Compilation

Cours n°4: Création du graphe de flot de contrôle: de UPP à RTL
Élimination des sous-expressions communes: de RTL vers lui-même

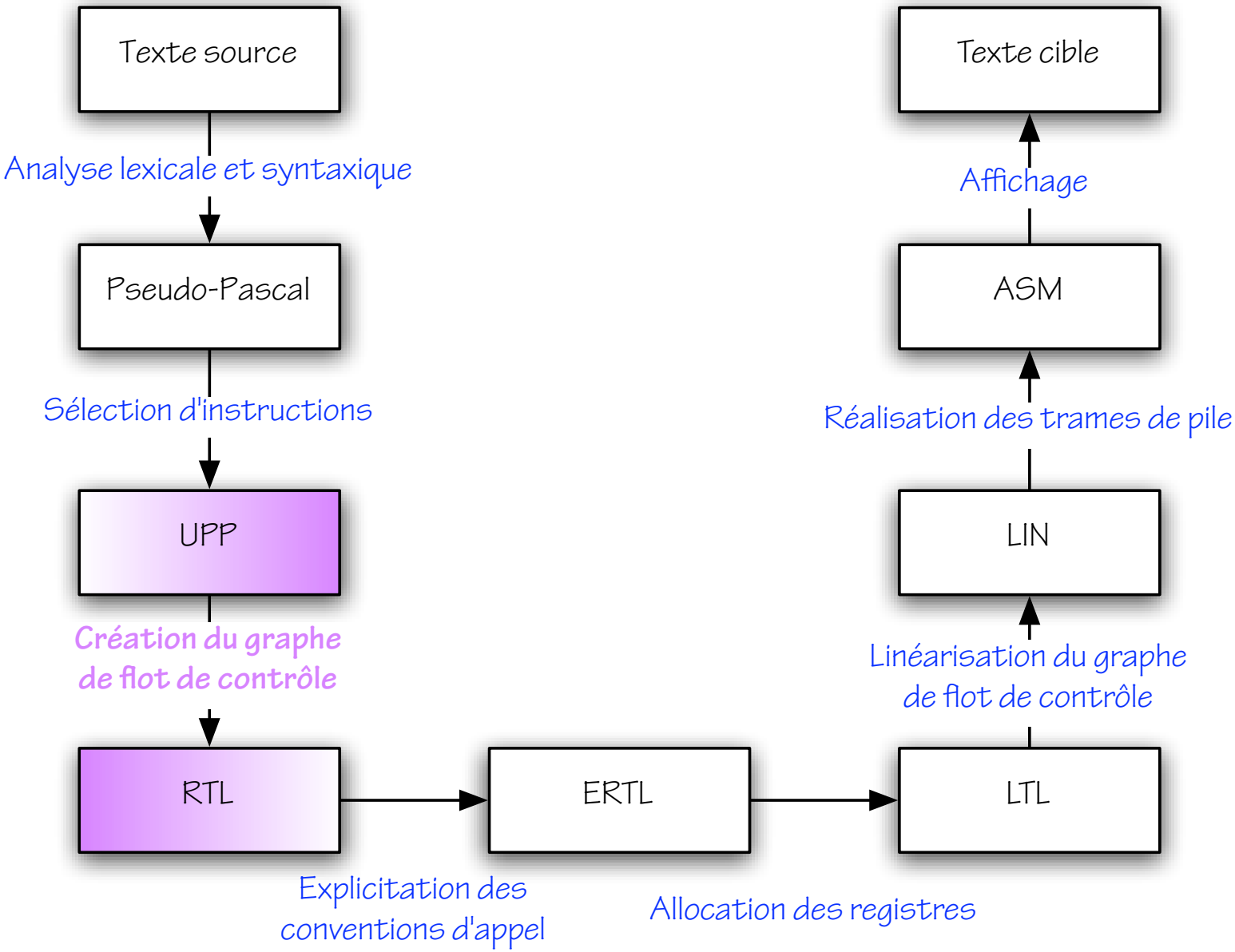
Sandrine Blazy
(d'après le cours de François Pottier)

ensiie - 2e année

17 novembre 2008

école nationale supérieure d'informatique pour l'industrie et l'entreprise

ensiie


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

ensiie

Register Transfer Language (RTL)

Dans RTL,

- expressions et instructions structurées sont *décomposées* en *instructions élémentaires* organisées en *graphe de flot de contrôle* ;
- les variables locales sont remplacées par des *pseudo-registres*.

ensiie

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.

ensiie

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
```

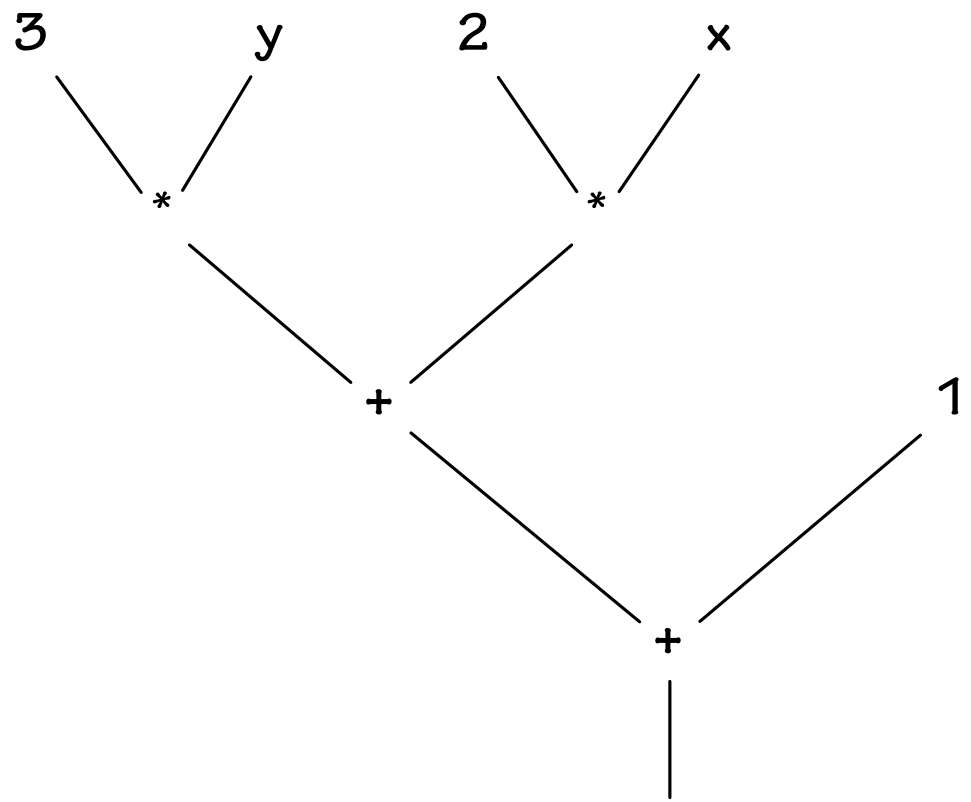
Paramètre, *résultat*, *variables locales* sont des pseudo-registres. Le graphe est donné par ses *labels d'entrée* et de *sortie* et par une table qui à chaque label associe une instruction. Chaque instruction mentionne explicitement le ou les labels de ses *successeurs*.

ensiie

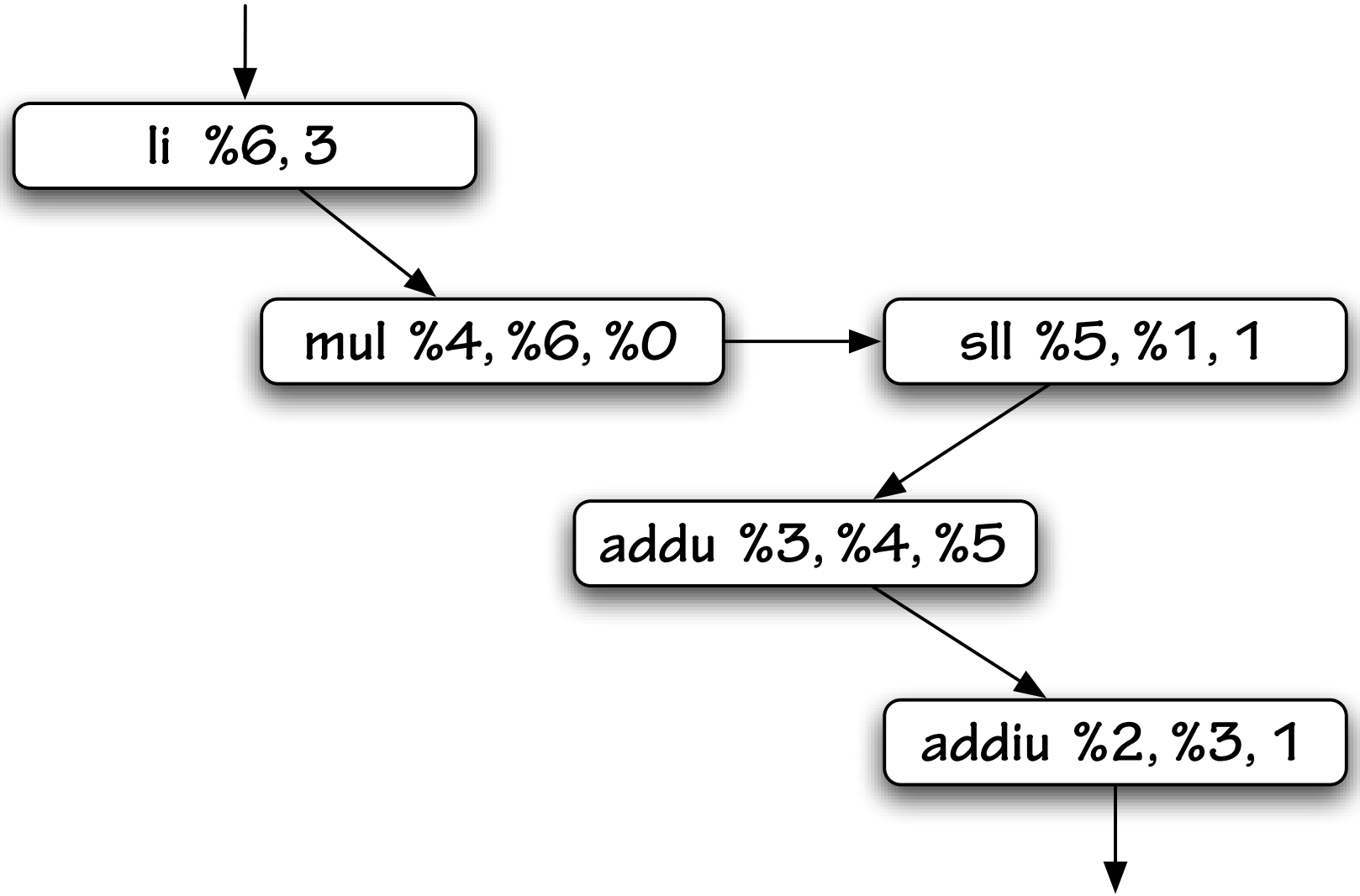
ensiie

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 %2, 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.

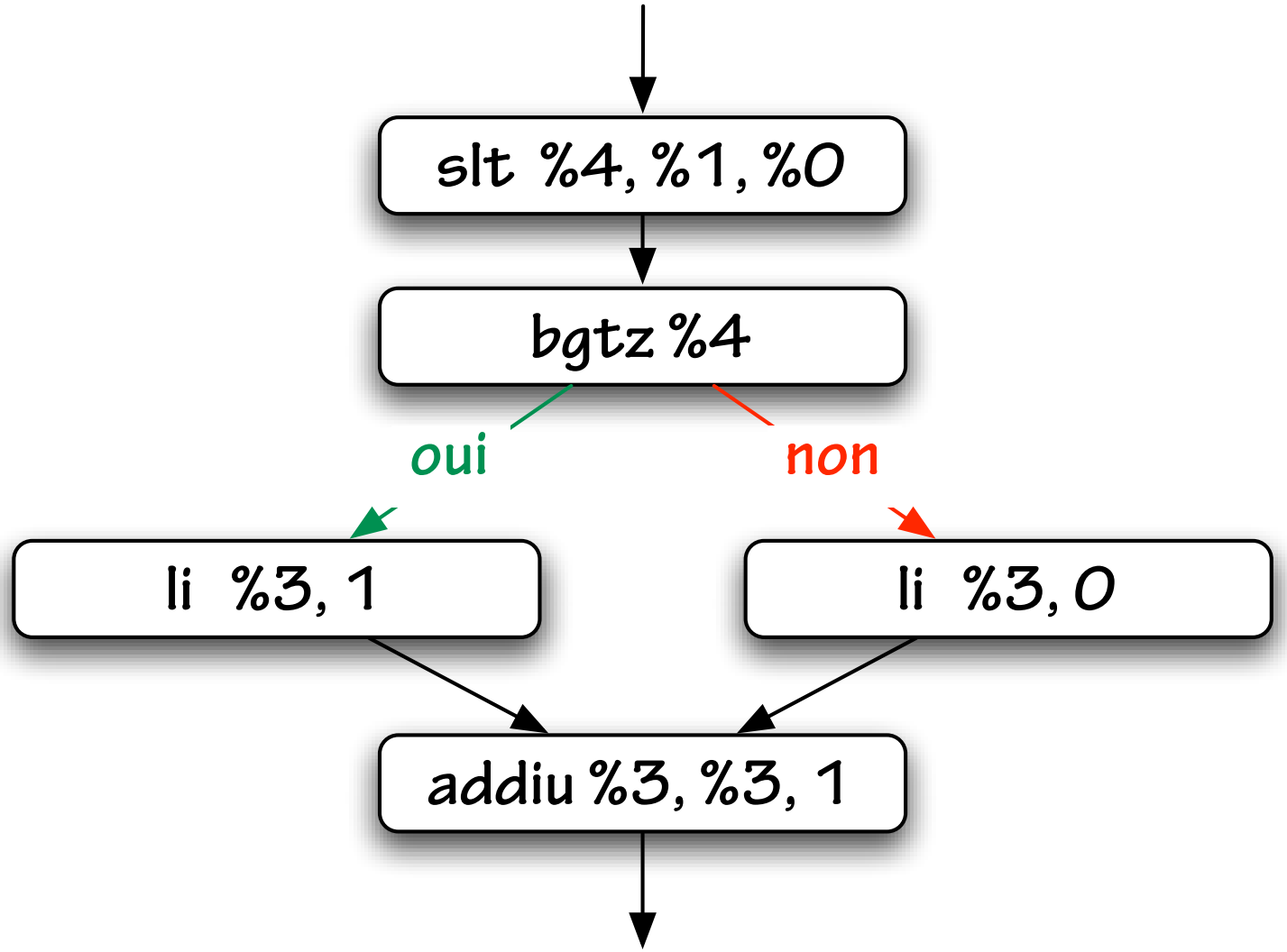
ensiie

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.

ensiie

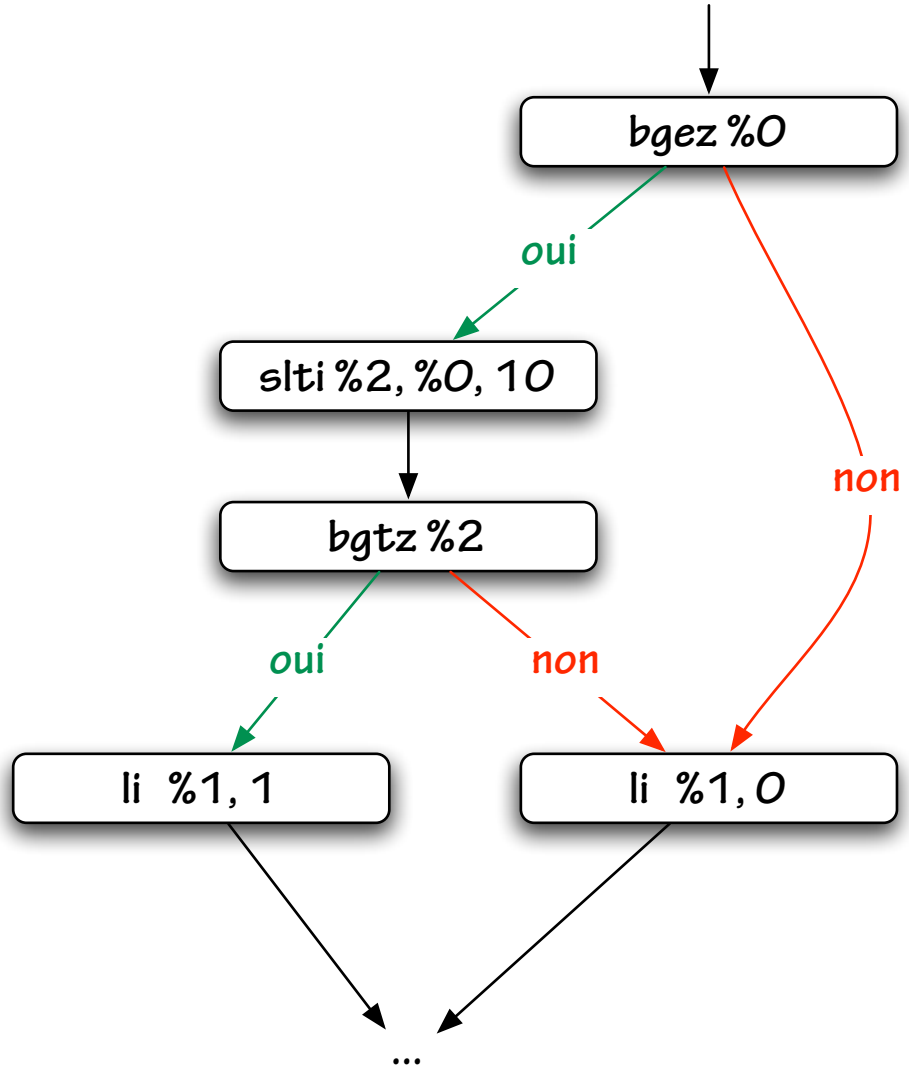
Traduction d'une conditionnelle plus complexe

Voici une conditionnelle plus complexe :

```
if x >= 0 and x <= 9 then
    chiffre := true
else
    chiffre := false
```

ensiie

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 >= 0 échoue, on n'effectue pas le test x <= 9, ce qui reflète le comportement « *court-circuit* » du **and** imposé par la sémantique de PP et UPP.

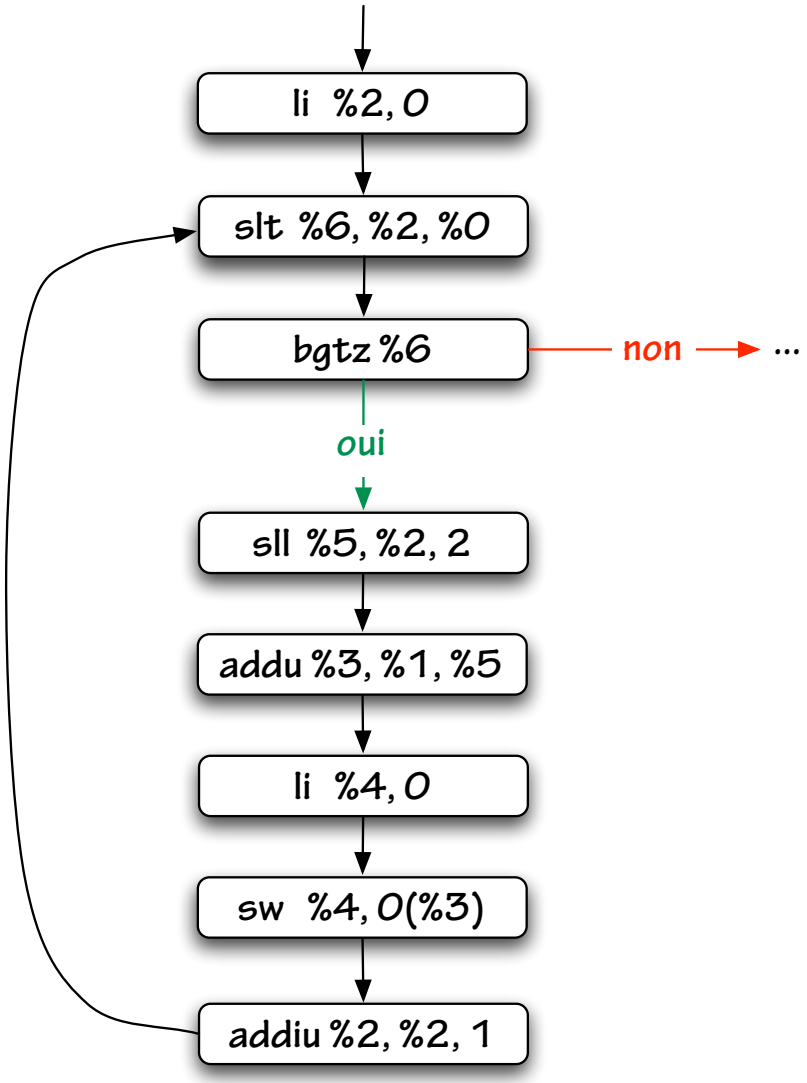
ensiie

Traduction d'une boucle

Voici enfin une boucle :

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

Voici sa traduction dans RTL :



Traduction d'une boucle

L'exemple précédent illustre le fait que les boucles rendent le graphe *cyclique*.

Toutefois, en l'absence de construction **goto** dans le langage source, les graphes obtenus restent *réductibles* : leurs boucles sont imbriquées de façon structurée.

Suit une petite digression...

ensiie

Une caractérisation de la réductibilité

Un graphe de flot de contrôle est *réductible* si et seulement si, dans tout cycle, il existe un sommet qui domine les autres.

Un sommet m *domine* un sommet n si et seulement si tout chemin du point d'entrée du graphe vers n passe par m.

En bref, *toute boucle admet un unique point d'entrée :* on ne peut sauter directement à l'intérieur.

Voir par exemple « A fast algorithm for finding dominators in a flow graph », de Lengauer et Tarjan, pour en savoir plus sur la notion de domination.

ensiie

Une seconde caractérisation de la réductibilité

Un graphe est *réductible* si et seulement si on peut le *réduire à un unique sommet* en répétant les deux transformations suivantes :

- fusionner un sommet avec son prédécesseur, *si celui-ci est unique ;*
- supprimer une *auto-boucle*, c'est-à-dire une arête d'un sommet vers lui-même.

Les graphes réductibles ont été étudiés dans les années 1970. Voir par exemple « Characterizations of reducible flow graphs », de Hecht et Ullmann.

ensiie

Blocs de base

De nombreux compilateurs appliquent la transformation suivante :

- fusionner les sommets m et n, si n est l'unique successeur de m et m l'unique prédécesseur de n.

Les sommets ainsi obtenus représentent alors non pas une instruction, mais une *séquence* d'instructions ayant la propriété que pour en exécuter une, il faut les exécuter toutes. De telles séquences (maximales) d'instructions sont appelées *blocs de base*.

Le graphe des blocs de base contient moins de sommets, ce qui conduit à des algorithmes plus efficaces en pratique.

ensiie

Au-delà du graphe de flot de contrôle

Certains compilateurs emploient des structures plus avancées qu'un simple graphe de flot de contrôle.

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*.

Voir « SSA is Functional Programming », d'Appel – 4 pages *lisibles !* – ou bien le chapitre 19 du livre d'Appel.

Un graphe sous forme SSA peut être représenté à l'aide d'un langage d'expressions *arborescentes* – voir « The anatomy of a loop », de Shivers, pour un exemple réaliste (et complexe).

ensiie

ensiie

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
  | IBinOp of binop * Register.t * Register.t * Register.t * Label.t
  | ICall of Register.t option * Primitive.callee * Register.t list * Label.t
  | ILoad of Register.t * Register.t * offset * Label.t
  | IStore of Register.t * offset * Register.t * Label.t
  | IGetGlobal of Register.t * offset * Label.t
  | ISetGlobal of offset * Register.t * Label.t
  | IGoto of 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*.

ensiie

Syntaxe abstraite de RTL

Et voici les informations associées à une procédure ou fonction :

```
and procedure = {
  formals: Register.t list;
  result: Register.t option;
  runiverse: Register.universe;
  locals: Register.Set.t;
  luniverse: Label.universe;
  entry: Label.t;
  exit: Label.t;
  graph: graph
}
and graph =
    instruction Label.Map.t
```

ensiie

Les modules Register et Label

Tous deux ont pour signature *AtomSig.S* :

```
module type S = sig
  type t
  type universe
  val new_universe: string → universe
  val fresh: universe → t
  val equal: t → t → bool
  val print: t → string
  module Set : ...
  module Map : ...
  module SetMap : ...
end
```

Noter que les types Register.t et Label.t sont *distincts*.

ensiie

De UPP à RTL

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

- *Upp2rtlI* 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.

ensiie

Interface de Upp2rtll

Voici l'interface *upp2rtll.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
end) : sig
  val translate_instruction: UPP.instruction → Label.t → Label.t
end
```

Upp2rtll.Make est un *foncteur* ou *module paramétré*.

Ce qu'attend Upp2rtll.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.)

ensiie

Ce qu'attend Upp2rtll.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 ?

ensiie

Ce qu'attend Upp2rtll.Make

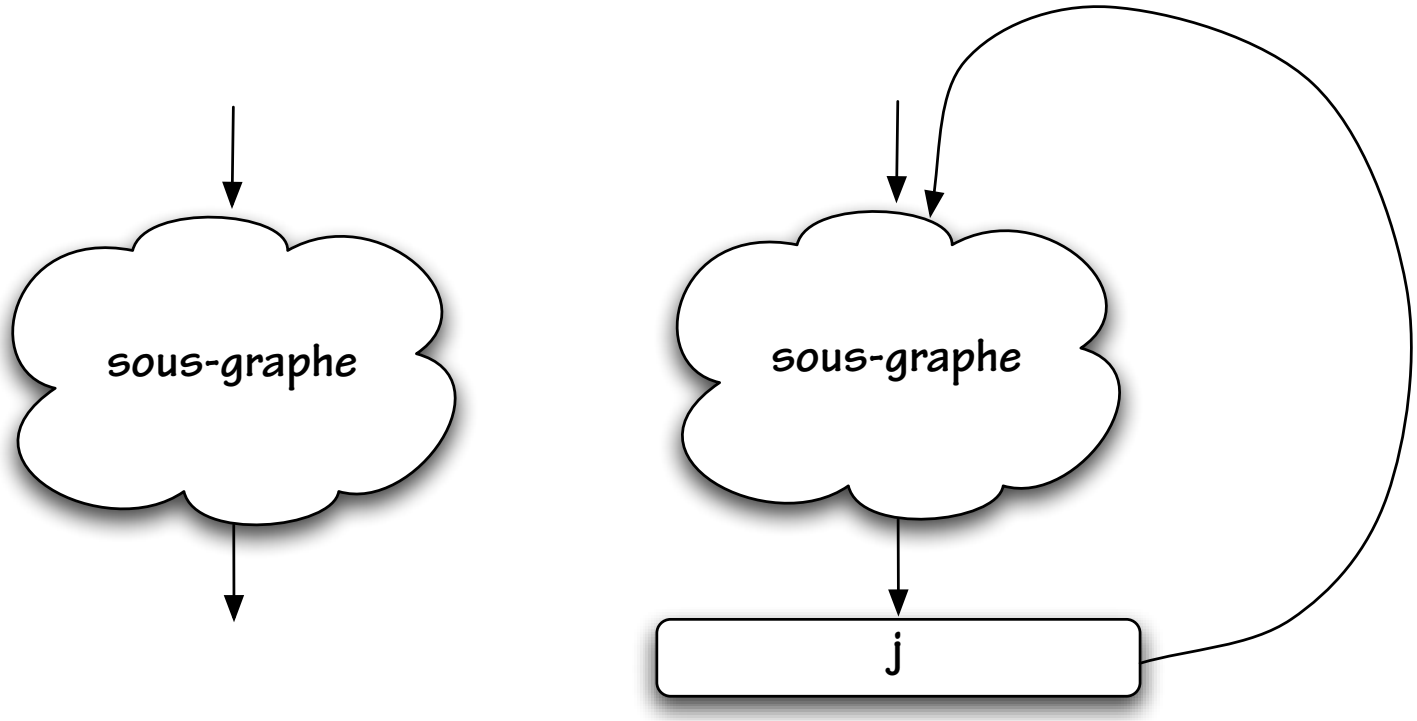
Ce foncteur exige enfin :

- 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.

ensiie

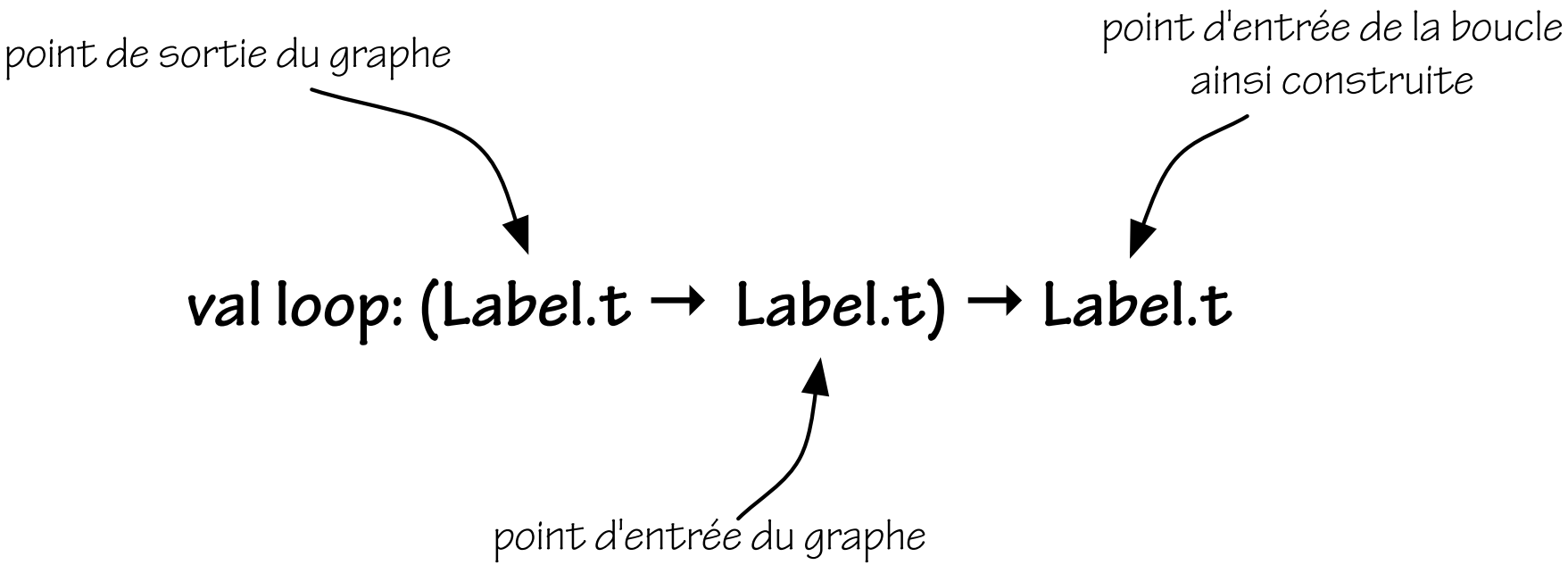
La fonction loop

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



La fonction loop

Voici le type de *loop :*



Ce que fournit Upp2rtlI.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.*

ensiie

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
```

ensiie

Présentation de RTL

De UPP à RTL par l'exemple

Détails pratiques

4 Élimination des sous-expressions communes

ensiie

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 ?

ensiie

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*.

ensiie

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.*

ensiie

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* $\alpha, \beta, \ldots$ et associons à chaque pseudo-registre, en chaque point du code, une *expression symbolique :*

%0 contient α	**addiu**	$\%1, \%0, 1$
%1 contient $\alpha + 1$	**addiu**	$\%2, \%0, 1$
%2 contient $\alpha + 1$	**li**	$\%0, 0$
%0 contient 0		

L'assertion « %0 contient 0 » *remplace* l'assertion « %0 contient α » mais *n'invalide pas* les deux assertions intermédiaires. On peut *correctement* conclure que %1 et %2 contiennent la même valeur finale.

ensiie

Variables et expressions symboliques

La syntaxe abstraite des expressions symboliques sera :

$$e ::= \alpha \mid k \mid \text{op}\ e \mid e\ \text{op}\ e$$

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).

ensiie

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.

ensiie

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.

La notion de *domination* et la *forme SSA* sont alors utiles. L'analyse devient significativement plus complexe.

Voir par exemple « Value numbering », de Briggs, Cooper, et Taylor Simpson.

ensiie

Transformation

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

%1 contient $\alpha + 1$	**addiu**	$\%2, \%0, 1$
%2 contient $\alpha + 1$		

est transformée en :

%1 contient $\alpha + 1$	**move**	$\%2, \%1$
%2 contient $\alpha + 1$		

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 ?

ensiie

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 α	

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 α	**lw**	$\%1, 0(\%0)$
%1 contient $0(\alpha)$	**lw**	$\%2, 0(\%0)$
%2 contient $0(\alpha)$	**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.

ensiie

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* et penser aux *interactions* entre transformations.

ensiie