

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

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

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;
             y := z * y } c2 } c1 } cF } p
      end
    x := 2; y := 1; call F } c0
  end

```

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

Correction: Operational Semantics of Proc's II

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

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

- Update function for variable declarations takes store into account:

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

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

- (block) rule becomes

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

- **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:

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

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

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

- No modification of stores and update function required:

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

$$\text{upd}_v : \text{VDec} \times \text{VEnv} \rightarrow \text{VEnv}$$

$$\text{upd}_v(\text{var } x ; , \rho) := \rho[x \mapsto \min(\text{Loc} \setminus \rho(\text{Var}))]$$

- 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

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:

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

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

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;
```

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 $\text{init} : \text{Cmd} \rightarrow \text{Lab}$ returns the **initial label** of a statement:

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

The mapping $\text{final} : \text{Cmd} \rightarrow 2^{\text{Lab}}$ returns the set of **final labels** of a statement:

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

Definition 14.4 (Flow relation)

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

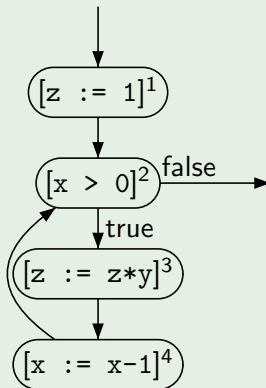
$$\begin{aligned}\text{flow}([\text{skip}]^l) &:= \emptyset \\ \text{flow}([x := a]^l) &:= \emptyset \\ \text{flow}(c_1; c_2) &:= \text{flow}(c_1) \cup \text{flow}(c_2) \cup \\ &\quad \{(l, \text{init}(c_2)) \mid l \in \text{final}(c_1)\} \\ \text{flow}(\text{if } [b]^l \text{ then } c_1 \text{ else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

Example 14.5

Visualization by **flow graph**:

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

```
init(c) = 1  
final(c) = {2}  
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

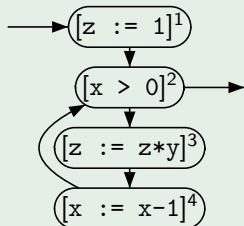
$$\{l \in Lab \mid (l, \text{init}(c)) \in \text{flow}(c)\} = \emptyset$$

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

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

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

Example 14.6



has an isolated entry but not isolated exits

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

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
- An expression a is **killed** in a block B if any of the variables in a is modified in B
- Formally: $kill_{AE} : Block_c \rightarrow 2^{AExp_c}$ is defined by
$$\begin{aligned}kill_{AE}([skip]^l) &:= \emptyset \\kill_{AE}([x := a]^l) &:= \{a' \in AExp_c \mid x \in FV(a')\} \\kill_{AE}([b]^l) &:= \emptyset\end{aligned}$$
- An expression a is **generated** in a block B if it is evaluated in and none of its variables are modified by B
- Formally: $gen_{AE} : Block_c \rightarrow 2^{AExp_c}$ is defined by
$$\begin{aligned}gen_{AE}([skip]^l) &:= \emptyset \\gen_{AE}([x := a]^l) &:= \{a \mid x \notin FV(a)\} \\gen_{AE}([b]^l) &:= AExp_b\end{aligned}$$

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

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

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

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