Semantics and Verification of Software

Lecture 14: Dataflow Analysis

Thomas Noll

Lehrstuhl für Informatik 2
RWTH Aachen University
noll@cs.rwth-aachen.de

http://www-i2.informatik.rwth-aachen.de/i2/svsw/

Summer semester 2007

Outline

Correction: Operational Semantics of Proc's I

Example

```
c = begin
      var x; var y;   } v
      proc F is
        begin
          var z;
          z := x;
          if z=1 then skip
                 else x := x-1;        }
                      call F;          } c2   } c1   } cF   } p
                      y := z * y       }
        end
      x := 2; y := 1; call F   } c0
    end
```

$$\frac{\mathsf{upd}_v(v,\rho), \mathsf{upd}_p(p, \mathsf{upd}_v(v,\rho), \pi) \vdash \langle c, \sigma\rangle \to \sigma'}{\rho, \pi \vdash \langle \texttt{begin}\ v\ p\ c\ \texttt{end}, \sigma\rangle \to \sigma'}\ \text{(block)}$$

Correction: Operational Semantics of Proc's II

- **Problem:** variable environments $VEnv := \{\rho \mid \rho : Var \dashrightarrow Loc\}$ (with $Loc := \mathbb{N}$) do not provide enough information about availability of locations (see Example 12.5)
- **Solution:** store maintains allocation information

$$Sto := \{\sigma \mid \sigma : Loc \dashrightarrow \mathbb{Z}\}$$

- Update function for variable declarations takes store into account:

$$\mathsf{upd}_v : VDec \times VEnv \times Sto \to VEnv \times Sto$$
$$\mathsf{upd}_v(\mathtt{var}\ x;, \rho, \sigma) := (\rho[x \mapsto l_x], \sigma[l_x \mapsto 0])$$

where $l_x := \min\{l \in \mathbb{N} \mid \sigma(l) = \bot\}$

- (block) rule becomes

$$\frac{\mathsf{upd}_v(v, \rho, \sigma) = (\rho', \sigma') \qquad \rho', \mathsf{upd}_p(p, \rho', \pi) \vdash \langle c, \sigma' \rangle \to \sigma''}{\rho, \pi \vdash \langle \mathtt{begin}\ v\ p\ c\ \mathtt{end}, \sigma \rangle \to \sigma''}\ \text{(block')}$$

Correction: Operational Semantics of Proc's II

- **Problem:** variable environments $VEnv := \{\rho \mid \rho : Var \dashrightarrow Loc\}$ (with $Loc := \mathbb{N}$) do not provide enough information about availability of locations (see Example 12.5)
- **Solution:** store maintains allocation information

$$Sto := \{\sigma \mid \sigma : Loc \dashrightarrow \mathbb{Z}\}$$

- Update function for variable declarations takes store into account:

$$\mathsf{upd}_v : VDec \times VEnv \times Sto \to VEnv \times Sto$$
$$\mathsf{upd}_v(\texttt{var } x\texttt{;}, \rho, \sigma) := (\rho[x \mapsto l_x], \sigma[l_x \mapsto 0])$$

where $l_x := \mathsf{min}\{l \in \mathbb{N} \mid \sigma(l) = \bot\}$

- (block) rule becomes

$$\frac{\mathsf{upd}_v(v, \rho, \sigma) = (\rho', \sigma') \qquad \rho', \mathsf{upd}_p(p, \rho', \pi) \vdash \langle c, \sigma' \rangle \to \sigma''}{\rho, \pi \vdash \langle \texttt{begin } v\ p\ c \texttt{ end}, \sigma \rangle \to \sigma''} \text{(block')}$$

Correction: Operational Semantics of Proc's II

- **Problem:** variable environments $VEnv := \{\rho \mid \rho : Var \dashrightarrow Loc\}$ (with $Loc := \mathbb{N}$) do not provide enough information about availability of locations (see Example 12.5)
- **Solution:** store maintains allocation information

$$Sto := \{\sigma \mid \sigma : Loc \dashrightarrow \mathbb{Z}\}$$

- Update function for variable declarations takes store into account:

$$\mathsf{upd}_v : VDec \times VEnv \times Sto \rightarrow VEnv \times Sto$$
$$\mathsf{upd}_v(\mathtt{var}\ x\mathtt{;}, \rho, \sigma) := (\rho[x \mapsto l_x], \sigma[l_x \mapsto 0])$$

where $l_x := \mathsf{min}\{l \in \mathbb{N} \mid \sigma(l) = \bot\}$

- (block) rule becomes

$$\frac{\mathsf{upd}_v(v, \rho, \sigma) = (\rho', \sigma') \qquad \rho', \mathsf{upd}_p(p, \rho', \pi) \vdash \langle c, \sigma' \rangle \rightarrow \sigma''}{\rho, \pi \vdash \langle \mathtt{begin}\ v\ p\ c\ \mathtt{end}, \sigma \rangle \rightarrow \sigma''}\ \text{(block')}$$

Correction: Operational Semantics of Proc's II

- **Problem:** variable environments $VEnv := \{\rho \mid \rho : Var \dashrightarrow Loc\}$ (with $Loc := \mathbb{N}$) do not provide enough information about availability of locations (see Example 12.5)
- **Solution:** store maintains allocation information

$$Sto := \{\sigma \mid \sigma : Loc \dashrightarrow \mathbb{Z}\}$$

- Update function for variable declarations takes store into account:

$$\begin{aligned}\mathsf{upd}_v : VDec \times VEnv \times Sto &\to VEnv \times Sto\\ \mathsf{upd}_v(\texttt{var}\ x\texttt{;}, \rho, \sigma) &:= (\rho[x \mapsto l_x], \sigma[l_x \mapsto 0])\end{aligned}$$

where $l_x := \min\{l \in \mathbb{N} \mid \sigma(l) = \bot\}$
- (block) rule becomes

$$\frac{\mathsf{upd}_v(v, \rho, \sigma) = (\rho', \sigma') \qquad \rho', \mathsf{upd}_p(p, \rho', \pi) \vdash \langle c, \sigma' \rangle \to \sigma''}{\rho, \pi \vdash \langle \texttt{begin}\ v\ p\ c\ \texttt{end}, \sigma \rangle \to \sigma''}\ \text{(block')}$$

- **Alternative solution (by Martin Plücker):** use same location for all instances of a variable and reset to old value upon leaving the block
- New block rule:

$$\frac{\rho', \mathsf{upd}_p(p, \rho', \pi) \vdash \langle c, \sigma \rangle \to \sigma''}{\rho, \pi \vdash \langle \texttt{begin}\ v\ p\ c\ \texttt{end}, \sigma \rangle \to \sigma'} \text{(block)}$$

where $\rho' := \mathsf{upd}_v(v, \rho)$ and, for every $l \in Loc$,

$$\sigma'(l) := \begin{cases} \sigma''(l) & \text{if ex. } x \in Var : \rho'(x) = \rho(x) = l \\ \sigma(l) & \text{otherwise} \end{cases}$$

- No modification of stores and update function required:

$$Sto := \{\sigma \mid \sigma : Loc \to \mathbb{Z}\}$$

$$\mathsf{upd}_v : VDec \times VEnv \to VEnv$$
$$\mathsf{upd}_v(\texttt{var}\ x\texttt{;}, \rho) := \rho[x \mapsto \mathsf{min}(Loc \setminus \rho(Var))]$$

Correction: Operational Semantics of Proc's III

- **Alternative solution (by Martin Plücker):** use same location for all instances of a variable and reset to old value upon leaving the block
- New block rule:

$$\frac{\rho', \mathsf{upd}_p(p, \rho', \pi) \vdash \langle c, \sigma \rangle \to \sigma''}{\rho, \pi \vdash \langle \texttt{begin}\ v\ p\ c\ \texttt{end}, \sigma \rangle \to \sigma'} \text{(block)}$$

where $\rho' := \mathsf{upd}_v(v, \rho)$ and, for every $l \in \mathit{Loc}$,

$$\sigma'(l) := \begin{cases} \sigma''(l) & \text{if ex. } x \in \mathit{Var} : \rho'(x) = \rho(x) = l \\ \sigma(l) & \text{otherwise} \end{cases}$$

- No modification of stores and update function required:

$$\mathit{Sto} := \{\sigma \mid \sigma : \mathit{Loc} \to \mathbb{Z}\}$$

$$\mathsf{upd}_v : \mathit{VDec} \times \mathit{VEnv} \to \mathit{VEnv}$$
$$\mathsf{upd}_v(\texttt{var}\ x\texttt{;}, \rho) := \rho[x \mapsto \mathsf{min}(\mathit{Loc} \setminus \rho(\mathit{Var}))]$$

Correction: Operational Semantics of Proc's III

- **Alternative solution (by Martin Plücker):**
  use same location for all instances of a variable and reset to old value upon leaving the block
- New block rule:

$$\frac{\rho', \mathsf{upd}_p(p, \rho', \pi) \vdash \langle c, \sigma \rangle \to \sigma''}{\rho, \pi \vdash \langle \texttt{begin}\ v\ p\ c\ \texttt{end}, \sigma \rangle \to \sigma'} \text{(block)}$$

  where $\rho' := \mathsf{upd}_v(v, \rho)$ and, for every $l \in Loc$,

$$\sigma'(l) := \begin{cases} \sigma''(l) & \text{if ex. } x \in Var : \rho'(x) = \rho(x) = l \\ \sigma(l) & \text{otherwise} \end{cases}$$

- No modification of stores and update function required:

$$Sto := \{\sigma \mid \sigma : Loc \to \mathbb{Z}\}$$

$$\mathsf{upd}_v : VDec \times VEnv \to VEnv$$
$$\mathsf{upd}_v(\texttt{var}\ x\texttt{;}, \rho) := \rho[x \mapsto \min(Loc \setminus \rho(Var))]$$

Outline

1. Correction: Operational Semantics of Blocks and Procedures
2. Preliminaries on Dataflow Analysis
3. An Example: Available Expressions Analysis

Dataflow Analysis: the Approach

- Traditional form of program analysis
- Idea: describe how analysis information flows through program
- Distinctions:

  direction of flow: forward vs. backward analyses

  procedures: interprocedural vs. intraprocedural analyses

  quantification over paths: may (union) vs. must (intersection) analyses

  dependence on statement order: flow–sensitive vs. flow–insensitive analyses

  distinction of procedure calls: context–sensitive vs. context–insensitive analyses

Dataflow Analysis: the Approach

- Traditional form of program analysis
- Idea: describe how analysis information flows through program
- Distinctions:

  direction of flow: forward vs. backward analyses
  procedures: interprocedural vs. intraprocedural analyses
  quantification over paths: may (union) vs. must (intersection) analyses
  dependence on statement order: flow–sensitive vs. flow–insensitive analyses
  distinction of procedure calls: context–sensitive vs. context–insensitive analyses

Dataflow Analysis: the Approach

- Traditional form of program analysis
- Idea: describe how analysis information flows through program
- Distinctions:

  direction of flow: forward vs. backward analyses

  procedures: interprocedural vs. intraprocedural analyses

  quantification over paths: may (union) vs. must (intersection) analyses

  dependence on statement order: flow–sensitive vs. flow–insensitive analyses

  distinction of procedure calls: context–sensitive vs. context–insensitive analyses

Labelled Programs

- Goal: localization of analysis information
- Dataflow information will be associated with
  - assignments
  - tests in conditionals (`if`) and loops (`while`)
  - `skip` statements

  These constructs will be called blocks.
- Assume set of labels Lab with meta variable $l \in Lab$ (usually $Lab = \mathbb{N}$)

Definition 14.1 (Labelled WHILE programs)

The syntax of labelled WHILE programs is defined by the following context–free grammar:

$$
\begin{aligned}
a &::= z \mid x \mid a_1\texttt{+}a_2 \mid a_1\texttt{-}a_2 \mid a_1\texttt{*}a_2 \in AExp \\
b &::= t \mid a_1\texttt{=}a_2 \mid a_1\texttt{>}a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp \\
c &::= [\texttt{skip}]^l \mid [x \texttt{ := } a]^l \mid c_1\texttt{;}c_2 \mid \\
&\qquad \texttt{if } [b]^l \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } [b]^l \texttt{ do } c \in Cmd
\end{aligned}
$$

Here all labels in a statement $c \in Cmd$ are assumed to be distinct.

Labelled Programs

- Goal: localization of analysis information
- Dataflow information will be associated with
  - assignments
  - tests in conditionals (`if`) and loops (`while`)
  - `skip` statements

  These constructs will be called blocks.
- Assume set of labels *Lab* with meta variable $l \in Lab$ (usually $Lab = \mathbb{N}$)

Definition 14.1 (Labelled WHILE programs)

The syntax of labelled WHILE programs is defined by the following context–free grammar:

$$
\begin{aligned}
a &::= z \mid x \mid a_1\texttt{+}a_2 \mid a_1\texttt{-}a_2 \mid a_1\texttt{*}a_2 \in AExp\\
b &::= t \mid a_1\texttt{=}a_2 \mid a_1\texttt{>}a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp\\
c &::= [\texttt{skip}]^l \mid [x \texttt{ := } a]^l \mid c_1\texttt{;}c_2 \mid\\
&\qquad \texttt{if } [b]^l \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } [b]^l \texttt{ do } c \in Cmd
\end{aligned}
$$

Here all labels in a statement $c \in Cmd$ are assumed to be distinct.

Labelled Programs

- Goal: localization of analysis information
- Dataflow information will be associated with
  - assignments
  - tests in conditionals (`if`) and loops (`while`)
  - `skip` statements

  These constructs will be called blocks.
- Assume set of labels *Lab* with meta variable $l \in Lab$ (usually $Lab = \mathbb{N}$)

Definition 14.1 (Labelled WHILE programs)

The syntax of labelled WHILE programs is defined by the following context–free grammar:

$$a ::= z \mid x \mid a_1\texttt{+}a_2 \mid a_1\texttt{-}a_2 \mid a_1\texttt{*}a_2 \in AExp$$
$$b ::= t \mid a_1\texttt{=}a_2 \mid a_1\texttt{>}a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp$$
$$c ::= [\texttt{skip}]^l \mid [x \texttt{ := } a]^l \mid c_1 \texttt{;} c_2 \mid$$
$$\texttt{if } [b]^l \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } [b]^l \texttt{ do } c \in Cmd$$

Here all labels in a statement $c \in Cmd$ are assumed to be distinct.

Labelled Programs

- Goal: localization of analysis information
- Dataflow information will be associated with
  - assignments
  - tests in conditionals (`if`) and loops (`while`)
  - `skip` statements

  These constructs will be called blocks.
- Assume set of labels Lab with meta variable $l \in Lab$ (usually $Lab = \mathbb{N}$)

Definition 14.1 (Labelled WHILE programs)

The syntax of labelled WHILE programs is defined by the following context–free grammar:

$$
\begin{aligned}
a &::= z \mid x \mid a_1\texttt{+}a_2 \mid a_1\texttt{-}a_2 \mid a_1\texttt{*}a_2 \in AExp \\
b &::= t \mid a_1\texttt{=}a_2 \mid a_1\texttt{>}a_2 \mid \neg b \mid b_1 \wedge b_2 \mid b_1 \vee b_2 \in BExp \\
c &::= [\texttt{skip}]^l \mid [x \texttt{ := } a]^l \mid c_1\texttt{;}c_2 \mid \\
&\qquad \texttt{if } [b]^l \texttt{ then } c_1 \texttt{ else } c_2 \mid \texttt{while } [b]^l \texttt{ do } c \in Cmd
\end{aligned}
$$

Here all labels in a statement $c \in Cmd$ are assumed to be distinct.

A WHILE Program

Example 14.2

```
x := 6;
y := 7;
z := 0;
while x > 0 do
  x := x - 1;
  v := y;
  while v > 0 do
    v := v - 1;
    z := z + 1;
```

A WHILE Program with Labels

Example 14.2

```
[x := 6]^1;
[y := 7]^2;
[z := 0]^3;
while [x > 0]^4 do
  [x := x - 1]^5;
  [v := y]^6;
  while [v > 0]^7 do
    [v := v - 1]^8;
    [z := z + 1]^9
```

Representing Control Flow I

Every (labelled) statement has a single entry (given by the initial label) and generally multiple exits (given by the final labels):

Definition 14.3 (Initial and final labels)

The mapping $\mathsf{init} : \mathit{Cmd} \to \mathit{Lab}$ returns the initial label of a statement:

$$\begin{aligned}
\mathsf{init}([\texttt{skip}]^l) &:= l \\
\mathsf{init}([x\ \texttt{:=}\ a]^l) &:= l \\
\mathsf{init}(c_1\,\texttt{;}\,c_2) &:= \mathsf{init}(c_1) \\
\mathsf{init}(\texttt{if}\ [b]^l\ \texttt{then}\ c_1\ \texttt{else}\ c_2) &:= l \\
\mathsf{init}(\texttt{while}\ [b]^l\ \texttt{do}\ c) &:= l
\end{aligned}$$

Representing Control Flow I

Every (labelled) statement has a single entry (given by the initial label) and generally multiple exits (given by the final labels):

Definition 14.3 (Initial and final labels)

The mapping $\mathsf{init} : \mathit{Cmd} \to \mathit{Lab}$ returns the initial label of a statement:

$$\begin{aligned}
\mathsf{init}([\mathtt{skip}]^l) &:= l \\
\mathsf{init}([x \; \mathtt{:=} \; a]^l) &:= l \\
\mathsf{init}(c_1 \, ; c_2) &:= \mathsf{init}(c_1) \\
\mathsf{init}(\mathtt{if} \; [b]^l \; \mathtt{then} \; c_1 \; \mathtt{else} \; c_2) &:= l \\
\mathsf{init}(\mathtt{while} \; [b]^l \; \mathtt{do} \; c) &:= l
\end{aligned}$$

The mapping $\mathsf{final} : \mathit{Cmd} \to 2^{\mathit{Lab}}$ returns the set of final labels of a statement:

$$\begin{aligned}
\mathsf{final}([\mathtt{skip}]^l) &:= \{l\} \\
\mathsf{final}([x \; \mathtt{:=} \; a]^l) &:= \{l\} \\
\mathsf{final}(c_1 \, ; c_2) &:= \mathsf{final}(c_2) \\
\mathsf{final}(\mathtt{if} \; [b]^l \; \mathtt{then} \; c_1 \; \mathtt{else} \; c_2) &:= \mathsf{final}(c_1) \cup \mathsf{final}(c_2) \\
\mathsf{final}(\mathtt{while} \; [b]^l \; \mathtt{do} \; c) &:= \{l\}
\end{aligned}$$

Representing Control Flow II

Definition 14.4 (Flow relation)

Given a statement $c \in Cmd$, the flow relation $\mathsf{flow}(c) \subseteq Lab \times Lab$ is defined by

$$\begin{aligned}
\mathsf{flow}([\texttt{skip}]^l) &:= \emptyset \\
\mathsf{flow}([x \texttt{ := } a]^l) &:= \emptyset \\
\mathsf{flow}(c_1 \texttt{;} c_2) &:= \mathsf{flow}(c_1) \cup \mathsf{flow}(c_2) \cup \\
&\quad\ \{(l, \mathsf{init}(c_2)) \mid l \in \mathsf{final}(c_1)\} \\
\mathsf{flow}(\texttt{if } [b]^l \texttt{ then } c_1 \texttt{ else } c_2) &:= \mathsf{flow}(c_1) \cup \mathsf{flow}(c_2) \cup \\
&\quad\ \{(l, \mathsf{init}(c_1)), (l, \mathsf{init}(c_2))\} \\
\mathsf{flow}(\texttt{while } [b]^l \texttt{ do } c) &:= \mathsf{flow}(c) \cup \{(l, \mathsf{init}(c))\} \cup \\
&\quad\ \{(l', l) \mid l' \in \mathsf{final}(c)\}
\end{aligned}$$

Representing Control Flow III

Example 14.5

$c = $ [z := 1]^1;
    while [x > 0]^2 do
      [z := z*y]^3;
      [x := x-1]^4

Representing Control Flow III

Example 14.5

$$c = [\texttt{z := 1}]^1;$$
$$\quad \texttt{while } [\texttt{x > 0}]^2 \texttt{ do}$$
$$\qquad [\texttt{z := z*y}]^3;$$
$$\qquad [\texttt{x := x-1}]^4$$

$$\begin{aligned} \mathsf{init}(c) &= 1 \\ \mathsf{final}(c) &= \{2\} \\ \mathsf{flow}(c) &= \{(1,2),(2,3),(3,4),(4,2)\} \end{aligned}$$

Example 14.5

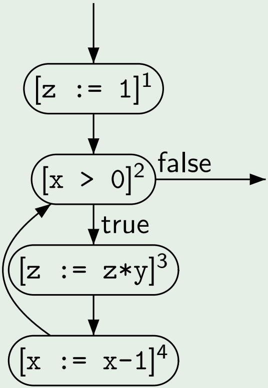
Visualization by flow graph:

```
c = [z := 1]^1;
    while [x > 0]^2 do
      [z := z*y]^3;
      [x := x-1]^4
```

$\mathsf{init}(c) = 1$

$\mathsf{final}(c) = \{2\}$

$\mathsf{flow}(c) = \{(1, 2), (2, 3), (3, 4), (4, 2)\}$



Representing Control Flow IV

- To simplify the presentation we will often assume that the program $c \in Cmd$ under consideration has an isolated entry, meaning that

  $$\{l \in Lab \mid (l, \mathsf{init}(c)) \in \mathsf{flow}(c)\} = \emptyset$$

  (which is the case when c does not start with a `while` loop)

Representing Control Flow IV

- To simplify the presentation we will often assume that the program $c \in Cmd$ under consideration has an isolated entry, meaning that

$$\{l \in Lab \mid (l, \mathsf{init}(c)) \in \mathsf{flow}(c)\} = \emptyset$$

(which is the case when c does not start with a `while` loop)

- Similarly: $c \in Cmd$ has isolated exits if

$$\{l' \in Lab \mid (l, l') \in \mathsf{flow}(c) \text{ for some } l \in \mathsf{final}(c)\} = \emptyset$$

Representing Control Flow IV

- To simplify the presentation we will often assume that the program $c \in Cmd$ under consideration has an isolated entry, meaning that

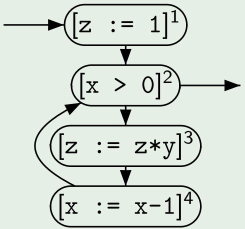
$$\{l \in Lab \mid (l, \mathsf{init}(c)) \in \mathsf{flow}(c)\} = \emptyset$$

(which is the case when c does not start with a `while` loop)

- Similarly: $c \in Cmd$ has isolated exits if

$$\{l' \in Lab \mid (l, l') \in \mathsf{flow}(c) \text{ for some } l \in \mathsf{final}(c)\} = \emptyset$$

Example 14.6



has an isolated entry but not isolated exits

Outline

Goal of the Analysis

Available Expressions Analysis

The goal of Available Expressions Analysis is to determine, for each program point, which (complex) expressions *must* have been computed, and not later modified, on all paths to the program point.

Goal of the Analysis

Available Expressions Analysis

The goal of Available Expressions Analysis is to determine, for each program point, which (complex) expressions *must* have been computed, and not later modified, on all paths to the program point.

- can be used to avoid recomputations of expressions
- for simplicity: only non–trivial arithmetic expressions

Goal of the Analysis

Available Expressions Analysis

The goal of Available Expressions Analysis is to determine, for each program point, which (complex) expressions *must* have been computed, and not later modified, on all paths to the program point.

- can be used to avoid recomputations of expressions
- for simplicity: only non–trivial arithmetic expressions

Example 14.7 (Available Expressions Analysis)

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

Goal of the Analysis

Available Expressions Analysis

The goal of Available Expressions Analysis is to determine, for each program point, which (complex) expressions *must* have been computed, and not later modified, on all paths to the program point.

- can be used to avoid recomputations of expressions
- for simplicity: only non–trivial arithmetic expressions

Example 14.7 (Available Expressions Analysis)

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

- a+b available at label 3

Goal of the Analysis

Available Expressions Analysis

The goal of Available Expressions Analysis is to determine, for each program point, which (complex) expressions *must* have been computed, and not later modified, on all paths to the program point.

- can be used to avoid recomputations of expressions
- for simplicity: only non–trivial arithmetic expressions

Example 14.7 (Available Expressions Analysis)

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

- a+b available at label 3
- a+b not available at label 5

Goal of the Analysis

Available Expressions Analysis

The goal of Available Expressions Analysis is to determine, for each program point, which (complex) expressions *must* have been computed, and not later modified, on all paths to the program point.

- can be used to avoid recomputations of expressions
- for simplicity: only non–trivial arithmetic expressions

Example 14.7 (Available Expressions Analysis)

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

- a+b available at label 3
- a+b not available at label 5
- possible optimization:
  `while [y > x]^3 do`

Formalizing the Analysis I

- Given $c \in Cmd$, $Lab_c/Block_c/AExp_c$ denote the sets of all labels/blocks/complex arithmetic expressions occurring in c, respectively

Formalizing the Analysis I

- Given $c \in Cmd$, $Lab_c/Block_c/AExp_c$ denote the sets of all labels/blocks/complex arithmetic expressions occurring in c, respectively
- An expression a is killed in a block B if any of the variables in a is modified in B

Formalizing the Analysis I

- Given $c \in Cmd$, $Lab_c/Block_c/AExp_c$ denote the sets of all labels/blocks/complex arithmetic expressions occurring in c, respectively
- An expression a is killed in a block B if any of the variables in a is modified in B
- Formally: $\mathsf{kill}_{\mathsf{AE}} : Block_c \to 2^{AExp_c}$ is defined by

$$\begin{aligned}\mathsf{kill}_{\mathsf{AE}}([\texttt{skip}]^l) &:= \emptyset\\ \mathsf{kill}_{\mathsf{AE}}([x \texttt{ := } a]^l) &:= \{a' \in AExp_c \mid x \in FV(a')\}\\ \mathsf{kill}_{\mathsf{AE}}([b]^l) &:= \emptyset\end{aligned}$$

Formalizing the Analysis I

- Given $c \in Cmd$, $Lab_c/Block_c/AExp_c$ denote the sets of all labels/blocks/complex arithmetic expressions occurring in c, respectively
- An expression a is killed in a block B if any of the variables in a is modified in B
- Formally: $\mathsf{kill_{AE}} : Block_c \to 2^{AExp_c}$ is defined by

$$\mathsf{kill_{AE}}([\texttt{skip}]^l) := \emptyset$$
$$\mathsf{kill_{AE}}([x \texttt{ := } a]^l) := \{a' \in AExp_c \mid x \in FV(a')\}$$
$$\mathsf{kill_{AE}}([b]^l) := \emptyset$$

- An expression a is generated in a block B if it is evaluated in and none of its variables are modified by B

Formalizing the Analysis I

- Given $c \in Cmd$, $Lab_c/Block_c/AExp_c$ denote the sets of all labels/blocks/complex arithmetic expressions occurring in c, respectively
- An expression a is killed in a block B if any of the variables in a is modified in B
- Formally: $\mathsf{kill}_{\mathsf{AE}} : Block_c \to 2^{AExp_c}$ is defined by

$$\begin{aligned}\mathsf{kill}_{\mathsf{AE}}([\mathtt{skip}]^l) &:= \emptyset\\ \mathsf{kill}_{\mathsf{AE}}([x \ \mathtt{:=}\ a]^l) &:= \{a' \in AExp_c \mid x \in FV(a')\}\\ \mathsf{kill}_{\mathsf{AE}}([b]^l) &:= \emptyset\end{aligned}$$

- An expression a is generated in a block B if it is evaluated in and none of its variables are modified by B
- Formally: $\mathsf{gen}_{\mathsf{AE}} : Block_c \to 2^{AExp_c}$ is defined by

$$\begin{aligned}\mathsf{gen}_{\mathsf{AE}}([\mathtt{skip}]^l) &:= \emptyset\\ \mathsf{gen}_{\mathsf{AE}}([x \ \mathtt{:=}\ a]^l) &:= \{a \mid x \notin FV(a)\}\\ \mathsf{gen}_{\mathsf{AE}}([b]^l) &:= AExp_b\end{aligned}$$

Formalizing the Analysis II

Example 14.8 ($\mathsf{kill}_{\mathsf{AE}}/\mathsf{gen}_{\mathsf{AE}}$ functions)

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

Example 14.8 ($\mathsf{kill}_{\mathsf{AE}}/\mathsf{gen}_{\mathsf{AE}}$ functions)

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

- $AExp_c = \{\texttt{a+b}, \texttt{a*b}, \texttt{a+1}\}$

Example 14.8 ($\mathsf{kill}_{\mathsf{AE}}/\mathsf{gen}_{\mathsf{AE}}$ functions)

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

- $AExp_c = \{\mathtt{a+b}, \mathtt{a*b}, \mathtt{a+1}\}$
-

Lab_c	$\mathsf{kill}_{\mathsf{AE}}(B^l)$	$\mathsf{gen}_{\mathsf{AE}}(B^l)$
1	$\emptyset$	$\{\mathtt{a+b}\}$
2	$\emptyset$	$\{\mathtt{a*b}\}$
3	$\emptyset$	$\{\mathtt{a+b}\}$
4	$\{\mathtt{a+b}, \mathtt{a*b}, \mathtt{a+1}\}$	$\emptyset$
5	$\emptyset$	$\{\mathtt{a+b}\}$