Syntactic Constructs

We use the following syntactic categories:

\[
\begin{align*}
    a & \in \text{AExp} & \text{arithmetic expressions} \\
    b & \in \text{BExp} & \text{boolean expressions} \\
    S & \in \text{Stmt} & \text{statements}
\end{align*}
\]
Abstract Syntax of WHILE

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

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

Syntactical Categories

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

\[
\begin{align*}
  x, y, z, \ldots &\in \text{Var} \quad \text{variables} \\
  n, m, \ldots &\in \text{Num} \quad \text{numerals} \\
  \ell, \ldots &\in \text{Lab} \quad \text{labels}
\end{align*}
\]

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

\[
\begin{align*}
  \text{op}_a &\in \text{Op}_a \quad \text{arithmetic operators, e.g. } +, -, \times, \ldots \\
  \text{op}_b &\in \text{Op}_b \quad \text{boolean operators, e.g. } \land, \lor, \ldots \\
  \text{op}_r &\in \text{Op}_r \quad \text{relational operators, e.g. } =, <, \leq, \ldots
\end{align*}
\]
Labelled Syntax of WHILE

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

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

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 \( z \):

\[ [y := x]^1; \]
\[ [z := 1]^2; \]
\[ \text{while } [y > 1]^3 \text{ do } ( \]
\[ [z := z \ast y]^4; \]
\[ [y := y - 1]^5); \]
\[ [y := 0]^6 \]

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:

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

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:

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

\[
\text{final} : \text{Stmt} \rightarrow \mathcal{P} (\text{Lab})
\]

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

\[
\text{final}([ \; \text{skip} ]^\ell) = \{ \ell \}
\]

\[
\text{final}(S_1; S_2) = \text{final}(S_2)
\]

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

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

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
- \([ \; \text{skip} ]^\ell\), 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

\[\text{blocks} : \text{Stmt} \rightarrow \mathcal{P}(\text{Block})\]

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

Labels

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

\[\text{labels} : \text{Stmt} \rightarrow \mathcal{P}(\text{Lab})\]

where

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

Clearly \(\text{init}(S) \in \text{labels}(S)\) and \(\text{final}(S) \subseteq \text{labels}(S)\).
Flow

\[ \text{flow} : \text{Stmt} \rightarrow \mathcal{P}(\text{Lab} \times \text{Lab}) \]

which maps statements to sets of flows:

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

An Example Flow

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

\[
\begin{align*}
[ &z := 1]^1; \\
\text{while } [x > 1]^2 \text{ do } ( &z := z * y]^3; \\
[ &x := x - 1]^4); \\
\end{align*}
\]

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

\[
\text{flow}(\text{power}) = \{(1, 2), (2, 3), (3, 4), (4, 2)\}
\]
The function $\text{flow}$ is used in the formulation of forward analyses. Clearly $\text{init}(S)$ is the (unique) entry node for the flow graph with nodes $\text{labels}(S)$ and edges $\text{flow}(S)$. Also

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

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

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

\[ \text{flow}^R : \text{Stmt} \rightarrow \mathcal{P}(\text{Lab} \times \text{Lab}) \]

\[ \text{flow}^R(S) = \{ (\ell, \ell') | (\ell', \ell) \in \text{flow}(S) \} \]

For the power program, \( \text{flow}^R \) produces

\[ \{ (2, 1), (2, 4), (3, 2), (4, 3) \} \]

Backward Analysis

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

\[ \text{labels}(S) = \text{final}(S) \cup \{ \ell | (\ell, \ell') \in \text{flow}^R(S) \} \cup \{ \ell' | (\ell, \ell') \in \text{flow}^R(S) \} \]
Notation

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

- $\text{Lab}_\star$ to represent the labels ($\text{labels}(S_\star)$) appearing in $S_\star$,
- $\text{Var}_\star$ to represent the variables ($\text{FV}(S_\star)$) appearing in $S_\star$,
- $\text{Block}_\star$ to represent the elementary blocks ($\text{blocks}(S_\star)$) occurring in $S_\star$, and
- $\text{AExp}_\star$ to represent the set of non-trivial arithmetic subexpressions in $S_\star$ as well as
- $\text{AExp}(a)$ and $\text{AExp}(b)$ 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:

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

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:

$$\forall \ell_1 \in \text{final}(S_\star) \forall \ell_2 \in \text{Lab} : (\ell_1, \ell_2) \notin \text{flow}(S_\star)$$
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 $\ell$ labels the block $B$. 