Program Analysis (470)
Language Syntax

Herbert Wiklicky
herbert@doc.ic.ac.uk

Department of Computing
Imperial College London

Spring 2015

Syntactic Constructs

We use the following syntactic categories:

$$\begin{array}{lcll} a & \in & \textbf{AExp} & \text{arithmetic expressions} \\ b & \in & \textbf{BExp} & \text{boolean expressions} \\ S & \in & \textbf{Stmt} & \text{statements} \end{array}$$

Abstract Syntax of WHILE

The syntax of the language WHILE is given by the following **abstract syntax**:

$$
\begin{array}{lcl}
a & ::= & x \mid n \mid a_1\ op_a\ a_2 \\
b & ::= & \mathit{true} \mid \mathit{false} \mid \mathit{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \\
S & ::= & x := a \\
 & & \mid \textbf{skip} \\
 & & \mid S_1 ; S_2 \\
 & & \mid \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \\
 & & \mid \textbf{while}\ b\ \textbf{do}\ S
\end{array}
$$

Syntactical Categories

We assume some countable/finite set of variables is given;

$$
\begin{array}{rcll}
x, y, z, \ldots & \in & \textbf{Var} & \text{variables} \\
n, m, \ldots & \in & \textbf{Num} & \text{numerals} \\
\ell, \ldots & \in & \textbf{Lab} & \text{labels}
\end{array}
$$

Numerals (integer constants) will not be further defined and neither will the operators:

$$
\begin{array}{rcll}
op_a & \in & \textbf{Op}_a & \text{arithmetic operators, e.g. } +, -, \times, \ldots \\
op_b & \in & \textbf{Op}_b & \text{boolean operators, e.g. } \wedge, \vee, \ldots \\
op_r & \in & \textbf{Op}_r & \text{relational operators, e.g. } =, <, \leq, \ldots
\end{array}
$$

Labelled Syntax of WHILE

The labelled syntax of the language WHILE is given by the following **abstract syntax**:

$$
\begin{array}{lcl}
a & ::= & x \mid n \mid a_1\ op_a\ a_2 \\
b & ::= & \mathit{true} \mid \mathit{false} \mid \mathit{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \\
S & ::= & [x := a]^\ell \\
 & & \mid [\textbf{skip}]^\ell \\
 & & \mid S_1; S_2 \\
 & & \mid \textbf{if}\ [b]^\ell\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \\
 & & \mid \textbf{while}\ [b]^\ell\ \textbf{do}\ S
\end{array}
$$

An Example in WHILE

An example of a program written in this WHILE language is the following one which computes the factorial of the number stored in x and leaves the result in z:

$$
\begin{array}{l}
[\ y := x\ ]^1; \\
[\ z := 1\ ]^2; \\
\textbf{while}\ [y > 1]^3\ \textbf{do}\ (\\
\qquad [\ z := z * y\ ]^4; \\
\qquad [\ y := y - 1\ ]^5); \\
[\ y := 0\ ]^6
\end{array}
$$

Note the use of meta-symbols, brackets, to group statements.

Concrete Syntax of WHILE

To avoid using brackets (as meta-symbols) we could also use the **concrete syntax** of the language WHILE as follows:

$$
\begin{array}{lcl}
a & ::= & x \mid n \mid a_1\ op_a\ a_2 \\
b & ::= & \mathit{true} \mid \mathit{false} \mid \mathit{not}\ b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \\
S & ::= & x := a \\
 & & \mid \textbf{skip} \\
 & & \mid S_1 ; S_2 \\
 & & \mid \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi} \\
 & & \mid \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}
\end{array}
$$

Sketches of a Formal Semantics [not for exam]

Memory is modelled by an abstract state, i.e. functions of type

$$\textbf{State} \quad = \quad \textbf{Var} \to \textbf{Z}.$$

For boolean and arithmetic expressions we assume that we know what they "evaluate to" in a state $s \in \textbf{State}$. Then the semantics for **AExp** is a *total* function

$$[\![\cdot]\!]_{\mathcal{A}}\ \cdot \quad : \quad \textbf{AExp} \to \textbf{State} \to \textbf{Z}$$

and the semantics of boolean expressions is given by

$$[\![\cdot]\!]_{\mathcal{B}}\ \cdot \quad : \quad \textbf{BExp} \to \textbf{State} \to \{\textbf{tt}, \textbf{ff}\}$$

Evaluating Expressions [not for exam]

Let us look at a program with two variables $\mathbf{Var} = \{x, y\}$.
Two possible states in this case could be for example:

$$s_0 = [x \mapsto 0, y \mapsto 1] \text{ and } s_1 = [x \mapsto 1, y \mapsto 1]$$

We can evaluate an expression like $x + y \in \mathbf{AExp}$:

$$\begin{aligned} [\![x + y]\!]_{\mathcal{A}} s_0 &= 0 + 1 = 1 \\ [\![x + y]\!]_{\mathcal{A}} s_1 &= 1 + 1 = 2 \end{aligned}$$

or a Boolean expression like $x + y \leq 1 \in \mathbf{BExp}$:

$$\begin{aligned} [\![x + y \leq 1]\!]_{\mathcal{B}} s_0 &= 1 \leq 1 = \mathbf{tt} \\ [\![x + y \leq 1]\!]_{\mathcal{B}} s_1 &= 2 \leq 1 = \mathbf{ff} \end{aligned}$$

Execution and Transitions [not for exam]

The configurations describe the current state of the execution.

$$\begin{aligned} \langle S, s \rangle &\quad \ldots \quad S \text{ is to be executed in state } s, \\ s &\quad \ldots \quad \text{a terminal state (i.e. } \langle ., s \rangle). \end{aligned}$$

The transition relation $\Rightarrow$ specify the (possible) computational steps during the execution starting from a certain configuration

$$\langle S, s \rangle \Rightarrow \langle S', s' \rangle$$

and at the end of the computation

$$\langle S, s \rangle \Rightarrow s'$$

Execution Rules (SOS) [not for exam]

$$\text{(ass)} \quad \langle [\mathtt{x} := a]^\ell, s\rangle \Rightarrow s[x \mapsto [\![a]\!]_{\mathcal{A}} s]$$

$$\text{(skip)} \quad \langle [\mathbf{skip}]^\ell, s\rangle \Rightarrow s$$

$$\text{(sq}^1\text{)} \quad \frac{\langle S_1, s\rangle \Rightarrow \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_1'; S_2, s'\rangle}$$

$$\text{(sq}^T\text{)} \quad \frac{\langle S_1, s\rangle \Rightarrow s'}{\langle S_1; S_2, s\rangle \Rightarrow \langle S_2, s'\rangle}$$

$$\text{(if}^T\text{)} \quad \langle \mathbf{if}\ [b]^\ell\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, s\rangle \Rightarrow \langle S_1, s\rangle \quad \text{if } [\![b]\!]_{\mathcal{B}} s = \mathbf{tt}$$

$$\text{(if}^F\text{)} \quad \langle \mathbf{if}\ [b]^\ell\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2, s\rangle \Rightarrow \langle S_2, s\rangle \quad \text{if } [\![b]\!]_{\mathcal{B}} s = \mathbf{ff}$$

$$\text{(wh}^T\text{)} \quad \langle \mathbf{while}\ [b]^\ell\ \mathbf{do}\ S, s\rangle \Rightarrow \langle S; \mathbf{while}\ [b]^\ell\ \mathbf{do}\ S, s\rangle \quad \text{if } [\![b]\!]_{\mathcal{B}} s = \mathbf{tt}$$

$$\text{(wh}^F\text{)} \quad \langle \mathbf{while}\ [b]^\ell\ \mathbf{do}\ S, s\rangle \Rightarrow s \quad \text{if } [\![b]\!]_{\mathcal{B}} s = \mathbf{ff}$$

A SOS Example [not for exam]

Consider a (perhaps rather vacuous) program like:

$$S \equiv [\, z := x + y \,]^\ell; \mathbf{while}\ [\mathit{true}]^{\ell'}\ \mathbf{do}\ [\ \mathbf{skip}\ ]^{\ell''}$$

$s_0 = [x \mapsto 0, y \mapsto 1, z \mapsto 0]$ and $s_1 = [x \mapsto 0, y \mapsto 1, z \mapsto 1]$

Then $\langle S, s_0\rangle$ executes as follows:

$$\begin{aligned}
\langle S, s_0\rangle &\Rightarrow \langle \mathbf{while}\ [\mathit{true}]^{\ell'}\ \mathbf{do}\ [\ \mathbf{skip}\ ]^{\ell''}, s_1\rangle \\
&\Rightarrow \langle [\ \mathbf{skip}\ ]^{\ell''}; \mathbf{while}\ [\mathit{true}]^{\ell'}\ \mathbf{do}\ [\ \mathbf{skip}\ ]^{\ell''}, s_1\rangle \\
&\Rightarrow \langle \mathbf{while}\ [\mathit{true}]^{\ell'}\ \mathbf{do}\ [\ \mathbf{skip}\ ]^{\ell''}, s_1\rangle \\
&\Rightarrow \langle [\ \mathbf{skip}\ ]^{\ell''}; \mathbf{while}\ [\mathit{true}]^{\ell'}\ \mathbf{do}\ [\ \mathbf{skip}\ ]^{\ell''}, s_1\rangle \\
&\Rightarrow \ldots
\end{aligned}$$

Initial Label

When presenting examples of Data Flow Analyses we will use a number of operations on programs and labels. The first of these is

$$init : \mathbf{Stmt} \to \mathbf{Lab}$$

which returns the initial label of a statement:

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

Final Labels

We will also need a function which returns the set of final labels in a statement; whereas a sequence of statements has a single entry, it may have multiple exits (e.g. in the conditional):

$$final : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$$

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

The **while**-loop terminates immediately after the test fails.

Elementary Blocks

The building blocks of our analysis is given by **Block** is the set of statements, or elementary blocks, of the form:

- $[\ x := a\ ]^\ell$, or
- $[\ \textbf{skip}\ ]^\ell$, as well as
- tests of the form $[b]^\ell$.

Blocks

To access the statements or test associated with a label in a program we use the function

$$blocks : \textbf{Stmt} \to \mathcal{P}(\textbf{Block})$$

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

Labels

Then the set of labels occurring in a program is given by

$$labels : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab})$$

where

$$labels(S) = \{\ell \mid [B]^\ell \in blocks(S)\}$$

Clearly $init(S) \in labels(S)$ and $final(S) \subseteq labels(S)$.

Flow

$$flow : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab})$$

which maps statements to sets of flows:

$$\begin{aligned}
flow([\ x := a\ ]^\ell) &= \emptyset \\
flow([\ \mathbf{skip}\ ]^\ell) &= \emptyset \\
flow(S_1;S_2) &= flow(S_1) \cup flow(S_2) \cup \\
&\quad \{(\ell, init(S_2)) \mid \ell \in final(S_1)\} \\
flow(\mathbf{if}\ [b]^\ell\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2) &= flow(S_1) \cup flow(S_2) \cup \\
&\quad \{(\ell, init(S_1)), (\ell, init(S_2))\} \\
flow(\mathbf{while}\ [b]^\ell\ \mathbf{do}\ S) &= flow(S) \cup \{(\ell, init(S))\} \cup \\
&\quad \{(\ell', \ell) \mid \ell' \in final(S)\}
\end{aligned}$$

An Example Flow

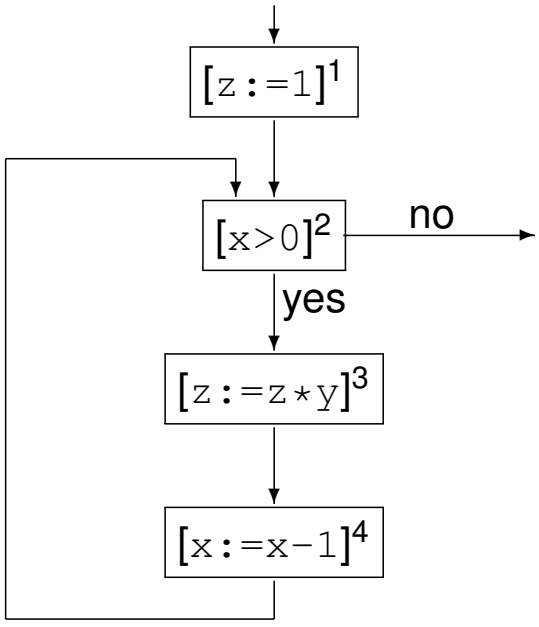
Consider the following program, power, computing the x-th power of the number stored in y:

$$
\begin{array}{l}
[\ z := 1\ ]^1; \\
\textbf{while}\ [x > 1]^2\ \textbf{do}\ (\\
\qquad [\ z := z * y\ ]^3; \\
\qquad [\ x := x - 1\ ]^4);
\end{array}
$$

We have $\mathit{labels}(\text{power}) = \{1, 2, 3, 4\}$, $\mathit{init}(\text{power}) = 1$, and $\mathit{final}(\text{power}) = \{2\}$. The function *flow* produces the set:

$$\mathit{flow}(\text{power}) = \{(1, 2), (2, 3), (3, 4), (4, 2)\}$$

Flow Graph



Forward Analysis

The function *flow* is used in the formulation of *forward analyses*. Clearly $init(S)$ is the (unique) entry node for the flow graph with nodes $labels(S)$ and edges $flow(S)$. Also

$$\begin{aligned} labels(S) &= \{init(S)\} \cup \\ &\quad \{\ell \mid (\ell, \ell') \in flow(S)\} \cup \\ &\quad \{\ell' \mid (\ell, \ell') \in flow(S)\} \end{aligned}$$

and for composite statements (meaning those not simply of the form $[B]^\ell$) the equation remains true when removing the $\{init(S)\}$ component.

Reverse Flow

In order to formulate *backward analyses* we require a function that computes reverse flows:

$$flow^R : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab} \times \mathbf{Lab})$$

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

For the power program, $flow^R$ produces

$$\{(2,1), (2,4), (3,2), (4,3)\}$$

Backward Analysis

In case $final(S)$ contains just one element that will be the unique entry node for the flow graph with nodes $labels(S)$ and edges $flow^R(S)$. Also

$$\begin{aligned} labels(S) \quad = \quad & final(S) \ \cup \\ & \{\ell \mid (\ell, \ell') \in flow^R(S)\} \ \cup \\ & \{\ell' \mid (\ell, \ell') \in flow^R(S)\} \end{aligned}$$

Notation

We will use the notation $S_\star$ to represent the program we are analysing (the “top-level" statement) and furthermore:

- $\mathbf{Lab}_\star$ to represent the labels ($labels(S_\star)$) appearing in $S_\star$,
- $\mathbf{Var}_\star$ to represent the variables ($FV(S_\star)$) appearing in $S_\star$,
- $\mathbf{Block}_\star$ to represent the elementary blocks ($blocks(S_\star)$) occurring in $S_\star$, and
- $\mathbf{AExp}_\star$ to represent the set of *non-trivial* arithmetic subexpressions in $S_\star$ as well as
- $\mathbf{AExp}(a)$ and $\mathbf{AExp}(b)$ to refer to the set of non-trivial arithmetic subexpressions of a given arithmetic, respectively boolean, expression.

An expression is trivial if it is a single variable or constant.

Isolated Entries & Exits

Program $S_\star$ has *isolated entries* if:

$$\forall \ell \in \mathbf{Lab} : (\ell, \mathit{init}(S_\star)) \notin \mathit{flow}(S_\star)$$

This is the case whenever $S_\star$ does not start with a **while**-loop.

Similarly, we shall frequently assume that the program $S_\star$ has *isolated exits*; this means that:

$$\forall \ell_1 \in \mathit{final}(S_\star)\ \forall \ell_2 \in \mathbf{Lab} : (\ell_1, \ell_2) \notin \mathit{flow}(S_\star)$$

Label Consistency

A statement, S, is label consistent if and only if:

$$[B_1]^\ell, [B_2]^\ell \in \mathit{blocks}(S) \text{ implies } B_1 = B_2$$

Clearly, if all blocks in S are uniquely labelled (meaning that each label occurs only once), then S is label consistent.

When S is label consistent the statement or clause "where $[B]^\ell \in \mathit{blocks}(S)$" is unambiguous in defining a partial function from labels to elementary blocks; we shall then say that ℓ labels the block B.