Projet: Variables vivantes

J.-P. Bodeveix Yannick Chevalier Martin Strecker

11 mai 2005

Plan

Concept (1)

Définition : Variable v est vivante dans une position P, si

- il y a un chemin d'exécution commençant avec P sur lequel v est lue
- sans avoir été redéfinie précédemment

Concept (2)

Exemple :

```
x = 3;        (1)
y = 5;        (2)
y = x + 3;    (3)
x = x + y;    (4)
```

x vivante (au moins) après (1), (2), (3)
y vivante (au moins) après (3)
y morte après (2)

Utilisation

Optimisation de compilateurs :

Affectations à des variables mortes peuvent être éliminées (“dead code elimination”)

```
x = 3;       (1)
y = 5;       (2)
y = x + 3;   (3)
x = x + y;   (4)
```

Utilisation

Optimisation de compilateurs :

Affectations à des variables mortes peuvent être éliminées (“dead code elimination”)

```
x = 3;        (1)
y = 5;        (2)
y = x + 3;    (3)
x = x + y;    (4)
```

```
x = 3;        (1)

y = x + 3;    (3)
x = x + y;    (4)
```

Raffinement

Question : x est-elle vivante après (4) ?

```
. . .
x = x + y;  (4)
. . .
```

Réponse :

- *Oui*, si x est utilisée plus tard, par ex. dans `return(x + y)`
- *Non*, si x n'est plus utilisée, par ex. dans `return(y)`

Variables vivantes : Complications (1)

y est-elle vivante après (1),
puisque y n'est plus utilisée dans (2) ?
L'optimisation suivante est-elle correcte ?

Original

```
y = 5;            (1)
IF (x > 0)
THEN x = x + 1    (2)
ELSE x = x + y    (3)
return(x)
```

Variables vivantes : Complications (1)

y est-elle vivante après (1),
puisque y n'est plus utilisée dans (2) ?
L'optimisation suivante est-elle correcte ?

Original

```
y = 5;              (1)
IF (x > 0)
THEN x = x + 1      (2)
ELSE x = x + y      (3)
return(x)
```

Optimisation (incorrecte)

```
IF (x > 0)
THEN x = x + 1      (2)
ELSE x = x + y      (3)
return(x)
```

Variables vivantes : Complications (2)

x est-elle vivante après (3),
puisque x n'est pas utilisée à la fin de la boucle ?
L'optimisation suivante est-elle correcte ?

Original

```
x = 2;            (1)
y = 5;            (2)
WHILE x > 0 DO
  x = x - 1;      (3)
return(y)
```

Variables vivantes : Complications (2)

x est-elle vivante après (3),
puisque x n'est pas utilisée à la fin de la boucle ?
L'optimisation suivante est-elle correcte ?

Original

```
x = 2;          (1)
y = 5;          (2)
WHILE x > 0 DO
  x = x - 1;    (3)
return(y)
```

Optimisation (incorrecte)

```
x = 2;          (1)
y = 5;          (2)
WHILE x > 0 DO
  SKIP;         (3)
return(y)
```

Concept

Définition : Une expression dans un programme est expression très utilisée si

- elle est utilisée sur tous les chemins d'exécution du programme
- avant que l'une de ses variables soit redéfinie

Exemple

```
while (u > z) do
  if (u + z > 1) then
    u := (x + y) - (u + z)
  else
    z := (x + y) + (u + z)
  end;
end;
return(x+y);
```

très utilisée : `x + y`
pas très utilisée : `u + z`

Optimisation

```
aux := (x + y);
while (u > z) do
  if (u + z > 1) then
    u := aux - (u + z)
  else
    z := aux + (u + z)
  end;
end;
return(aux);
```

Expressions

Constructeurs :

- Constantes (entier) :
  `3, 42`, ...
- Variables (chaîne de caractère) :
  `x, y5`
- Expressions binaires :
  `a + b - 5 == c + 4`

Expressions

Constructeurs :

- Constantes (entier) : `3, 42`, ...
- Variables (chaîne de caractère) : `x, y5`
- Expressions binaires : `a + b - 5 == c + 4`

Déf. en Caml :

```
type expr =
   Const of int

 | Var of string

 | Op of
    binop * expr * expr
```

Expressions

Constructeurs :

- Constantes (entier) : `3, 42`, ...
- Variables (chaîne de caractère) : `x, y5`
- Expressions binaires : `a + b - 5 == c + 4`

Déf. en Caml :

```
type expr =
   Const of int

 | Var of string

 | Op of
    binop * expr * expr
```

Exemple :

`x + 3 == 7` $\rightsquigarrow$

`Op(BEq, Op(BPlus, Var "x", Const 3), Const 7)`

Commandes

Constructeurs :

- Skip : “ne fait rien”
- Affectation : `x = x + 4`
- Séquence : `x = x + 4; y = z;`
- Sélection : `if (..) .. else ..`
- Boucle `while (..) ..`
- Return `return (x + 1)`

Déf. en Caml ? ?

Mini-langage impératif (3)

Synt. concrète :

```
x = 5;
y = 0;

while (x) {
  x = x - 1;
  y = y + 2;
  }

return y;
```

Syntaxe abstraite :

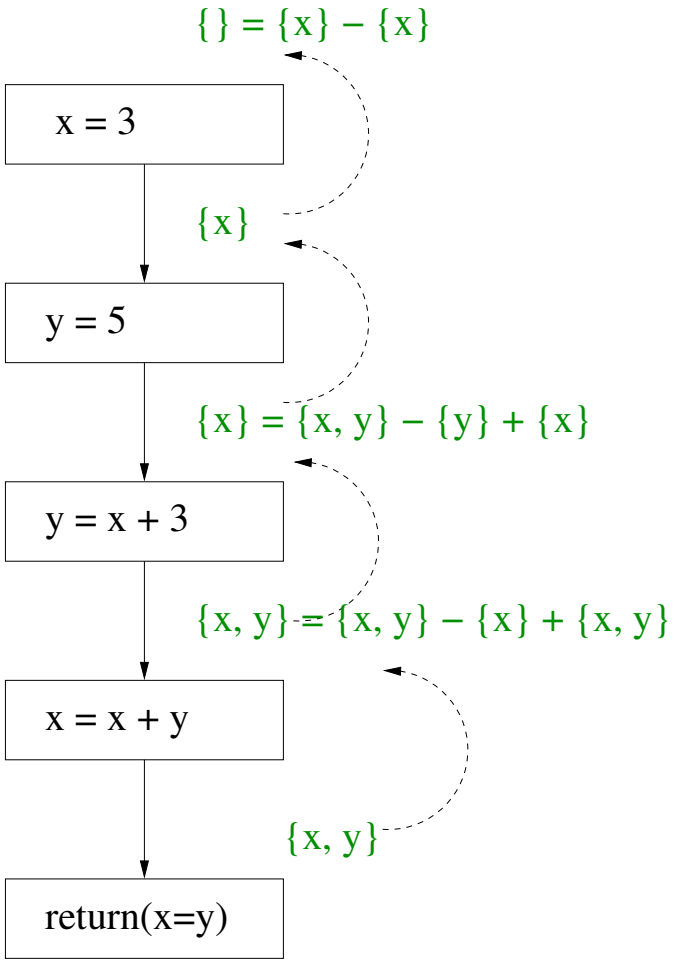
```
Comp (Comp (Comp (
  Assign ("x", Const 5),
  Assign ("y", Const 7)),

  While ((Var "x"),
   Comp(Assign("x",
    Op (BMinus, Var "x", Const 1)),
        Assign("y",
    Op (BPlus, Var "y", Const 2))))),

  Return (Var "y"))
```

A faire : écrire fonction qui imprime des termes

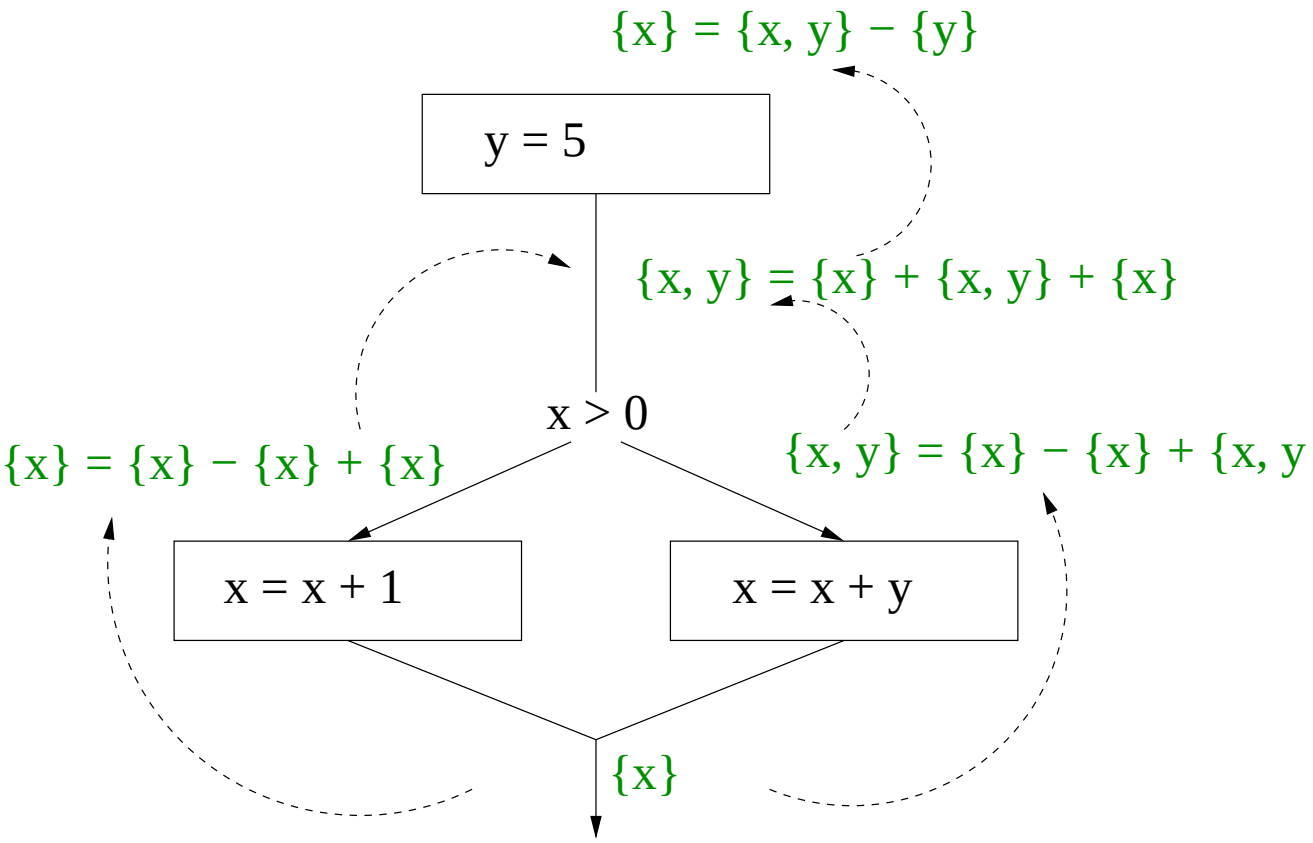
Principe

- Propager l'ensemble des variables vivantes . . .
- de la fin vers le début du programme
- Comparable au calcul de la plus faible précondition

Return, Affectation, Séquence



Sélection



Boucle (1)

Idée : Déplier la boucle. Sont équivalents :

Original

```
while (x > 0) {
  x = x - 1;
  }
```

Boucle (1)

Idée : Déplier la boucle. Sont équivalents :

Original

```
while (x > 0) {
  x = x - 1;
  }
```

Déplié 1 fois :

```
if (x > 0) {
  x = x - 1;
  }
while (x > 0) {
  x = x - 1;
  }
```

Boucle (1)

Idée : Déplier la boucle. Sont équivalents :

Original

```
while (x > 0) {
  x = x - 1;
  }
```

Déplié 1 fois :

```
if (x > 0) {
  x = x - 1;
  }
while (x > 0) {
  x = x - 1;
  }
```

Déplié *n* fois :

```
if (x > 0) {
  x = x - 1;
  }
...
if (x > 0) {
  x = x - 1;
  }
Skip;
```

Boucle (2)

```
while (x > 0) {

  x = x - 1;

  }
return(y)
```

Boucle (2)

```
                    {x, y}
while (x > 0) {
                    {y}
  x = x - 1;
                    {y}
  }
return(y)
```

Boucle (2)

```
                    {x, y}        {x, y}
while (x > 0) {
                    {y}           {x, y}
  x = x - 1;
                    {y}           {x, y}
  }
return(y)
```

Principe

Comparable au calcul des variables vivantes :

- Propager l'ensemble des *expressions très utilisées*
- de la fin vers le début du programme
- *Cas de l'affectation* `x = e`
  - Supprimer de l'ensemble toute sous-expr. contenant x
  - Ajouter toutes les sous-expressions de e
- *Cas de la sélection* `if` b `then` c_1 `else` c_2
  - Prendre intersection des expr. très utilisées de c_1 et c_2
  - Ajouter toutes les sous-expressions de b

Exemple

```
while (u > z) {

   if (u + z > 1) {

     u := (x + y) - (u + z)
      }
   else {

     v := (x + y) + (u + z)
      }
   }

return(x + y);
```

Exemple

```
{ x+y, u>z }
while (u > z) {
  { x+y, u+z, u+z > 1 }
   if (u + z > 1) {
     { x+y, u+z, (x+y) - (u+z) }
     u := (x + y) - (u + z)
     {  } }
   else {
     { x+y, u+z, (x+y) + (u+z) }
     v := (x + y) + (u + z)
     { } }
  {  } }
{ x+y }
return(x + y);
```

Exemple

```
{ x+y, u>z }
while (u > z) {
  { x+y, u+z, u+z > 1 }
   if (u + z > 1) {
     { x+y, u+z, (x+y) - (u+z) }
     u := (x + y) - (u + z)
     { x+y, u>z } }
   else {
     { x+y, u+z, (x+y) + (u+z), u>z }
     v := (x + y) + (u + z)
     { x+y, u>z } }
  { x+y, u>z } }
{ x+y }
return(x + y);
```

Principe

- Traverser programme (fin $\rightsquigarrow$ début)
- Pour chaque affectation `x = e` : Calculer ensemble $\mathcal{V}$ des variables vivantes
- Deux cas :
  - $x \notin \mathcal{V}$: Remplacer `x = e` par `Skip`
  - $x \in \mathcal{V}$: Garder `x = e`
- Simplifier `c ; Skip` ou `Skip ; c`

Exemple

Progr. original

```
x = 3;

y = 5;

y = x + 3;

return (x + y);
```

Exemple

Progr. original	var. vivantes
	{ }
`x = 3;`	{x}
`y = 5;`	{x}
`y = x + 3;`	{x, y}
`return (x + y);`	

Exemple

Progr. original	var. vivantes	optimisé
	{ }	
`x = 3;`		`x = 3;`
	{x}	
`y = 5;`		
	{x}	
`y = x + 3;`		`y = x + 3;`
	{x, y}	
`return (x + y);`		`return (x + y)`

Travail à faire

- Écrire fonctions en OCaml / Java pour la partie “analyse de programme”
- Écrire fonctions en C pour la partie “affichage de graphe”
- Écrire interface entre OCaml / Java / C