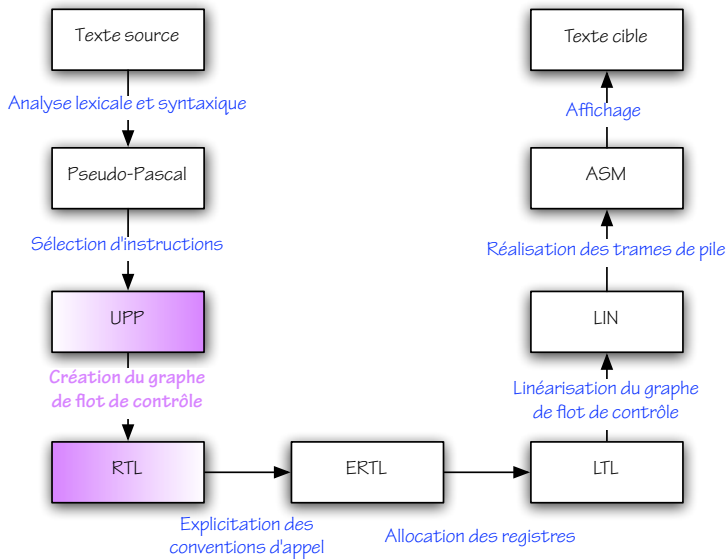


Compilation (INF 564)

Création du graphe de flot de contrôle: de UPP à RTL
Élimination des sous-expressions communes: de RTL vers lui-même

François Pottier

13 janvier 2016



Présentation de RTL

De UPP à RTL par l'exemple

Détails pratiques

Alternatives aux CFG

Identification des appels terminaux

Élimination des sous-expressions communes

Register Transfer Language (RTL)

Dans RTL,

- ▶ expressions et instructions structurées sont *décomposées* en *instructions élémentaires* organisées en *graphe de flot de contrôle* (“control-flow graph”, CFG);
- ▶ les variables locales sont remplacées par des *pseudo-registres*.

Register Transfer Language (RTL)

Voici ce qui justifie ces choix :

- ▶ L'organisation en graphe facilite *l'insertion* ou la *suppression* d'instructions par les phases d'optimisation ultérieures.
- ▶ Elle est *simple* et *générale* : elle peut refléter les constructions **while**, **repeat**, **for**, **if**, **case**, **break**, **continue**, et même **goto**.
- ▶ La *structure arborescente* des expressions, exploitée lors de la sélection d'instructions, ne sera plus utile au-delà.
- ▶ Pour ne pas trop compliquer les choses, les pseudo-registres de RTL sont *en nombre illimité* et *locaux* à chaque fonction, donc *préservés lors des appels*. Le fait que les registres physiques sont *en nombre fini* et sont *partagés* par toutes les fonctions sera traité ultérieurement.

Register Transfer Language (RTL)

Voici une traduction de la fonction factorielle dans RTL :

```
function f(%0) : %1
var %0, %1, %2, %3
entry f6
exit f0
f6: li    %1, 0      → f5
f5: blez %0         → f4, f3
f3: addiu %3, %0, -1 → f2
f2: call  %2, f(%3) → f1
f1: mul   %1, %0, %2 → f0
f4: li    %1, 1      → f0
```

Paramètre, *résultat*, *variables locales* sont des pseudo-registres. Le graphe est donné par ses *labels d'entrée* et de *sortie* et par une table qui à chaque label associe une instruction. Chaque instruction mentionne explicitement le ou les labels de ses *successeurs*.

Présentation de RTL

De UPP à RTL par l'exemple

Détails pratiques

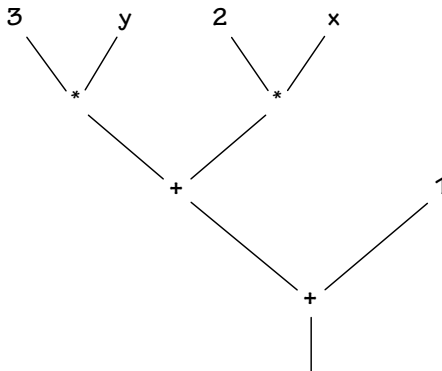
Alternatives aux CFG

Identification des appels terminaux

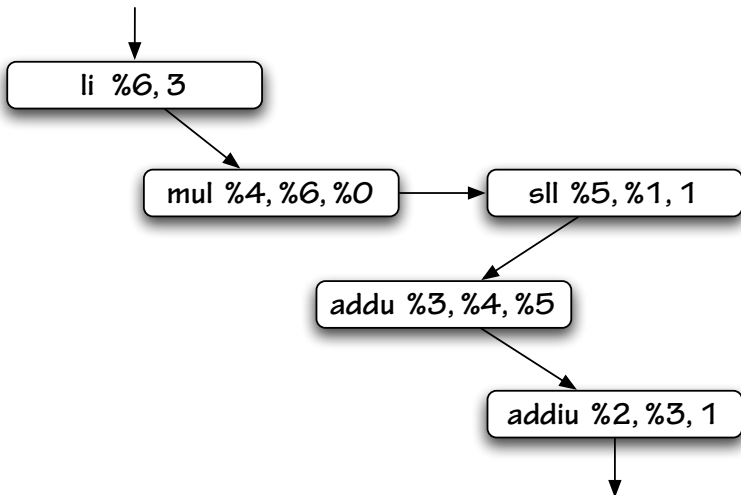
Élimination des sous-expressions communes

Traduction d'une expression

Voici l'arbre de syntaxe abstraite d'une expression UPP :



Voici sa traduction dans RTL :



Traduction d'une expression

L'exemple précédent illustre plusieurs points :

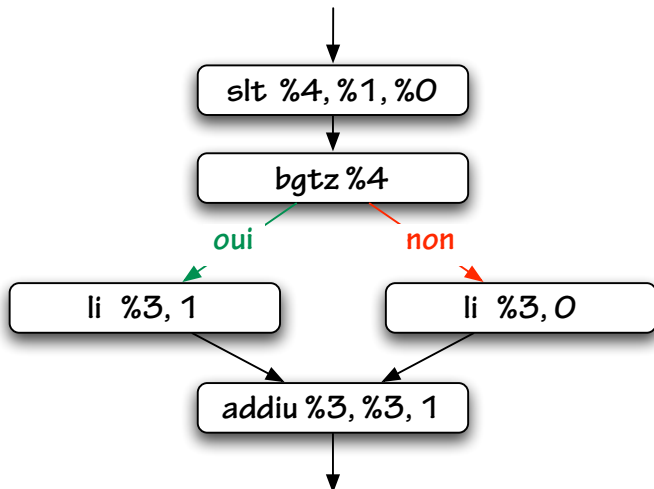
- ▶ Un *environnement* est nécessaire pour mémoriser le fait que x devient $\%1$, y devient $\%0$, etc.
- ▶ Un pseudo-registre *frais* reçoit le résultat de chaque sous-expression.
- ▶ Chaque (sous-)expression est traduite par un *fragment* de graphe doté d'un *label d'entrée*, un *label de sortie*, et un *pseudo-registre destination* distingués.
- ▶ Les fragments de graphe correspondant aux différentes sous-expressions sont *reliés* les uns aux autres d'une façon qui reflète *l'ordre d'évaluation* imposé par la sémantique de PP et UPP.

Traduction d'une conditionnelle

Voici une construction conditionnelle exprimée dans UPP :

```
if  $x < y$  then  
     $z := 1$   
else  
     $z := 0$ ;  
 $z := z + 1$ 
```

Voici sa traduction dans RTL :



Traduction d'une conditionnelle

L'exemple précédent illustre plusieurs points :

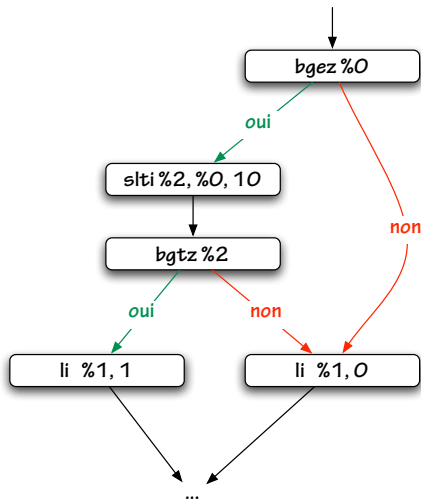
- ▶ La traduction *la plus simple* de la conditionnelle consiste à évaluer la condition vers un pseudo-registre, qui contient alors 0 ou 1, puis à utiliser (par exemple) l'instruction **bgtz**.
- ▶ Les deux branches *se rejoignent* à l'issue de la conditionnelle. On voit apparaître une structure de *graphe* acyclique et non simplement de liste.
- ▶ Chaque instruction est traduite par un *fragment* de graphe doté d'un *label d'entrée* et d'un *label de sortie* distingués.

Traduction d'une conditionnelle plus complexe

Voici une conditionnelle plus complexe :

```
if  $x \geq 0$  and  $x \leq 9$  then  
    chiffre := true  
else  
    chiffre := false
```

Voici sa traduction dans RTL :



Traduction d'une conditionnelle plus complexe

L'exemple précédent illustre plusieurs points :

- ▶ Une conditionnelle peut parfois être traduite *sans évaluer explicitement* la condition : c'est ce que permettent les instructions spécialisées **bgez**, **bgtz**, **blez**, **bltz**, **ble**, **bne**.
- ▶ Si le test $x \geq 0$ échoue, on n'effectue pas le test $x \leq 9$, ce qui reflète le comportement "*court-circuit*" du **and** imposé par la sémantique de PP et UPP.

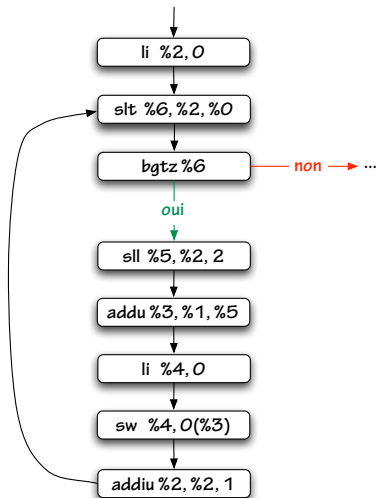
Traduction d'une boucle

Voici enfin une boucle :

```
i := 0;  
while i < n do begin  
    t[i] := 0;  
    i := i + 1  
end
```

Sa traduction est bien sûr un graphe *cyclique*...

Voici sa traduction dans RTL :



Présentation de RTL

De UPP à RTL par l'exemple

Détails pratiques

Alternatives aux CFG

Identification des appels terminaux

Élimination des sous-expressions communes

Syntaxe abstraite de RTL

Voici le jeu d'instructions de RTL :

```
type instruction =  
  | IConst of Register.t * int32 * Label.t  
  | IUnOp of unop * Register.t * Register.t * Label.t  
  | ...  
  | IUnBranch of uncon * Register.t * Label.t * Label.t  
  | IBinBranch of bincon * Register.t * Register.t * Label.t * Label.t
```

Chaque instruction mentionne explicitement ses *successeurs*.

De UPP à RTL

La traduction de UPP à RTL est confiée à deux modules :

- ▶ *Upp2rtl* implémente la traduction des expressions, conditions, et instructions, en supposant données quelques fonctions d'allocation de pseudo-registres et de labels;
- ▶ *Upp2rtl* fournit ces quelques fonctions ainsi que la traduction des procédures et programmes.

Interface de Upp2rtl

Voici l'interface `upp2rtl.mli` :

```
module Make (Env : sig
  val lookup: string → Register.t
  val allocate: unit → Register.t
  val generate: RTL.instruction → Label.t
  val loop: (Label.t → Label.t) → Label.t
  val is_exit: Label.t → bool
  val result: string option
end) : sig
  val translate_instruction: UPP.instruction → Label.t → Label.t
end
```

Upp2rtl.Make est un *foncteur* ou *module paramétré*.

Ce qu'attend Upp2rtl.Make

Ce foncteur exige qu'on lui fournisse d'abord :

- ▶ Une fonction *lookup* associant à chaque variable locale un pseudo-registre particulier. (Elle encapsule un *environnement*.)
- ▶ Une fonction *allocate* produisant un pseudo-registre *frais* à chaque appel. (Elle encapsule une *référence* vers un ensemble croissant de pseudo-registres.)

Ce qu'attend Upp2rtl.Make

Ce foncteur exige de plus :

- ▶ Une fonction *generate* qui, étant donnée une instruction, ajoute au graphe de flot de contrôle un nouveau sommet, portant cette instruction. (Elle encapsule une *référence* vers le graphe en cours de construction.)

La spécification de *generate* conduit à construire le graphe *d'arrière en avant*.

Quels graphes peut-on construire avec *generate* seule ?

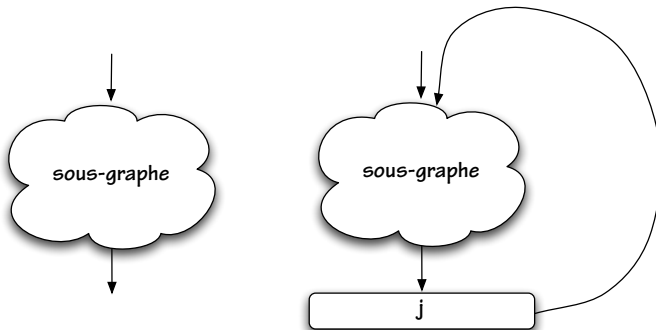
Ce qu'attend Upp2rtl.Make

Ce foncteur exige ensuite :

- ▶ Une fonction *loop* qui, étant donné un *sous-graphe* doté d'un *point d'entrée* et d'un *point de sortie* distingués, construit une *boucle* autour de ce sous-graphe.

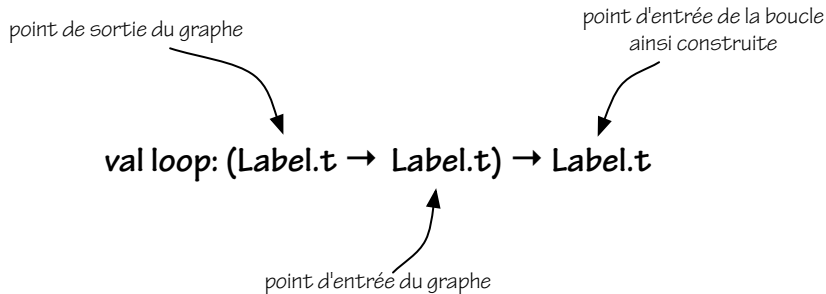
La fonction loop

Graphiquement, voici l'effet d'un appel à *loop* :



La fonction loop

Voici le type de *loop* :



La fonction loop

loop peut sembler magique, mais est en fait très simple :

```
let loop (subgraph : Label.t → Label.t) : Label.t =  
  let exit = Label.fresh luniverse in  
  let entry = subgraph exit in  
  graph := Label.Map.add exit (RTL.IGoto entry) !graph;  
  entry
```

Elle est définie dans *Upp2rtl*.

Ce qu'attend Upp2rtl.Make

Ce foncteur exige enfin :

- ▶ une fonction *is_exit* qui permet de reconnaître le label de sortie du graphe;
- ▶ une valeur *result* qui indique si nous traduisons une procédure ou une fonction, et dans le second cas, quel est le nom de cette fonction.

Ces informations servent à l'identification des appels terminaux.

Ce que fournit Upp2rtl.Make

Ce foncteur propose une fonction de traduction des instructions :

```
val translate_instruction: UPP.instruction → Label.t → Label.t
```

Étant donnée une *instruction* i et un *label* l , elle construit un sous-graphe dont la sémantique correspond à i , dont le *point de sortie* est l , et dont elle renvoie le *point d'entrée*.

Il s'agit toujours d'un schéma de construction *d'arrière en avant*.

Ce que fournit Upp2rtl.Make

De façon interne, ce foncteur construit également une fonction de traduction des conditions, où le sous-graphe construit a *deux* points de sortie :

```
val translate_condition: UPP.condition → Label.t → Label.t → Label.t
```

et une fonction de traduction des expressions, à laquelle on doit indiquer dans quel pseudo-registre stocker le *résultat* de l'expression :

```
val translate_expression: Register.t → UPP.expression → Label.t → Label.t
```

Présentation de RTL

De UPP à RTL par l'exemple

Détails pratiques

Alternatives aux CFG

Identification des appels terminaux

Élimination des sous-expressions communes

Limitations des CFG

Un graphe de contrôle est un objet *simple*, mais *peu structuré* :

- ▶ sa *structure* de graphe peut être arbitraire;
- ▶ les *dépendances* entre écritures et lectures (des pseudo-registres) n'y sont pas explicites.

Cela peut compliquer les analyses et transformations.

La forme SSA

Des structures plus avancées ont été proposées.

Par exemple, un graphe de flot de contrôle *sous forme SSA* garantit que chaque pseudo-registre *reçoit* une valeur en un *unique* sommet, qui *domine* tous les sommets où ce pseudo-registre est *utilisé*.

La forme SSA

Par exemple,

au lieu d'autoriser <i>ceci</i> ,	on exige <i>cela</i> :
$a \leftarrow x + y$	$a_1 \leftarrow x + y$
$b \leftarrow a - 1$	$b_1 \leftarrow a_1 - 1$
$a \leftarrow y + b$	$a_2 \leftarrow y + b_1$
$b \leftarrow x \times 4$	$b_2 \leftarrow x \times 4$
$a \leftarrow a + b$	$a_3 \leftarrow a_2 + b_2$
...	...

(a , b , a_1 , b_1 , sont des variables locales, ou pseudo-registres.)

Cela vous rappelle-t-il quelque chose? ...

La forme SSA est un langage fonctionnel

En fait, cela revient à choisir un langage intermédiaire doté de *variables non modifiables* :

au lieu d'autoriser ceci,	on exige cela :
$a \leftarrow x + y$	let $a_1 = x + y$ in
$b \leftarrow a - 1$	let $b_1 = a_1 - 1$ in
$a \leftarrow y + b$	let $a_2 = y + b_1$ in
$b \leftarrow x \times 4$	let $b_2 = x \times 4$ in
$a \leftarrow a + b$	let $a_3 = a_2 + b_2$ in
...	...

Voir “[SSA is Functional Programming](#)”, d’Appel – 4 pages *lisibles!* – ou bien le chapitre 19 du livre d’Appel.

Un langage intermédiaire fonctionnel

Et s'il y a des branchements et/ou des boucles?

```
x ← 0
```

```
y ← 2
```

```
while x ≤ 10 do
```

```
  x ← x + y
```

```
writeln(x)
```

Un langage intermédiaire fonctionnel

Et s'il y a des branchements et/ou des boucles ?

```
x ← 0
y ← 2

while x ≤ 10 do
  x ← x + y

writeln(x)

let x1 = 0 in
let y1 = 2 in
let rec loop(x2) =
  if x2 ≤ 10 then
    let x3 = x2 + y1 in
    loop(x3)
  else
    writeln(x2)
in loop(x1)
```

Cela vous rappelle-t-il quelque chose ? ...

Un langage intermédiaire fonctionnel

Dans ce dernier exemple, le langage intermédiaire a conservé une *structure arborescente*, avec

- ▶ des variables non modifiables,
- ▶ des fonctions (récurrentes) locales, parfois appelées *continuations*.

Définir une continuation “loop” revient à *étiqueter* un point du code, et appeler cette continuation revient à *sauter* vers cette étiquette.

Voir par exemple “*Compiling with continuations, continued*”, de Kennedy, pour un exemple réaliste.

Présentation de RTL

De UPP à RTL par l'exemple

Détails pratiques

Alternatives aux CFG

Identification des appels terminaux

Élimination des sous-expressions communes

Identification des appels terminaux

Le langage RTL distingue deux instructions d'appel de procédure ou fonction :

- ▶ *ICall* est un appel ordinaire; lorsque l'appelé aura terminé, il nous rendra la main;
- ▶ *ITailCall* est un appel terminal; lorsque l'appelé aura terminé, il rendra la main non pas à nous mais à *notre propre appelant*.

On peut considérer ITailCall comme une optimisation de la séquence (ICall; IReturn).

Identification des appels terminaux

En termes du langage source (ici, UPP), un appel de g à h est dit *terminal* si cet appel constitue *la dernière opération* effectuée par g .

En particulier, dans le cas où g et h sont des fonctions, il faut que le résultat soit transmis, c'est-à-dire que le résultat de h devienne celui de g .

Identification des appels terminaux

L'appel récursif ci-dessous est-il terminal?

```
function f (n : integer) : integer;  
begin  
  if n <= 0 then  
    f := 1  
  else  
    f := n * f (n - 1)  
end;
```

Identification des appels terminaux

Et celui-ci?

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

Identification des appels terminaux

Lors de la traduction de UPP vers RTL, nous devons *identifier* les appels terminaux et les traduire à l'aide de ITailCall.

Il serait *correct* de toujours utiliser ICall et jamais ITailCall.

Cependant, nous verrons la semaine prochaine pourquoi ITailCall est *plus efficace* en temps et en espace.

Présentation de RTL

De UPP à RTL par l'exemple

Détails pratiques

Alternatives aux CFG

Identification des appels terminaux

Élimination des sous-expressions communes

Exposé du problème

L'élimination des “sous-expressions communes” vise à supprimer certains *calculs redondants*.

Considérons par exemple le fragment de code suivant :

```
x := t[i] ;  
t[i] := t[i-1] ;  
t[i-1] := x ;
```

Que vont produire les traductions de PP vers UPP puis RTL ?

Exposé du problème

Une traduction naïve calcule *quatre fois* $\$a0 + 4 \times \$a3$:

sll \$v0, \$a3, 2	addu \$v0, \$a0, \$v0
addu \$v0, \$a0, \$v0	lw \$v0, -4(\$v0)
lw \$a2, 0(\$v0)	sw \$v0, 0(\$a1)
sll \$v0, \$a3, 2	sll \$v0, \$a3, 2
addu \$a1, \$a0, \$v0	addu \$v0, \$a0, \$v0
sll \$v0, \$a3, 2	sw \$a2, -4(\$v0)

Ce calcul redondant est celui de l'adresse que l'on pourrait écrire, en C, sous la forme $t + i$.

En Pseudo-Pascal, le programmeur n'a *aucun moyen* de modifier le programme pour améliorer le code produit!

Exposé du problème

On préférerait obtenir ceci :

```
sll    $v0, $a2, 2
addu  $a1, $a0, $v0
lw    $a3, 0($a1)
lw    $v0, -4($a1)
sw    $v0, 0($a1)
sw    $a3, -4($a1)
```

La multiplication et l'addition ne sont effectuées qu'une fois et leur résultat, à savoir l'adresse \$a1, est utilisée quatre fois.

Comment détecter les calculs redondants ?

Plaçons-nous au niveau de RTL. L'idée est de *simuler l'exécution* du code en *mémorisant des relations* entre pseudo-registres :

	sll	%12,%2,2
%12 = sll(%2,2)	addu	%11,%1,%12
%11 = %1 + sll(%2,2)	lw	%3,0(%11)
	sll	%10,%2,2
%10 = sll(%2,2)	addu	%6,%1,%10
%6 = %1 + sll(%2,2)		

Dans un second temps, on *transformera* le code en utilisant cette information. (Comment?) Le résultat sera *un nouveau programme RTL*.

Attention

Mémoriser naïvement des relations entre pseudo-registres serait *incorrect* :

```
                                addiu %1,%0,1
%1 = %0 + 1                    addiu %2,%0,1
%2 = %0 + 1                    li      %0,0
!?
```

Après l'instruction **li**, les deux équations deviennent *fausses*!

On pourrait les oublier purement et simplement, mais alors on perdrait leur conséquence $\%1 = \%2$ qui, elle, *reste vraie*...

Une solution

Évitons de faire apparaître des objets *modifiables* – les pseudo-registres – au sein d’expressions mathématiques.

Utilisons des *variables symboliques* a, β, \dots et associons à chaque pseudo-registre, en chaque point du code, une *expression symbolique* :

%0 contient a	addiu	%1, %0, 1
%1 contient $a + 1$	addiu	%2, %0, 1
%2 contient $a + 1$	li	%0, 0
%0 contient 0		

L’assertion “%0 contient 0” *remplace* l’assertion “%0 contient a ” mais *n’invalide pas* les deux assertions intermédiaires. On peut *correctement* conclure que %1 et %2 contiennent la même valeur finale.

Variables et expressions symboliques

La syntaxe abstraite des expressions symboliques sera :

$$e ::= a \mid k \mid op\ e \mid e\ op\ e$$

On se donne des variables symboliques en nombre infini.

On se donne des *environnements* associant des expressions symboliques aux pseudo-registres, et on effectue une *exécution symbolique en avant*.

Prise en compte des branchements

Pour du code *linéaire*, (presque) tout est dit, mais comment analyser un *graphe* de flot de contrôle?

Si une instruction a plusieurs *successeurs*, il suffit de distribuer à tous deux l'environnement obtenu à la sortie de cette instruction.

Si une instruction a plusieurs *prédécesseurs*, que faire? Il faudrait *fusionner* les environnements issus de chacun des prédécesseurs, mais certains d'entre eux n'ont peut-être encore été *jamais atteints* par l'analyse (le graphe peut être cyclique).

Une solution simple

Une solution simple consiste à découper le graphe en *blocs de base étendus* disjoints et à examiner chacun indépendamment.

Un bloc de base étendu est un arbre (maximal) d'instructions où chaque instruction, hormis la racine, a *exactement un prédécesseur* dans le graphe — son parent dans l'arbre.

C'est l'approche adoptée dans le petit compilateur.

Des approches plus ambitieuses

On peut vouloir mieux faire et détecter les calculs redondants à l'échelle du graphe de flot de contrôle tout entier.

L'analyse devient alors significativement plus complexe.

Voir par exemple “[A Polynomial-Time Algorithm for Global Value Numbering](#)”, de Gulwani et Necula.

Transformation

Une fois l'exécution symbolique effectuée, la *transformation* du code est simple. Une instruction redondante, par exemple :

```
%1 contient a + 1      addiu %2,%0,1  
%2 contient a + 1
```

est transformée en :

```
%1 contient a + 1      move %2,%1  
%2 contient a + 1
```

Noter que cela suppose une notion *d'égalité* entre expressions symboliques. (Pourquoi? Comment la définir?)

Que deviendra cette instruction **move** dans les phases suivantes du compilateur?

Quelques points délicats (I)

Lorsque le résultat d'une instruction n'est pas exprimable à l'aide d'une expression symbolique, il est représenté par une variable symbolique *fraîche*.

C'est le cas par exemple des instructions **lw**, **getg** ou **call** de RTL :

```
%1 contient e      lw %1,4(%0)
%1 contient a
```

Quelques points délicats (II)

On peut ajouter les accès à la mémoire (*ELoad*) et aux variables globales (*EGetGlobal*) à la syntaxe des expressions symboliques.

Cela permet par exemple d'éliminer le second **lw** ci-dessous :

```
%0 contient a      lw  %1,0(%0)
%1 contient O(a)   lw  %2,0(%0)
%2 contient O(a)   sw  %3,0(%4)
!?                 lw  %5,0(%0)
```

Attention toutefois, les deux dernières assertions sont *invalidées* par l'instruction **sw**. La dernière instruction **lw** ne peut donc pas être éliminée, du moins pas sans une analyse *d'aliasing* plus poussée.

Quelques points délicats (III)

Réutiliser un résultat précédent *augmente la durée de vie* du pseudo-registre qui le contient, ce qui *complique* l'allocation de registres et peut mener à un code final *moins efficace*.

Si cette durée de vie vient traverser un appel de procédure, on devra typiquement utiliser un registre “callee-save” de plus.

Pour éviter cela, une approche prudente consiste à *oublier* toutes les informations amassées par l'exécution symbolique lorsque celle-ci traverse un appel de procédure.

Ne pas croire qu'une “optimisation” améliore toujours le code! *Mesurer les performances* et penser aux *interactions* entre transformations.