

Data-Flow Analysis

Sergey Mechtaev
mechtaev@pku.edu.cn

Peking University

The While Language

- Simple language for studying analyses
- A While program is a statement (or a sequence of statements)
- Elementary blocks (assignments, tests and *skip* statements) are labelled

Syntactic Categories

$a \in AExp$ – arithmetic expressions

$b \in BExp$ – boolean expressions

$S \in Stmt$ – statements

$x, y \in Var$ – variables

$n \in Num$ – numerals

$l \in Lab$ – labels

$op_a \in Op_a$ – arithmetic operators

$op_b \in Op_b$ – boolean operators

$op_r \in Op_r$ – relational operators

Syntax

$a ::= x \mid n \mid a_1 \text{ op}_a a_2$

$b ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2$

$S ::= [x := a]^l \mid [\text{skip}]^l \mid S_1; S_2$
 $\mid \text{if } [b]^l \text{ then } S_1 \text{ else } S_2 \mid \text{while } [b]^l \text{ do } S$

a = Arithmetic expression

b = Boolean expression

S = Statement (program)

$[...]^l \rightarrow$ elementary block

$^l \rightarrow$ label allows to identify the primitive constructs of a program

Example Program (Factorial)

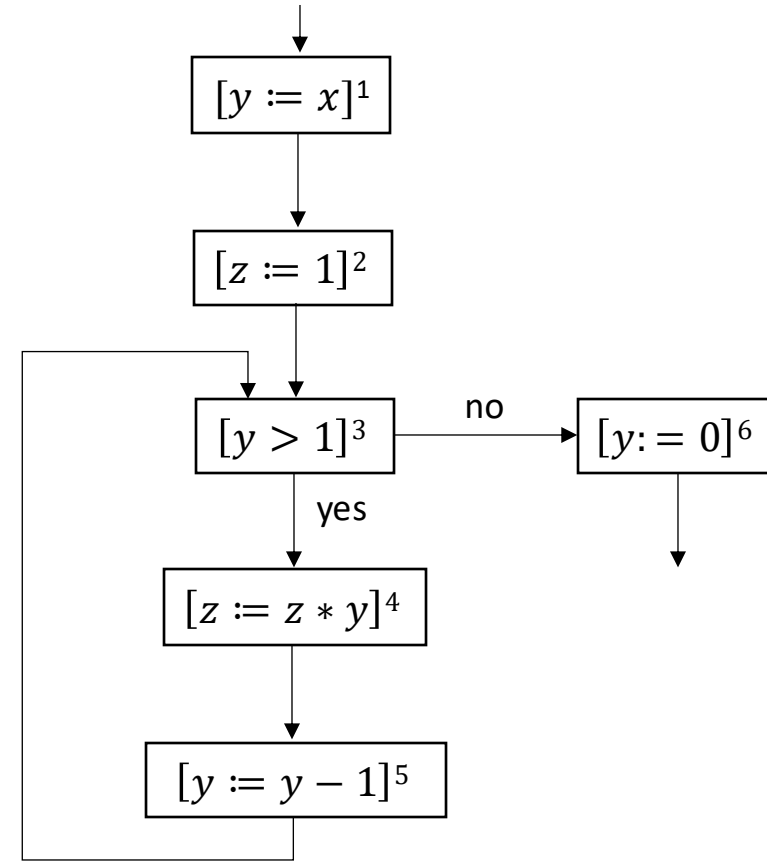
```
[y := x]1;  
[z := 1]2;  
while [y > 1]3 do  
    ([z := z * y]4;  
    [y := y - 1]5);  
[y := 0]6
```

Data Flow Analysis

- **Data flow analysis** is a technique for gathering information about the possible set of values calculated at various points in a computer program.
- The program represented using **control-flow graph** (CFG)
- The information inferred from each node of CFG is described using **lattice**.

Control Flow Graph

```
[y := x]1;  
[z := 1]2;  
while [y > 1]3 do  
  ([z := z * y]4;  
   [y := y - 1]5);  
[y := 0]6
```



NODES = elementary blocks

EDGES = describe how control might pass from one elementary block to another

Initial Labels

The *init* function returns the initial label of a statement:

$$\mathit{init}([x := a]^l) = l$$

$$\mathit{init}([skip]^l) = l$$

$$\mathit{init}(S_1; S_2) = \mathit{init}(S_1)$$

$$\mathit{init}(\text{if } [b]^l \text{ then } S_1 \text{ else } S_2) = l$$

$$\mathit{init}(\text{while } [b]^l \text{ do } S) = l$$

Final Labels

The *final* function returns the set of final labels of a statement:

$$\begin{aligned} \text{final}([x := a]^l) &= \{l\} \\ \text{final}([skip]^l) &= \{l\} \\ \text{final}(S_1; S_2) &= \text{final}(S_2) \\ \text{final}(\text{if } [b]^l \text{ then } S_1 \text{ else } S_2) &= \text{final}(S_1) \cup \text{final}(S_2) \\ \text{final}(\text{while } [b]^l \text{ do } S) &= \{l\} \end{aligned}$$

Blocks

The *blocks* function collects the elementary blocks associated with a statements:

$$blocks([x := a]^l) = \{[x := a]^l\}$$

$$blocks([skip]^l) = \{[skip]^l\}$$

$$blocks(S_1; S_2) = blocks(S_1) \cup blocks(S_2)$$

$$blocks(if [b]^l then S_1 else S_2) = \{[b]^l\} \cup blocks(S_1) \cup blocks(S_2)$$

$$blocks(while [b]^l do S) = \{[b]^l\} \cup blocks(S)$$

Flow

The *flow* function extracts edges of the flow graph as pairs (l, l') :

$$flow([x := a]^l) = \emptyset$$

$$flow([skip]^l) = \emptyset$$

$$flow(S_1; S_2) = flow(S_1) \cup flow(S_2) \cup \{(l, init(S_2)) \mid l \in final(S_1)\}$$

$$init(if [b]^l then S_1 else S_2) = flow(S_1) \cup flow(S_2)$$

$$\cup \{(l, init(S_1)), (l, init(S_2))\}$$

$$init(while [b]^l do S) = flow(S) \cup \{(l, init(S))$$

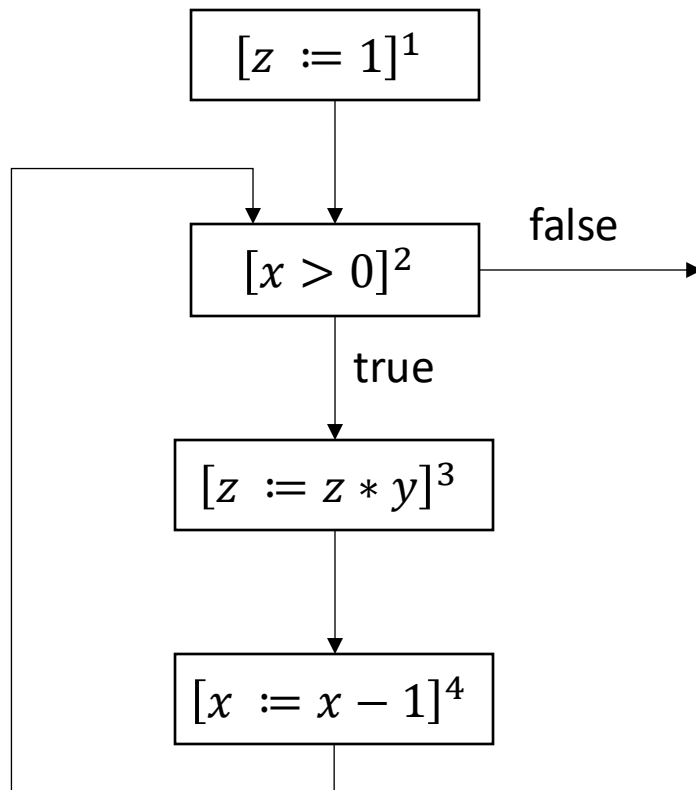
$$\cup \{(l', l) \mid l' \in final(S)\}$$

Reverse Flow

Edges of the flow graph for backward analysis:

$$flow^R(S) = \{(l', l) \mid (l, l') \in flow(S)\}$$

Example (The Power Program)



$init(Power) = 1$
 $final(Power) = \{2\}$
 $labels(Power) = \{1,2,3,4\}$
 $flow(Power)$
 $= \{(1,2), (2,3), (3,4), (4,2)\}$

Classification of Analyses

- Intraprocedural vs interprocedural
- Flow-sensitive vs flow-insensitive
- Context-insensitive vs context-sensitive
- Must vs may
- Forward vs backward

Intraprocedural vs Interprocedural

- **Intraprocedural analysis** is a mechanism for performing analysis for each function, using only the information available for that.
- **Interprocedural analysis** is a mechanism for performing analysis across function boundaries.

Flow-sensitive vs flow-insensitive

- **Flow-sensitive** analyses is an analysis whose results depends on the order of statements (requires a model of program state at each program point).
- **Flow-insensitive** is an analysis whose result is the same regardless of the statement order (requires only a single global state).

Context-insensitive vs context-sensitive

- A **context-insensitive** analysis is an interprocedural analysis that cannot distinguish between different calls of a procedure (the analysis information is combined for all call sites)
- A **context-sensitive** analysis is an interprocedural analysis that takes the context of procedure calls into account (more precise, but also more costly).

Must vs may

- **Must analysis** detect properties that are satisfied by all paths of execution.
- **May analysis** detect properties that are satisfied by at least one execution path.

Forward vs Backward

- **Forward analysis** propagates information from the beginning to the end of the program
- **Backward analysis** propagates information from the end to the beginning of the program.

Four Classic Analyses

	Forward	Backward
Must	Available Expressions	Very Busy Expressions
May	Reaching Definitions	Live Variables

Four Classic Analyses

	Forward	Backward
Must	Available Expressions	Very Busy Expressions
May	Reaching Definitions	Live Variables

Available Expressions

Definition. For each program point, which expressions must have already been computed, and not later modified, on all paths to the program point.

It is a *forward must* analysis.

Application: optimization (don't recompute expressions that are still available).

Example

```
[x := a + b]1;  
[y := a * b]2;  
while [y > a + b]3 do  
    [a := a + 1]4;  
    [x := a + b]5;
```

The expression $a + b$ is available every time execution reaches the condition 3, therefore the expression need not be recomputed.

Killed Expression

An expression is killed in a block if any of the variables used in the expression are modified in the block:

$$\begin{aligned}kill_{AE}([x := a]^l) &= \{a' \in AExp_* \mid x \in Vars(a')\} \\kill_{AE}([skip]^l) &= \emptyset \\kill_{AE}([b]^l) &= \emptyset\end{aligned}$$

where $AExp_*$ are all expressions in the program

Killed Expression

An expression is killed in a block if any of the variables used in the expression are modified in the block:

$$kill_{AE}([x := a]^l) = \{a' \in AExp_* \mid x \in Vars(a')\}$$

$$kill_{AE}([skip]^l) = \emptyset$$

$$kill_{AE}([b]^l) = \emptyset$$

Assignment statement:
kills all expressions that
use variable x assigned
in the block because
they have to be
recomputed again

where $AExp_*$ are all expressions in the program

Generated Expression

A generated expression is an expression that is evaluated in the block and where none of the variables used in the expression are later modified in the block:

$$\begin{aligned} gen_{AE}([x := a]^l) &= \{a' \in AExp(a) \mid x \notin Vars(a')\} \\ get_{AE}([skip]^l) &= \emptyset \\ gen_{AE}([b]^l) &= AExp(b) \end{aligned}$$

Analysis

The goal of the analysis is to compute the largest set satisfying the equation for AE_{entry} :

$$AE_{entry}(l) = \begin{cases} \emptyset & \text{if } l = \text{init}(\text{program}) \\ \cap \{AE_{exit}(l') \mid (l', l) \in \text{flow}(\text{program})\} & \text{otherwise} \end{cases}$$

$$AE_{exit}(l) = \left(AE_{entry}(l) \setminus \text{kill}(B^l) \right) \cup \text{gen}_{AE}(B^l)$$

where $B^l \in \text{blocks}(\text{program})$

Example

```
[x := a + b]1;  
[y := a * b]2;  
while [y > a + b]3 do  
  [a := a + 1]4;  
  [x := a + b]5;
```

l	$kill_{AE}(l)$	$gen_{AE}(l)$
1	\emptyset	$\{a + b\}$
2	\emptyset	$\{a * b\}$
3	\emptyset	$\{a + b\}$
4	$\{a + b, a * b, a + 1\}$	\emptyset
5	\emptyset	$\{a + b\}$

Example

$$AE_{entry}(1) = \emptyset$$

$$AE_{entry}(2) = AE_{exit}(1)$$

$$AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)$$

$$AE_{entry}(4) = AE_{exit}(3)$$

$$AE_{entry}(5) = AE_{exit}(4)$$

$$AE_{exit}(1) = AE_{entry}(1) \cup \{a + b\}$$

$$AE_{exit}(2) = AE_{entry}(2) \cup \{a * b\}$$

$$AE_{exit}(3) = AE_{entry}(3) \cup \{a + b\}$$

$$AE_{exit}(4)$$

$$= AE_{entry}(4) \setminus \{a + b, a * b, a + 1\}$$

$$AE_{exit}(5) = AE_{entry}(5) \cup \{a + b\}$$

Example

Equations for entry and exit functions:

$$AE_{entry}(1) = \emptyset$$

$$AE_{entry}(2) = AE_{exit}(1)$$

$$AE_{entry}(3) = AE_{exit}(2) \cap AE_{exit}(5)$$

$$AE_{entry}(4) = AE_{exit}(3)$$

$$AE_{entry}(5) = AE_{exit}(4)$$

$$AE_{exit}(1) = AE_{entry}(1) \cup \{a + b\}$$

$$AE_{exit}(2) = AE_{entry}(2) \cup \{a * b\}$$

$$AE_{exit}(3) = AE_{entry}(3) \cup \{a + b\}$$

$$AE_{exit}(4) = AE_{entry}(4) \setminus \{a + b, a * b, a + 1\}$$

$$AE_{exit}(5) = AE_{entry}(5) \cup \{a + b\}$$

AE at the entry of Block 3 = AE available at the exit of Block 2 (when entering the loop for the first time) and of Block 5 (when coming back from the exit of the loop)

Example

```
[x := a + b]1;  
[y := a * b]2;  
while [y > a + b]3 do  
  [a := a + 1]4;  
  [x := a + b]5;
```

l	$AE_{entry}(l)$	$AE_{exit}(l)$
1	\emptyset	$\{a + b\}$
2	$\{a + b\}$	$\{a + b, a * b\}$
3	$\{a + b\}$	$\{a + b\}$
4	$\{a + b\}$	\emptyset
5	\emptyset	$\{a + b\}$

Four Classic Analyses

	Forward	Backward
Must	Available Expressions	Very Busy Expressions
May	Reaching Definitions	Live Variables

Reaching Definitions

Reaching definitions analysis determines for each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.


It is *forward may* analysis.

Applications: Bug-finding (uninitialized variables), optimization (constant propagation)

Example

```
[x := 5]1;  
[y := 1]2;  
while [x > 1]3 do  
    [y := x * y]4;  
    [x := x - 1]5;
```

All assignments reach the entry of 4; only the assignments 1,4,5 reach the entry of 5.



because the assignment 2 is overwritten by the assignment 4

Killed Assignments

An assignment is killed by a block if the block assigns a new value to the variable:

$$kill_{RD} : Blocks_* \rightarrow P(Var_* \times Lab_*)$$



Set of pairs of variables and labels corresponding to the place where the variables are assigned

Killed Assignments

An assignment is killed by a block if the block assigns a new value to the variable:

$$\begin{aligned} kill_{RD}([x := a]^l) &= \{(x, ?)\} \cup \{(x, l') \mid B^{l'} \text{ is an assignment to } x\} \\ kill_{RD}([skip]^l) &= \emptyset \\ kill_{RD}([b]^l) &= \emptyset \end{aligned}$$

where ? is the label for uninitialised variables.

Generated Assignments

Assignments that appear in the block:

$$gen_{RD}([x := a]^l) = \{(x, l)\}$$

$$gen_{RD}([skip]^l) = \emptyset$$

$$gen_{RD}([b]^l) = \emptyset$$

Analysis

The goal of the analysis is to compute the smallest set satisfying the equation for RD_{entry} :

$$RD_{entry}(l) = \begin{cases} \{(x, ?) \mid x \in Vars\} & \text{if } l = \text{init}(\text{program}) \\ \cup \{RD_{exit}(l') \mid (l', l) \in \text{flow}(\text{program})\} & \text{otherwise} \end{cases}$$

$$RD_{exit}(l) = \left(RD_{entry}(l) \setminus kill_{RD}(B^l) \right) \cup gen_{RD}(B^l)$$

where $B^l \in \text{blocks}(\text{program})$

Example

```
[x := 5]1;  
[y := 1]2;  
while [x > 1]3 do  
  [y := x * y]4;  
  [x := x - 1]5;
```

l	$kill_{RD}(l)$	$gen_{RD}(l)$
1	$\{(x, ?), (x, 1), (x, 5)\}$	$\{(x, 1)\}$
2	$\{(y, ?), (y, 2), (y, 4)\}$	$\{(y, 2)\}$
3	\emptyset	\emptyset
4	$\{(y, ?), (y, 2), (y, 4)\}$	$\{(y, 4)\}$
5	$\{(x, ?), (x, 1), (x, 5)\}$	$\{(x, 5)\}$

Example

$$RD_{entry}(1) = \{(x, ?), (y, ?)\}$$

$$RD_{entry}(2) = RD_{exit}(1)$$

$$RD_{entry}(3) = RD_{exit}(2) \cup RD_{exit}(5)$$

$$RD_{entry}(4) = RD_{exit}(3)$$

$$RD_{entry}(5) = RD_{exit}(4)$$

$$RD_{exit}(1) = (RD_{entry}(1) \setminus \{(x, ?), (x, 1), (x, 5)\}) \cup \{(x, 1)\}$$

$$RD_{exit}(2) = (RD_{entry}(2) \setminus \{(y, ?), (y, 2), (y, 4)\}) \cup \{(y, 2)\}$$

$$RD_{exit}(3) = RD_{entry}(3)$$

$$RD_{exit}(4) = (RD_{entry}(4) \setminus \{(y, ?), (y, 2), (y, 4)\}) \cup \{(y, 4)\}$$

$$RD_{exit}(5) = (RD_{entry}(5) \setminus \{(x, ?), (x, 1), (x, 5)\}) \cup \{(x, 5)\}$$

Example

```
[x := 5]1;  
[y := 1]2;  
while [x > 1]3 do  
  [y := x * y]4;  
  [x := x - 1]5;
```

l	$RD_{entry}(l)$	$RD_{exit}(l)$
1	$\{(x, ?), (y, ?)\}$	$\{(y, ?), (x, 1)\}$
2	$\{(y, ?), (x, 1)\}$	$\{(x, 1), (y, 2)\}$
3	$\{(x, 1), (y, 2), (y, 4), (x, 5)\}$	$\{(x, 1), (y, 2), (y, 4), (x, 5)\}$
4	$\{(x, 1), (y, 2), (y, 4), (x, 5)\}$	$\{(x, 1), (y, 4), (x, 5)\}$
5	$\{(x, 1), (y, 4), (x, 5)\}$	$\{(y, 4), (x, 5)\}$

Four Classic Analyses

	Forward	Backward
Must	Available Expressions	Very Busy Expressions
May	Reaching Definitions	Live Variables

Very Busy Expressions

An expression is **very busy** at the exit from a label if, no matter what path is taken from the label, the expression must always be used before any of the variables occurring in it are redefined.

Very busy expressions analysis determines for each program point, which expressions must be busy at the exit from the point.

It is *backward must* analysis.

Applications: Optimization (evaluate the expression in the block and store its value for later use, aka *hoisting* the expression)

Example

```
if  $[a > b]^1$  then  
     $[x := b - a]^2;$   
     $[y := a - b]^3;$   
else  
     $[y := b - a]^4;$   
     $[x := a - b]^5;$ 
```

$a - b$ and $b - a$ are both very busy at the start of the conditional, can be hoisted to reduce the size of generated code.

Killed Expressions

An expression is killed in a block if any of the variables used in the expression are modified in the block:

$$kill_{VB}: Blocks_* \rightarrow P(AExp_*)$$

$$kill_{VB}([x := a]^l) = \{a' \in AExp_* \mid x \in Vars(a')\}$$

$$kill_{VB}([skip]^l) = \emptyset$$

$$kill_{VB}([b]^l) = \emptyset$$

where $AExp_*$ are all expressions in the program.

Killed Expressions

An expression is killed in a block if any of the variables used in the expression are modified in the block:

$$kill_{VB}: Blocks_* \rightarrow P(AExp_*)$$

$$kill_{VB}([x := a]^l) = \{a' \in AExp_* \mid x \in Vars(a')\}$$

$$kill_{VB}([skip]^l) = \emptyset$$

$$kill_{VB}([b]^l) = \emptyset$$

For **assignment statements**: all the variables that hold expressions using the variables that has been assigned need to be killed

where $AExp_*$ are all expressions in the program.

Generated Expressions

A very busy expression is generated:

$$gen_{VB} : Blocks_* \rightarrow P(AExp_*)$$

$$gen_{VB}([x := a]^l) = AExp(a)$$

$$gen_{VB}([skip]^l) = \emptyset$$

$$gen_{VB}([b]^l) = AExp(b)$$

if it appears on the **right-hand side** of an assignment or inside an **if or loop condition**.

Analysis

The goal of the analysis is to compute the largest set satisfying the equation for VB_{exit} :

$$VB_{exit}(l) = \begin{cases} \emptyset & \text{if } l = \text{init}(\text{program}) \\ \cap \{VB_{entry}(l') \mid (l', l) \in \text{flow}^R(\text{program})\} & \text{otherwise} \end{cases}$$

$$VB_{entry}(l) = \left(VB_{exit}(l) \setminus \text{kill}_{VB}(B^l) \right) \cup \text{gen}_{VB}(B^l)$$

where $B^l \in \text{blocks}(\text{program})$

Example

if $[a > b]^1$ *then*
 $[x := b - a]^2$;
 $[y := a - b]^3$;
else
 $[y := b - a]^4$;
 $[x := a - b]^5$;

l	$kill_{RD}(l)$	$gen_{RD}(l)$
1	\emptyset	\emptyset
2	\emptyset	$\{b - a\}$
3	\emptyset	$\{a - b\}$
4	\emptyset	$\{b - a\}$
5	\emptyset	$\{a - b\}$

Example

$$VB_{entry}(1) = VB_{exit}(1)$$

$$VB_{entry}(2) = VB_{exit}(2) \cup \{b - a\}$$

$$VB_{entry}(3) = \{a - b\}$$

$$VB_{entry}(4) = VB_{exit}(4) \cup \{b - a\}$$

$$VB_{entry}(5) = \{a - b\}$$

$$VB_{exit}(1) = VB_{entry}(2) \cap VB_{entry}(4)$$

$$VB_{exit}(2) = VB_{entry}(3)$$

$$VB_{exit}(3) = \emptyset$$

$$VB_{exit}(4) = VB_{entry}(5)$$

$$VB_{exit}(5) = \emptyset$$

Example

if $[a > b]^1$ *then*
 $[x := b - a]^2$;
 $[y := a - b]^3$;
else
 $[y := b - a]^4$;
 $[x := a - b]^5$;

l	$VB_{entry}(l)$	$VB_{exit}(l)$
1	$\{a - b, b - a\}$	$\{a - b, b - a\}$
2	$\{a - b, b - a\}$	$\{a - b\}$
3	$\{a - b\}$	\emptyset
4	$\{a - b, b - a\}$	$\{a - b\}$
5	$\{a - b\}$	\emptyset

Four Classic Analyses

	Forward	Backward
Must	Available Expressions	Very Busy Expressions
May	Reaching Definitions	Live Variables

Live Variables

A **variable is live** at the exit from a label if there exists a path from the label to a use of the variable that does not re-define the variable.

Live variables analysis determines for each program point, which variables may be live at the exit from the point.

It is *backward may* analysis.

Applications: Optimization (don't store variables that aren't live, eliminate assignments where variables are dead)

Example

```
[x := 2]1;  
[y := 4]2;  
[x := 1]3;  
if [y > x]4 then  
    [z := y]5;  
else  
    [z := y * y]6;  
[x := z]7
```

The variable x is not live at the exit from label 1 (the assignment is redundant).

Both x and y are live at the exit from label 3.

Killed Variables

A variable is killed by an assignment if it appears on the left hand side of it:

$$kill_{LV}: Blocks_* \rightarrow P(Var_*)$$

$$kill_{LV}([x := a]^l) = \{x\}$$

$$kill_{LV}([skip]^l) = \emptyset$$

$$kill_{LV}([b]^l) = \emptyset$$

Generated Variables

A variable is generated in the block:

$$gen_{LV} : Blocks_* \rightarrow P(Var_*)$$

$$gen_{LV}([x := a]^l) = Vars(a)$$

$$gen_{LV}([skip]^l) = \emptyset$$

$$gen_{LV}([b]^l) = Vars(b)$$

if it appears on the **right-hand side** of an assignment or in some **condition**.

Analysis

The goal of the analysis is to compute the smallest set satisfying the equation for LV_{exit} :

$$LV_{exit}(l) = \begin{cases} \emptyset & \text{if } l = final(program) \\ \cup \{LV_{entry}(l') \mid (l', l) \in flow^R(program)\} & \text{otherwise} \end{cases}$$

$$LV_{entry}(l) = \left(LV_{exit}(l) \setminus kill(B^l) \right) \cup gen_{LV}(B^l)$$

where $B^l \in blocks(program)$

Example

```
[x := 2]1;  
[y := 4]2;  
[x := 1]3;  
if [y > x]4 then  
    [z := y]5;  
else  
    [z := y * y]6;  
[x := z]7
```

l	$kill_{AE}(l)$	$gen_{AE}(l)$
1	$\{x\}$	\emptyset
2	$\{y\}$	\emptyset
3	$\{x\}$	\emptyset
4	\emptyset	$\{x, y\}$
5	$\{z\}$	$\{y\}$
6	$\{z\}$	$\{y\}$
7	$\{x\}$	$\{z\}$

Example

$$LV_{entry}(1) = LV_{exit}(1) \setminus \{x\}$$

$$LV_{entry}(2) = LV_{exit}(2) \setminus \{y\}$$

$$LV_{entry}(3) = LV_{exit}(3) \setminus \{x\}$$

$$LV_{entry}(4) = LV_{exit}(4) \cup \{x, y\}$$

$$LV_{entry}(5) = (LV_{exit}(5) \setminus \{z\}) \cup \{y\}$$

$$LV_{entry}(6) = (LV_{exit}(6) \setminus \{z\}) \cup \{y\}$$

$$LV_{entry}(7) = \{z\}$$

$$LV_{exit}(1) = LV_{entry}(2)$$

$$LV_{exit}(2) = LV_{entry}(3)$$

$$LV_{exit}(3) = LV_{entry}(4)$$

$$LV_{exit}(4) = LV_{entry}(5) \cup LV_{entry}(6)$$

$$LV_{exit}(5) = LV_{entry}(7)$$

$$LV_{exit}(6) = LV_{entry}(4)$$

$$LV_{exit}(7) = \emptyset$$

Example

Equations for entry and exit functions:

$$LV_{exit}(1) = LV_{entry}(2)$$

$$LV_{exit}(2) = LV_{entry}(3)$$

$$LV_{exit}(3) = LV_{entry}(4)$$

$$LV_{exit}(4) = LV_{entry}(5) \cup LV_{entry}(6)$$

$$LV_{exit}(5) = LV_{entry}(7)$$

$$LV_{exit}(6) = LV_{entry}(4)$$

$$LV_{exit}(7) = \emptyset$$

$$LV_{entry}(1) = LV_{exit}(1) \setminus \{x\}$$

$$LV_{entry}(2) = LV_{exit}(2) \setminus \{y\}$$

$$LV_{entry}(3) = LV_{exit}(3) \setminus \{x\}$$

$$LV_{entry}(4) = LV_{exit}(4) \cup \{x, y\}$$

$$LV_{entry}(5) = (LV_{exit}(5) \setminus \{z\}) \cup \{y\}$$

$$LV_{entry}(6) = (LV_{exit}(6) \setminus \{z\}) \cup \{y\}$$

$$LV_{entry}(7) = \{z\}$$

Live variables at exit of Block 4 = union of live variables at entry of Block 5 and Block 6, corresponding to both the branches (if the variables are used somewhere, they are going to be live also in the if condition).

Example

$[x := 2]^1;$
 $[y := 4]^2;$
 $[x := 1]^3;$
if $[y > x]^4$ *then*
 $[z := y]^5;$
else
 $[z := y * y]^6;$
 $[x := z]^7$

l	$LV_{entry}(l)$	$LV_{exit}(l)$
1	\emptyset	\emptyset
2	\emptyset	$\{y\}$
3	$\{y\}$	$\{x, y\}$
4	$\{x, y\}$	$\{y\}$
5	$\{y\}$	$\{z\}$
6	$\{y\}$	$\{z\}$
7	$\{z\}$	\emptyset