

Optimisations de code indépendantes de la machine

Objectif

- Améliorer les performances du programme
- Mesures de performance :
 - Réduction du temps d'exécution
 - Réduction de la taille du code
 - Réduction de la consommation d'énergie

Contraintes

- **Préservation de la sémantique de programme**
- Effort minimal pour un gain maximal
- Une transformation doit produire un gain mesurable

Plan

1. Introduction
2. Représentation intermédiaire
3. Principales sources d'optimisation
4. Optimisations locales
5. Optimisation globales : Analyses de flot de données
6. Analyses de flot de contrôle

1. Introduction : place des optimisations

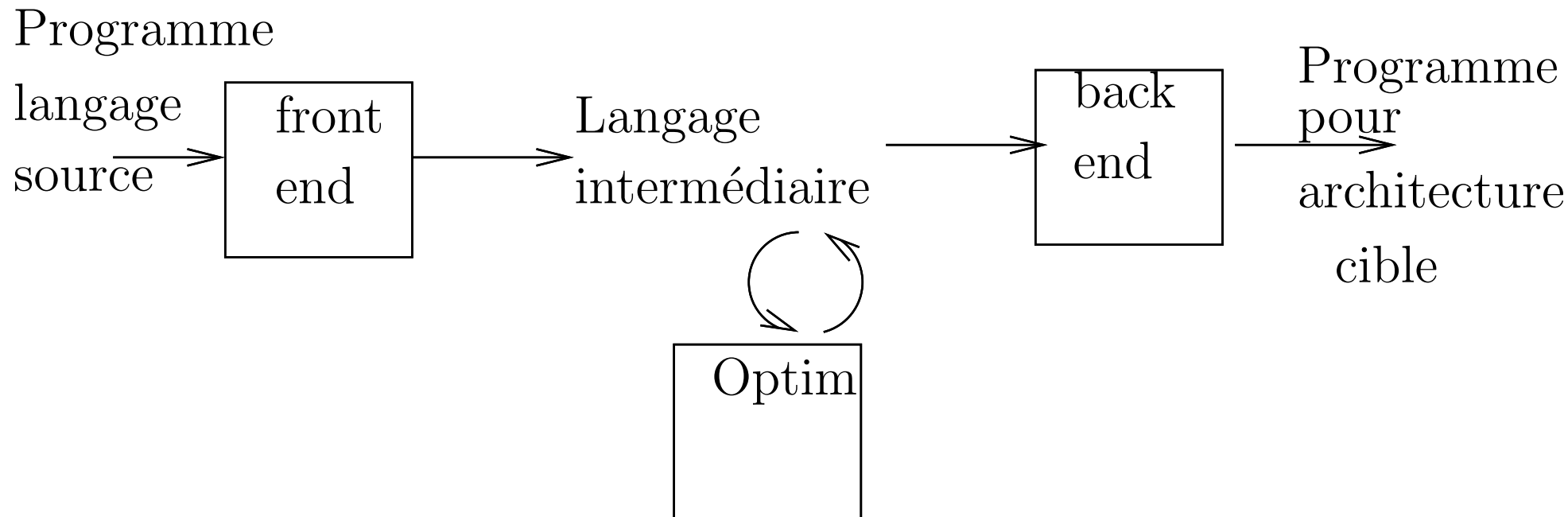
Places des optimisations

- Arbre abstrait ?
- Représentation intermédiaire (code 3 adresses) ?
- Assembleur ?

Classification

- locales
- globales intraprocédurales
- interprocédurales

1. Introduction : place des optimisations



Front end Analyses lexicales, syntaxiques et types

Back end Générateur de code et optimisations dépendantes de la machine.

1. Introduction : principales optimisations

- Sous expressions communes, expressions partiellement redondantes,
- Propagation des copies, des constantes
- Elimination du code inutile
- Optimisation des boucles, variables d'induction, invariants de boucle.
- Réduction de force, simplifications algébriques
- Déplacement de code

1. Introduction : Modules d'optimisation

Entrée code 3 adresses

Sortie code 3 adresses

Fonction

- Analyse du programme
- Transformation du programme

Analyse et transformation

Analyse	Transformation
Expressions disponibles	Elimination des sous-expressions communes
Variables actives	Eliminatio du code inutile
Invariants de boucle	Déplacement du code de l'invariant hors de la boucle
Variables d'induction	Réduction de force
Propagation de constantes	Suppression d'affectation de constante à des variables

2. Représentations intermédiaires

- Arbre abstrait
- Code à 3 adresses
- Graphes de flot de contrôle et blocs de bases
- Graphes de dépendance et graphes acycliques
- Forme SSA

Code à trois adresses intraprocédural

- $x := y \text{ op } x$
- $x := \text{op } y$
- $x := y$
- goto L
- si $x \text{ oprel } y$ goto L
- $*x := y$, $x := *y$, $x := \&y$
- $x[i] := y$, $x := y[i]$

Dans ce qui suit, on considère l'analyse intraprocédurale (sans appel de procédure).

Blocs de base et graphes de flot

Bloc de base : séquence d'instructions maximale dans laquelle le flot de contrôle est activé au début de celle-ci et inhibé à la fin de celle-ci.

Autrement dit, cette séquence

- n'a aucune étiquette, sauf sur la première instruction
- n'a aucune instruction de branchement, sauf la dernière instruction.

Flot de contrôle : déterminé par les branchements relatifs, absolus

Algorithme de partition en bloc de base

- On détermine les têtes de bloc (première instruction, instruction atteinte par une instruction de branchement ou toute instruction suivant un branchement).
- un bloc de base débute par une instruction de tête de bloc et se termine par l'instruction précédant l'instruction de tête de bloc suivante.

Graphe de flot de contrôle

Les nœuds du graphe sont les blocs de base et un arc relie le nœud n_1 au nœud n_2 si

- n_2 suit n_1 dans la séquence d'instructions,
- la dernière instruction de n_1 est un branchement vers la première instruction de n_2 .

3. Principales optimisations

- Sous expressions communes, expressions partiellement redondantes,
- Propagation des copies, des constantes
- Elimination du code inutile
- Optimisation des boucles, variables d'induction, invariants de boucle.
- Réduction de force, simplifications algébriques
- Déplacement de code

Optimisations locales

- Simplifications algébriques
- Evaluation de constantes ("constant folding")
- Sous expressions communes
- Réduction de force
- Propagation des copies
- Suppression de code inaccessible (ou code mort)

Exemple

Code initial

a := x ** 2

b := 3

c := x

d := c * c

e := b * 2

f := a + d

g := e * f

Exemple

Optimisation algébrique

<code>a := x ** 2</code>	<code>a := x * x</code>
<code>b := 3</code>	<code>b := 3</code>
<code>c := x</code>	<code>c := x</code>
<code>d := c * c</code>	<code>d := c * c</code>
<code>e := b * 2</code>	<code>e := b << 1</code>
<code>f := a + d</code>	<code>f := a + d</code>
<code>g := e * f</code>	<code>g := e * f</code>

Exemple

Propagation des copies

a := x * x	a := x * x
b := 3	b := 3
c := x	c := x
d := c * c	d := x * x
e := b << 1	e := 3 << 1
f := a + d	f := a + d
g := e * f	g := e * f

Exemple

“Constant folding”

a := x * x

b := 3

c := x

d := x * x

e := 3 << 1

f := a + d

g := e * f

a := x * x

b := 3

c := x

d := x * x

e := 6

f := a + d

g := e * f

Exemple

Elimination des sous expressions communes

a := x * x

b := 3

c := x

d := x * x

e := 6

f := a + d

g := e * f

a := x * x

b := 3

c := x

d := a

e := 6

f := a + d

g := e * f

Exemple

Propagation des copies

a := x * x

b := 3

c := x

d := a

e := 6

f := a + d

g := e * f

a := x * x

b := 3

c := x

d := a

e := 6

f := a + a

g := 6 * f

Exemple

Elimination de code mort

a := x * x	a := x * x
b := 3	
c := x	
d := a	
e := 6	
f := a + a	f := a + a
g := 6 * f	g := 6 * f

Optimiser

- Modéliser la propriété sur les chemins,
- Effet sur les blocs de base,
- Transformer les blocs de base (optimisation locale)
- Calcul sur le graphe de flot de contrôle
- Modifier le graphe de flot de contrôle (optimisation globale)

Transformations des blocs de base

- Elimination des sous expressions communes,
- Elimination du code inutile,
- Renommage de variables temporaires,
- Réorganisation des instructions indépendantes.
- Transformations algébriques

Exemple

Considérons l'arbre abstrait du programme :

```
    for (i=0 ; i < 10 ; i ++)  
for (j=0 ; j < 10 ; j++)  
    S[i,j] := A[i,j] + B[i,j]
```

La séquence de quadruplets (taille d'un entier = 4)

B1: $i := 0$

B2: if $i > 10$ goto B7

B3: $j := 0$

B4: if $j > 10$ goto B6

Le bloc B5

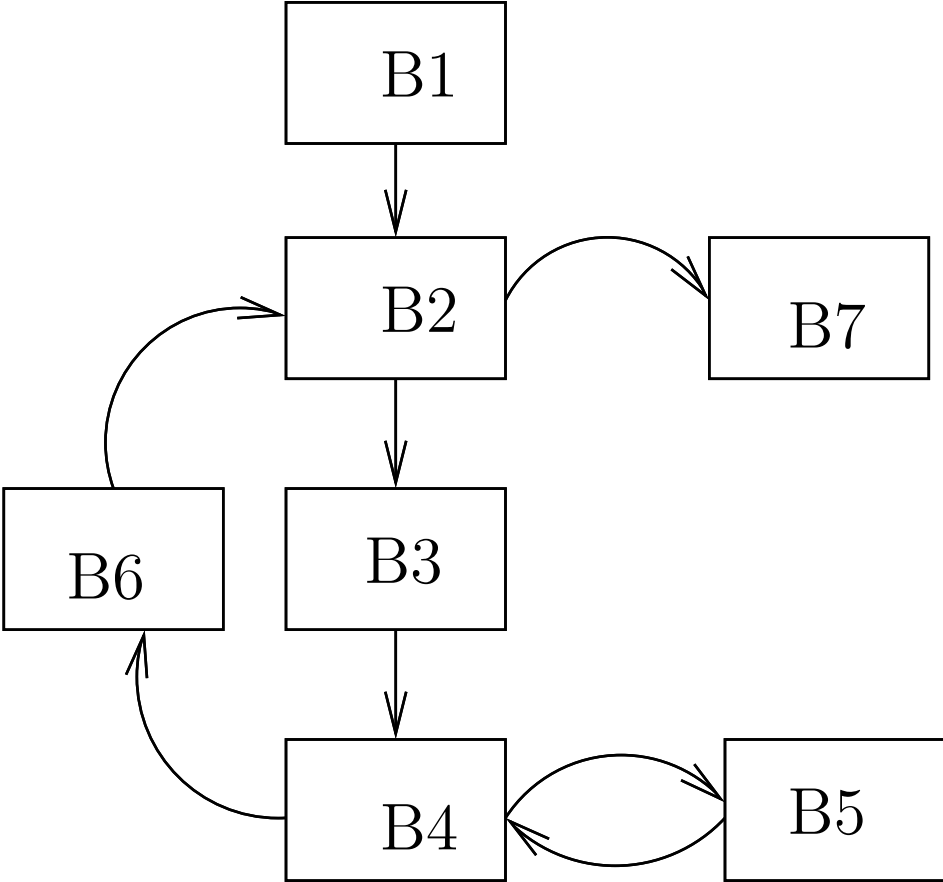
B5: $t1 := 4 * i$
 $t2 := 40 * j$
 $t3 := t1 + t2$
 $t4 := A[t3]$
 $t5 := 4 * i$
 $t6 := 40 * j$
 $t7 := t5 + t6$

$t8 := B[t7]$
 $t9 := t4 + t8$
 $t10 := 4 * i$
 $t11 := 40 * j$
 $t12 := t10 + t11$
 $S[t12] := t9$
 $j := j + 1$
goto B4

B6: $i := i + 1$

goto B2

B7:



Optimisation de B5

B5: $t1 := 4 * i$
 $t2 := 40 * j$
 $t3 := t1 + t2$
 $t4 := A[t3]$
 $t5 := 4 * i$
 $t6 := 40 * j$
 $t7 := t5 + t6$

$t8 := B[t7]$
 $t9 := t4 + t8$
 $t10 := 4 * i$
 $t11 := 40 * j$
 $t12 := t10 + t11$
 $S[t12] := t9$
 $j := j + 1$
goto B4

$t1, t5, t10$ désignent la même quantité.

Optimisation de B5

B5: $t1 := 4 * i$
 $t2 := 40 * j$
 $t3 := t1 + t2$
 $t4 := A[t3]$
 $t6 := 40 * j$
 $t7 := t1 + t6$

$t8 := B[t7]$
 $t9 := t4 + t8$
 $t11 := 40 * j$
 $t12 := t1 + t11$
 $S[t12] := t9$
 $j := j + 1$
goto B4

$t2, t6, t11$ désignent la même quantité.

Optimisation de B5

B5: $t1 := 4 * i$

$t2 := 40 * j$

$t3 := t1 + t2$

$t4 := A[t3]$

$t7 := t1 + t2$

$t8 := B[t7]$

$t9 := t4 + t8$

$t12 := t1 + t2$

$S[t12] := t9$

$j := j + 1$

goto B4

$t3, t7, t12$ désignent la même quantité.

Optimisation de B5

B5: $t1 := 4 * i$
 $t2 := 40 * j$
 $t3 := t1 + t2$
 $t4 := A[t3]$
 $t8 := B[t3]$
 $t9 := t4 + t8$
 $S[t3] := t9$
 $j := j + 1$
goto B4

Remarques

- Expressions syntaxiquement identiques,
- Pas de modification directe ou indirecte des opérandes entre deux occurrences consécutives

\implies **Calculs redondants**

5. Analyse flot de données

Exemple : calculs redondants

Un calcul est redondant en un point du programme si :

- le calcul se trouve en ce point,
- tout chemin allant du bloc initial jusqu'à ce point contient ce calcul
- les opérandes ne sont pas modifiés entre le dernier calcul et le calcul se trouvant en ce point.

On va propager vers l'avant les calculs pour détecter les redondances.

Analyse flot de données

- Analyse en avant, en arrière,
- Définition de systèmes d'équations, pour modéliser des propriétés : à chaque bloc de base est associée une propriété locale, les équations expriment le flot,
- Résolution de systèmes d'équations, calcul de plus grand, plus petit point-fixe
- Notion d'abstraction (interprétation abstraite, connexions de Galois),
- Notion d'approximation, extrapolation : élargissement, rétrécissement.

Forme des équations

Analyse en avant

Pour chaque bloc de base b , est définie une fonction de transfert F_b :

Analyse en avant, PPPF	$\begin{aligned} \text{In}(b) &= \begin{cases} \perp & \text{si } b \text{ est initial} \\ \bigsqcup_{b' \in \text{pre}(b)} \text{Out}(b') & \text{sinon.} \end{cases} \\ \text{Out}(b) &= F_b(\text{In}(b)) \end{aligned}$
Analyse en avant, PGPF	$\begin{aligned} \text{In}(b) &= \begin{cases} \perp & \text{si } b \text{ est initial} \\ \prod_{b' \in \text{pre}(b)} \text{Out}(b') & \text{sinon.} \end{cases} \\ \text{Out}(b) &= F_b(\text{In}(b)) \end{aligned}$

Analyse en arrière

<p>Analyse en arrière, PPPF</p>	$\text{Out}(b) = \begin{cases} \perp & \text{si } b \text{ est final} \\ \bigsqcup_{b' \in \text{succ}(b)} \text{In}(b') & \text{sinon.} \end{cases}$ $\text{In}(b) = F_b(\text{Out}(b))$
<p>Analyse en arrière, PGPF</p>	$\text{Out}(b) = \begin{cases} \perp & \text{si } b \text{ est final} \\ \prod_{b' \in \text{succ}(b)} \text{In}(b') & \text{sinon.} \end{cases}$ $\text{In}(b) = F_b(\text{Out}(b))$

Résolution dans un treillis

Soit (E, \leq) un *ordre partiel*, X une partie de E , $a \in E$. a est un majorant (resp minorant) de X si $\forall x \in X . x \leq a$ (resp. $\forall x \in X . a \leq x$).

La borne supérieure (resp. inférieure) de X est le plus petit des majorants (resp. le plus grand des minorants).

Un treillis (resp. treillis complet) est un *ordre partiel* tel que toute partie finie (resp. toute partie) admet une borne supérieure et une borne inférieure.

Une fonction est continue pour la borne supérieure (resp. inférieure) si pour toute chaîne croissante (resp. décroissante) l'image de la borne égale la borne de l'image.

Théorèmes

Théorème de Tarski : L'ensemble des points-fixes d'une fonction monotone croissante d'un treillis complet vers lui même est un sous treillis complet. La borne inf de l'ensemble $\{x \in E \mid f(x) \leq x\}$ (ensemble des post-points-fixes de f) est le plus petit point-fixe de f .

Théorème de Kleene : Le PPPF (resp. PGPF) d'une fonction continue s'obtient en itérant à partir du plus petit (resp. grand) élément ;

$$\text{lfp}(f) = \sqcup \{f^i(\perp) \mid i \in \mathbb{N}\} \quad \text{gfp}(f) = \sqcap \{f^i(\top) \mid i \in \mathbb{N}\}$$

Calculs redondants

- Calcul des expressions disponibles
- Suppression des calculs redondants

Expressions disponibles

Propriétés locales (par rapport à un bloc de base b)

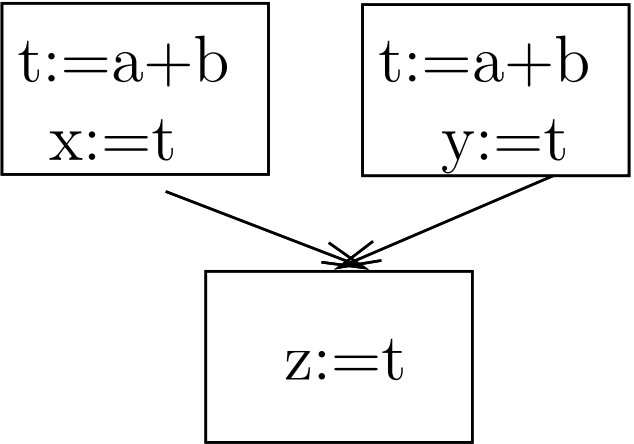
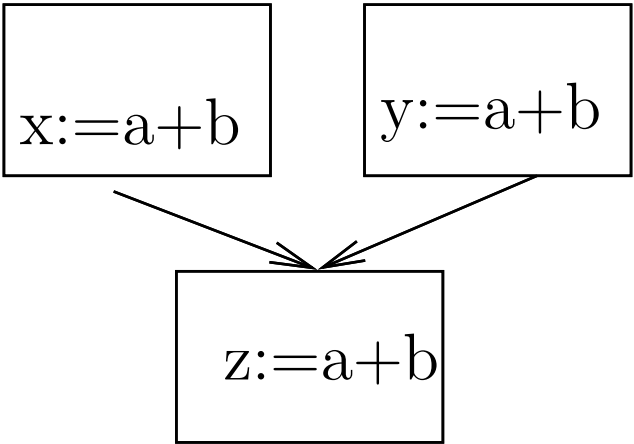
1. $\text{Kill}(b)$ est l'ensemble des expressions supprimées de l'ensemble des expressions disponibles, dans le bloc de base.
2. $\text{Gen}(b)$ est l'ensemble des expressions générées dans le bloc de base.
3. $F_b = (\text{In}(b) \setminus \text{Kill}(b)) \cup \text{Gen}(b)$

Suppression des calculs redondants

Résolution

1. $\sqcap = \bigcap$, $\sqcup = \bigcup$, on initialise tous les blocs différents du bloc initial à l'ensemble de toutes les expressions, le bloc initial est initialisé à \emptyset .
2. Suppression des calculs redondants on remplace $\mathbf{x} := \mathbf{e}$, par
$$\begin{cases} u := e \\ x := u \end{cases}$$

Calculs redondants



Variables actives

- Une variable x est *inactive* en un point i du programme si tout chemin allant de i à un point j ne contient aucune utilisation de cette variable ; j étant soit le bloc de sortie soit un bloc contenant une affectation à x , qui ne soit pas précédée d'une utilisation de x .
- Une instruction $d:x := e$ est *inutile* si x est inactive en fin de bloc et n'est pas utilisée dans le bloc après l'instruction.

Analyse des variables actives

Propriétés locales

$\text{Gen}(i)$ est l'ensemble des variables x telles que : x est utilisée dans le bloc i , et dans ce bloc i , toute éventuelle affectation à x est placée après la première utilisation de x .

$\text{Kill}(i)$ est l'ensemble des variables x affectées dans le bloc i .

Variables actives

Résolution

-

$$\text{Out}(b) = \bigcup_{b' \in \text{succ}(b)} \text{In}(b')$$

$$\text{In}(b) = (\text{Out}(b) \setminus \text{Kill}(b)) \cup \text{Gen}(b)$$

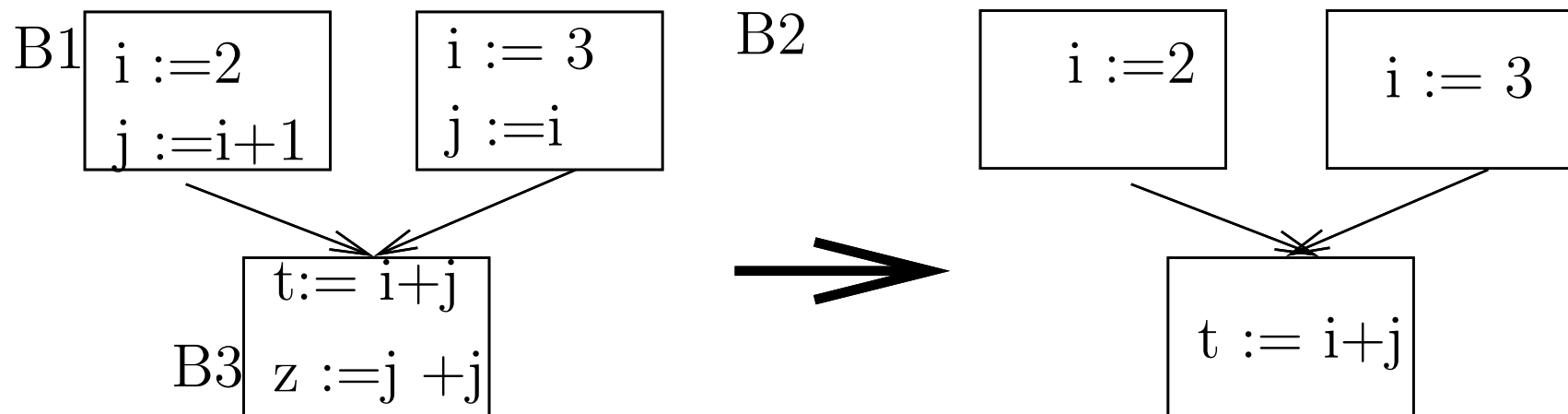
- initialisation des $\text{Out}(b)$ à \emptyset .
- suppression des instructions inutiles.

Suppression du code inutile

Une fois réalisée le calcul des variables actives,

- on détermine les variables inactives en fin de bloc de base,
- on supprime les instructions $x := e$ telles que
 - x est inactive en fin de bloc,
 - x n'est pas utilisée jusqu'à la fin du bloc.

Propagation des constantes



- Une variable est constante si sa valeur est connue à la compilation.
- A La sortie de B1 et B2, i et j sont constantes
- A l'entrée de B3 i est non constante, j est constante.

Propagation des constantes

Chaque variable prend ses valeurs dans un domaine. étendu par deux valeurs : \top (information non-constante) et \perp indique l'absence d'information. On veut définir un treillis sur le domaine étendu $D \cup \{\top, \perp\}$. La relation d'ordre est définie comme suit :

si $v \in D$ alors $\perp < v$ et $v < \top$.

On définit la borne supérieure \sqcup :

Pour $x \in D \cup \{\top, \perp\}$ et $v_1, v_2 \in D$

$x \sqcup \top = \top$	$x \sqcup \perp = x$	$v_1 \sqcup v_2 = \top$ si $v_1 \neq v_2$	$v_1 \sqcup v_1 = v_1$
------------------------	----------------------	---	------------------------

Propagation des constantes : équations

Treillis $[\mathcal{V} \mapsto D \cup \{\top, \perp\}]$

$$\begin{aligned} In(b) &= \begin{cases} \lambda x. \perp & \text{si } b \text{ est initial,} \\ \bigsqcup_{b' \in pre(b)} Out(b') & \text{sinon} \end{cases} \\ Out(b) &= F_b(In(b)) \end{aligned}$$

Fonction de transfert (induction sur la séquence des affectations de b)

$\mathbf{b} ::= \epsilon \mid \mathbf{x} := e ; \mathbf{b}$

$$F_{\mathbf{x} := e ; \mathbf{b}}(f) = F_{\mathbf{b}}(f[x \mapsto f(e)])$$

$$F_{\epsilon}(f) = f$$

6. Analyse de flot de contrôle

Objectifs

- Découvrir la structure de contrôle du programme,
- Identifier les boucles.
- Algorithmes de parcours de graphe, en *profondeur*, en *largeur*, calcul des composantes fortement connexes maximales, arbres de couverture.
- Calcul des dominants : *a domine b* ssi tout chemin d'exécution allant du bloc initial à *b* passe par *a*. Dualement, *a postdomine b* ssi tout chemin d'exécution allant du bloc *a* au bloc final passe par *b*.
- Calcul des intervalles,
- Analyse structurelle.