

Program Analysis (70020)

While Language

Herbert Wiklicky

Department of Computing
Imperial College London

herbert@doc.ic.ac.uk
h.wiklicky@imperial.ac.uk

Autumn 2023

1/21

Syntactic Constructs

We use the following syntactic categories:

$a \in \mathbf{AExp}$ arithmetic expressions
 $b \in \mathbf{BExp}$ boolean expressions
 $S \in \mathbf{Stmt}$ statements

2/21

Abstract Syntax of WHILE

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

$$\begin{aligned} a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\ b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\ S & ::= x := a \\ & \quad | \text{skip} \\ & \quad | S_1 ; S_2 \\ & \quad | \text{if } b \text{ then } S_1 \text{ else } S_2 \\ & \quad | \text{while } b \text{ do } S \end{aligned}$$

3/21

Syntactical Categories

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

$$\begin{aligned} x, y, z, \dots & \in \mathbf{Var} && \text{variables} \\ n, m, \dots & \in \mathbf{Num} && \text{numerals} \\ l, \dots & \in \mathbf{Lab} && \text{labels} \end{aligned}$$

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

$$\begin{aligned} \text{op}_a & \in \mathbf{Op}_a && \text{arithmetic operators, e.g. } +, -, \times, \dots \\ \text{op}_b & \in \mathbf{Op}_b && \text{boolean operators, e.g. } \wedge, \vee, \dots \\ \text{op}_r & \in \mathbf{Op}_r && \text{relational operators, e.g. } =, <, \leq, \dots \end{aligned}$$

4/21

Labelled Syntax of WHILE

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

$$\begin{aligned} a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\ b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\ S & ::= [x := a]^\ell \\ & \quad | [\text{skip}]^\ell \\ & \quad | S_1; S_2 \\ & \quad | \text{if } [b]^\ell \text{ then } S_1 \text{ else } S_2 \\ & \quad | \text{while } [b]^\ell \text{ do } S \end{aligned}$$

5/21

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 :

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

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

6/21

Concrete Syntax of WHILE

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

$$\begin{aligned} a & ::= x \mid n \mid a_1 \text{ op}_a a_2 \\ b & ::= \text{true} \mid \text{false} \mid \text{not } b \mid b_1 \text{ op}_b b_2 \mid a_1 \text{ op}_r a_2 \\ S & ::= x := a \\ & \quad | \text{skip} \\ & \quad | S_1 ; S_2 \\ & \quad | \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ fi} \\ & \quad | \text{while } b \text{ do } S \text{ od} \end{aligned}$$

7/21

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

$$\text{init} : \text{Stmt} \rightarrow \text{Lab}$$

which returns the initial label of a statement:

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

8/21

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):

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

$$final([x := a]^\ell) = \{\ell\}$$

$$final([\mathbf{skip}]^\ell) = \{\ell\}$$

$$final(S_1; S_2) = final(S_2)$$

$$final(\mathbf{if} [b]^\ell \mathbf{then} S_1 \mathbf{else} S_2) = final(S_1) \cup final(S_2)$$

$$final(\mathbf{while} [b]^\ell \mathbf{do} S) = \{\ell\}$$

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

9/21

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
- ▶ $[\mathbf{skip}]^\ell$, as well as
- ▶ tests of the form $[b]^\ell$.

10/21

Blocks

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

$$\mathit{blocks} : \mathbf{Stmt} \rightarrow \mathcal{P}(\mathbf{Block})$$

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

$$\mathit{blocks}([\mathbf{skip}]^\ell) = \{[\mathbf{skip}]^\ell\}$$

$$\mathit{blocks}(S_1; S_2) = \mathit{blocks}(S_1) \cup \mathit{blocks}(S_2)$$

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

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

11/21

Labels

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

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

where

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

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

12/21

Flow

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

which maps statements to sets of flows:

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

$$flow([\mathbf{skip}]^\ell) = \emptyset$$

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

$$flow(\mathbf{if} [b]^\ell \mathbf{then} S_1 \mathbf{else} S_2) = flow(S_1) \cup flow(S_2) \cup \{(\ell, init(S_1)), (\ell, init(S_2))\}$$

$$flow(\mathbf{while} [b]^\ell \mathbf{do} S) = flow(S) \cup \{(\ell, init(S))\} \cup \{(\ell', \ell) \mid \ell' \in final(S)\}$$

13/21

An Example Flow

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

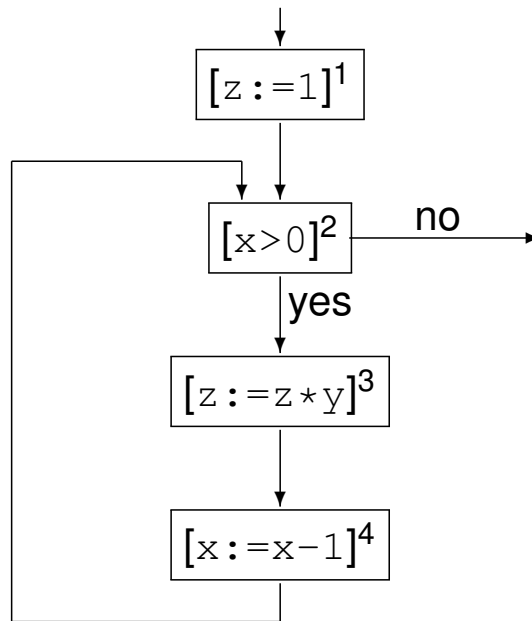
```
[ z := 1 ]1;  
while [ x > 1 ]2 do (  
    [ z := z * y ]3;  
    [ x := x - 1 ]4)
```

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

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

14/21

Flow Graph



15/21

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

$$labels(S) = \{init(S)\} \cup \{l \mid (l, l') \in flow(S)\} \cup \{l' \mid (l, l') \in flow(S)\}$$

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

16/21

Reverse Flow

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

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

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

For the power program, flow^R produces

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

17/21

Backward Analysis

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

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

18/21

Notation

We will use the notation S_* to represent the program we are analysing (the “top-level” statement) and furthermore:

- ▶ **Lab** $_*$ to represent the labels ($labels(S_*)$) appearing in S_* ,
- ▶ **Var** $_*$ to represent the variables ($FV(S_*)$) appearing in S_* ,
- ▶ **Block** $_*$ to represent the elementary blocks ($blocks(S_*)$) occurring in S_* , and
- ▶ **AExp** $_*$ to represent the set of *non-trivial* arithmetic subexpressions in S_* as well as
- ▶ **AExp**(a) and **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.

19/21

Isolated Entries & Exits

Program S_* has *isolated entries* if:

$$\forall l \in \mathbf{Lab} : (l, init(S_*)) \notin flow(S_*)$$

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

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

$$\forall l_1 \in final(S_*) \forall l_2 \in \mathbf{Lab} : (l_1, l_2) \notin flow(S_*)$$

20/21

Label Consistency

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

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

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 $[B]^\ell \in \text{blocks}(S)$ ” is unambiguous in defining a partial function from labels to elementary blocks; we shall then say that ℓ **labels** the block B .