble de travail
> tant que l'ensemble de travail n'est pas vide
>   en retirer un point ℓ
>   *réévaluer* la propriété au point ℓ
>   *si elle a cru strictement,*
>   ajouter les prédécesseurs de ℓ à l'ensemble de travail

Un algorithme moins naïf

On peut implémenter cet algorithme de calcul de point fixe de façon *générique*, *efficace*, *paresseuse*, avec *découverte automatique* du graphe de dépendances.

C'est joli ! Voir le module Fix du petit compilateur et mon article "Lazy least fixed points in ML".

Analyse de durée de vie

Calculs de point fixe

Graphe d'interférences

Élimination du code mort

Interférence

On pourrait poser que deux variables distinctes *interfèrent* si elles sont toutes deux *vivantes en un même point*. Cette définition ne serait *pas correcte*... (Pourquoi ?)

En fait, il faut poser que deux variables distinctes interfèrent si *l'une est vivante à la sortie d'une instruction qui définit l'autre.*

Deux variables qui n'interfèrent pas *peuvent* être réalisées par un *unique* emplacement — registre physique ou emplacement de pile.

Inversement, deux variables qui interfèrent *doivent* être réalisées par deux emplacements distincts.

Une exception

Supposons x vivante à la sortie d'une instruction qui définit y. Si *la valeur reçue par y est certainement celle de x*, alors il n'y a pas lieu de considérer que les deux variables interfèrent.

Cette propriété est en général indécidable, mais il en existe *un cas particulier simple*, celui d'une instruction **move** y, x.

Ce cas particulier est important car, dans ce cas, on *souhaite* justement que x et y soient réalisés par le même emplacement, de façon à supprimer l'instruction **move**.

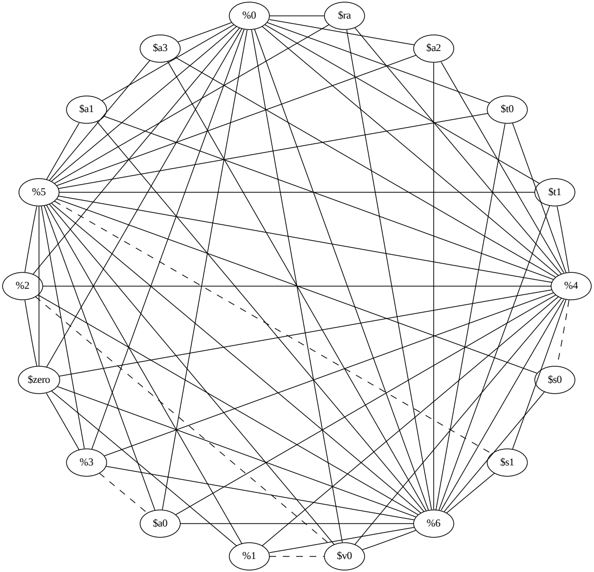
Le graphe d'interférences

On construit un *graphe* dont les sommets sont les variables et dont les arêtes représentent les relations *d'interférence* et de *préférence*.

On crée une arête d'interférence entre deux variables qui interfèrent. On crée une arête de préférence entre deux variables reliées par une instruction **move**.

Ce graphe permet de *spécifier* à l'allocateur de registres les contraintes sous lesquelles il doit travailler.

Voici le graphe d'interférences correspondant à la fonction factorielle.



Les arêtes de préférence sont en pointillés.

Analyse de durée de vie

Calculs de point fixe

Graphe d'interférences

Élimination du code mort

Élimination du code mort

Une instruction *pure* dont la variable de destination est *morte* à la sortie de l'instruction est dite *éliminable* et peut être supprimée.

Une instruction est *pure* si elle n'a pas d'effet autre que de modifier sa variable de destination. Par exemple, IConst, IUnOp, IBinOp sont pures ; ICall *n'est pas* pure.

L'élimination des sous-expressions communes (CSE) étudiée lors du cinquième cours peut laisser derrière elle des instructions pures éliminables, qu'il est donc intéressant de détecter et supprimer.

Élimination des initialisations superflues

Lors du passage de PP à UPP, des instructions sont *insérées* au début de chaque procédure pour *initialiser* chaque variable locale à 0, même celle-ci est explicitement initialisée plus loin.

Si une de ces instructions est superflue, son pseudo-registre destination est *mort*. L'instruction est alors *supprimée* après l'analyse de durée de vie, lors du passage de ERTL à LTL.

On obtient ainsi sans difficulté un code *efficace* et qui néanmoins *respecte* la sémantique de PP selon laquelle les variables sont implicitement initialisées à 0.

Raffinement de l'analyse de durée de vie

On peut prendre en compte la notion d'instruction éliminable pour améliorer l'analyse de durée de vie.

Il suffit de considérer qu'une instruction *ne définit ni n'engendre aucune variable* si, *d'après la valuation courante*, elle est éliminable.