We use the following syntactic categories:

a	$\in$	**AExp**	arithmetic expressions
b	$\in$	**BExp**	boolean expressions
S	$\in$	**Stmt**	statements

We assume some countable set of variables is given; numerals and labels will not be further defined and neither will the operators:

x, y	$\in$	**Var**	variables
n	$\in$	**Num**	numerals
ℓ	$\in$	**Lab**	labels

op_a	$\in$	$\mathbf{Op}_a$	arithmetic operators
op_b	$\in$	$\mathbf{Op}_b$	boolean operators
op_r	$\in$	$\mathbf{Op}_r$	relational operators

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

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

Initial and final labels. 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:

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

We will also need a function

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

which returns the set of final labels in a statement; whereas a sequence of statements has a single entry, it may have multiple exits (as for example in the conditional):

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

Note that the `while`-loop terminates immediately after the test has evaluated to false.

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

$$blocks : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Blocks})$$

where **Blocks** is the set of statements, or elementary blocks, of the form $[x\texttt{:=}a]^\ell$ or $[\texttt{skip}]^\ell$ as well as the set of tests of the form $[b]^\ell$.

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

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 $final(S) \subseteq labels(S)$.

Flows and reverse flows.

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

which maps statements to sets of flows:

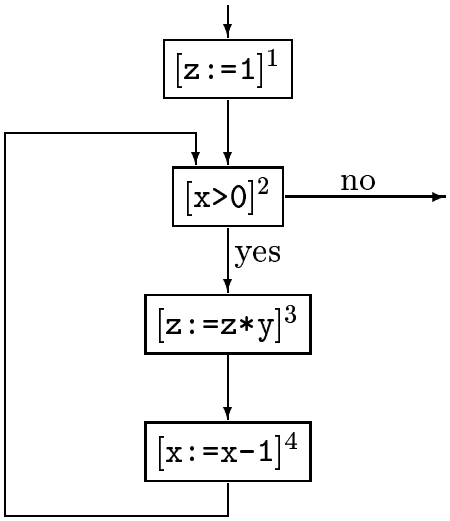
$$\begin{aligned}
flow([x := a]^\ell) &= \emptyset \\
flow([\texttt{skip}]^\ell) &= \emptyset \\
flow(S_1; S_2) &= flow(S_1) \cup flow(S_2) \cup \\
&\quad \{(\ell, init(S_2)) \mid \ell \in final(S_1)\} \\
flow(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2) &= flow(S_1) \cup flow(S_2)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

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

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

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

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



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

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

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

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

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

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

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

We will use the notation $S_\star$ to represent the program that we are analysing (the "top-level" statement), $\mathbf{Lab}_\star$ to represent the labels ($labels(S)$) appearing in $S_\star$, $\mathbf{Var}_\star$ to represent the variables ($FV(S_\star)$) appearing in $S_\star$, $\mathbf{Blocks}_\star$ 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 $S_\star$; an expression is trivial if it is a single variable or constant. We will also write $\mathbf{AExp}(a)$ and $\mathbf{AExp}(b)$ to refer to the set of non-trivial arithmetic subexpressions of a given arithmetic, respectively boolean, expression.

Program $S_\star$ has isolated entries if:

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

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:

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

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 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.

Available Expressions Analysis

The Available Expressions Analysis will determine:

> For each program point, which expressions *must* have already been computed, and not later modified, on all paths to the program point.

This information can be used to avoid the re-computation of an expression. For clarity, we will concentrate on arithmetic expressions.

$[\texttt{x:=a+b}]^1; [\texttt{y:=a*b}]^2; \texttt{while } [\texttt{y>a+b}]^3 \texttt{ do } ([\texttt{a:=a+1}]^4; [\texttt{x:=a+b}]^5)$

It should be clear that the expression `a+b` is available every time execution reaches the test (label 3) in the loop; as a consequence, the expression need not be recomputed.

$$kill_{\mathsf{AE}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

$$gen_{\mathsf{AE}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

$$\mathsf{AE}_{entry}, \mathsf{AE}_{exit} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

$$\begin{array}{rcl}
kill_{\mathsf{AE}}([x := a]^\ell) & = & \{a' \in \mathbf{AExp}_\star \mid x \in FV(a')\} \\
kill_{\mathsf{AE}}([\mathtt{skip}]^\ell) & = & \emptyset \\
kill_{\mathsf{AE}}([b]^\ell) & = & \emptyset \\
gen_{\mathsf{AE}}([x := a]^\ell) & = & \{a' \in \mathbf{AExp}(a) \mid x \notin FV(a')\} \\
gen_{\mathsf{AE}}([\mathtt{skip}]^\ell) & = & \emptyset \\
gen_{\mathsf{AE}}([b]^\ell) & = & \mathbf{AExp}(b)
\end{array}$$

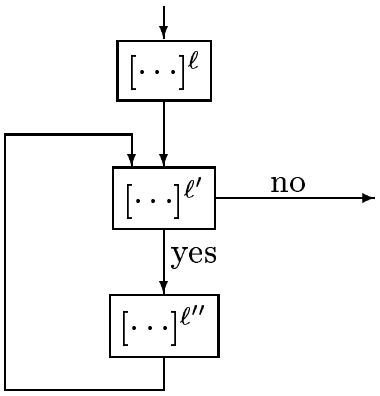
$$\mathsf{AE}_{entry}(\ell) = \begin{cases} \emptyset, \text{ if } \ell = init(S_\star) \\ \bigcap\{\mathsf{AE}_{exit}(\ell') \mid (\ell', \ell) \in flow(S_\star)\}, \text{ otherwise} \end{cases}$$

$$\begin{array}{rcl}
\mathsf{AE}_{exit}(\ell) & = & (\mathsf{AE}_{entry}(\ell) \backslash kill_{\mathsf{AE}}(B^\ell)) \cup gen_{\mathsf{AE}}(B^\ell) \\
 & & \text{where } B^\ell \in blocks(S_\star)
\end{array}$$

Largest solution. The analysis is a *forward analysis* and, as we shall see, we are interested in the *largest* sets satisfying the equation for AE_{entry}.

$$[\texttt{z:=x+y}]^{\ell};\texttt{while}\ [\texttt{true}]^{\ell'}\ \texttt{do}\ [\texttt{skip}]^{\ell''}$$

$$\begin{aligned}
\mathsf{AE}_{entry}(\ell) &= \emptyset \\
\mathsf{AE}_{entry}(\ell') &= \mathsf{AE}_{exit}(\ell) \cap \mathsf{AE}_{exit}(\ell'') \\
\mathsf{AE}_{entry}(\ell'') &= \mathsf{AE}_{exit}(\ell') \\
\mathsf{AE}_{exit}(\ell) &= \mathsf{AE}_{entry}(\ell) \cup \{\texttt{x+y}\} \\
\mathsf{AE}_{exit}(\ell') &= \mathsf{AE}_{entry}(\ell') \\
\mathsf{AE}_{exit}(\ell'') &= \mathsf{AE}_{entry}(\ell'')
\end{aligned}$$



After some simplification, we find that:

$$\mathsf{AE}_{entry}(\ell') = \{\texttt{x+y}\} \cap \mathsf{AE}_{entry}(\ell')$$

$[\texttt{x:=a+b}]^1; [\texttt{y:=a*b}]^2; \texttt{while } [\texttt{y>a+b}]^3 \texttt{ do } ([\texttt{a:=a+1}]^4; [\texttt{x:=a+b}]^5)$

ℓ	$kill_{\mathsf{AE}}(\ell)$	$gen_{\mathsf{AE}}(\ell)$
1	$\emptyset$	$\{\texttt{a+b}\}$
2	$\emptyset$	$\{\texttt{a*b}\}$
3	$\emptyset$	$\{\texttt{a+b}\}$
4	$\{\texttt{a+b}, \texttt{a*b}, \texttt{a+1}\}$	$\emptyset$
5	$\emptyset$	$\{\texttt{a+b}\}$

$[\texttt{x:=a+b}]^1; [\texttt{y:=a*b}]^2; \texttt{while } [\texttt{y>a+b}]^3 \texttt{ do } ([\texttt{a:=a+1}]^4; [\texttt{x:=a+b}]^5)$

$$
\begin{aligned}
\mathsf{AE}_{entry}(1) &= \emptyset \\
\mathsf{AE}_{entry}(2) &= \mathsf{AE}_{exit}(1) \\
\mathsf{AE}_{entry}(3) &= \mathsf{AE}_{exit}(2) \cap \mathsf{AE}_{exit}(5) \\
\mathsf{AE}_{entry}(4) &= \mathsf{AE}_{exit}(3) \\
\mathsf{AE}_{entry}(5) &= \mathsf{AE}_{exit}(4) \\
\mathsf{AE}_{exit}(1) &= \mathsf{AE}_{entry}(1) \cup \{\texttt{a+b}\} \\
\mathsf{AE}_{exit}(2) &= \mathsf{AE}_{entry}(2) \cup \{\texttt{a*b}\} \\
\mathsf{AE}_{exit}(3) &= \mathsf{AE}_{entry}(3) \cup \{\texttt{a+b}\} \\
\mathsf{AE}_{exit}(4) &= \mathsf{AE}_{entry}(4) \setminus \{\texttt{a+b}, \texttt{a*b}, \texttt{a+1}\} \\
\mathsf{AE}_{exit}(5) &= \mathsf{AE}_{entry}(5) \cup \{\texttt{a+b}\}
\end{aligned}
$$

ℓ	$\mathsf{AE}_{entry}(\ell)$	$\mathsf{AE}_{exit}(\ell)$
1	$\emptyset$	{a+b}
2	{a+b}	{a+b, a*b}
3	{a+b}	{a+b}
4	{a+b}	$\emptyset$
5	$\emptyset$	{a+b}

Note that, even though `a` is redefined in the loop, the expression `a+b` is re-evaluated in the loop and so it is always available on entry to the loop. On the other hand, `a*b` is available on the first entry to the loop but is killed before the next iteration.

Reaching Definitions Analysis. This analysis is analogous to the previous one except that we are interested in:

> For each program point, which assignments *may* have been made and not overwritten, when program execution reaches this point along some path.

A main application of Reaching Definitions Analysis is in the construction of direct links between blocks that produce values and blocks that use them.

$[\texttt{x:=5}]^1; [\texttt{y:=1}]^2; \texttt{while } [\texttt{x>1}]^3 \texttt{ do } ([\texttt{y:=x*y}]^4; [\texttt{x:=x-1}]^5)$

All of the assignments reach the entry of 4 (the assignments labelled 1 and 2 reach there on the first iteration); only the assignments labelled 1, 4 and 5 reach the entry of 5.

$$kill_{\mathsf{RD}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star)$$

$$gen_{\mathsf{RD}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star)$$

$$\mathsf{RD}_{entry}, \mathsf{RD}_{exit} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star)$$

$$\begin{aligned}
kill_{\mathsf{RD}}([x := a]^\ell) &= \{(x, ?)\} \cup \\
&\quad \{(x, \ell') \mid B^{\ell'} \text{ is an assignment to } x \text{ in } S_\star\} \\
kill_{\mathsf{RD}}([\mathtt{skip}]^\ell) &= \emptyset \\
kill_{\mathsf{RD}}([b]^\ell) &= \emptyset \\
gen_{\mathsf{RD}}([x := a]^\ell) &= \{(x, \ell)\} \\
gen_{\mathsf{RD}}([\mathtt{skip}]^\ell) &= \emptyset \\
gen_{\mathsf{RD}}([b]^\ell) &= \emptyset
\end{aligned}$$

$$
\begin{array}{rcl}
\mathsf{RD}_{entry}(\ell) & = & \begin{cases} \{(x,?) \mid x \in FV(S_\star)\}, \text{ if } \ell = init(S_\star) \\ \bigcup\{\mathsf{RD}_{exit}(\ell') \mid (\ell',\ell) \in flow(S_\star)\}, \text{ otherwise} \end{cases} \\
\mathsf{RD}_{exit}(\ell) & = & (\mathsf{RD}_{entry}(\ell) \backslash kill_{\mathsf{RD}}(B^\ell)) \cup gen_{\mathsf{RD}}(B^\ell) \\
 & & \text{where } B^\ell \in blocks(S_\star)
\end{array}
$$

Smallest solution. Similar to the previous example, this is a *forward analysis* but, as we shall see, we are interested in the *smallest* sets satisfying the equation for RD_{entry}.

$$[\mathtt{z:=x+y}]^\ell; \mathtt{while}\ [\mathtt{true}]^{\ell'}\ \mathtt{do}\ [\mathtt{skip}]^{\ell''}$$

$$
\begin{array}{rcl}
\mathsf{RD}_{entry}(\ell) & = & \{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?)\} \\
\mathsf{RD}_{entry}(\ell') & = & \mathsf{RD}_{exit}(\ell) \cup \mathsf{RD}_{exit}(\ell'') \\
\mathsf{RD}_{entry}(\ell'') & = & \mathsf{RD}_{exit}(\ell')
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{RD}_{exit}(\ell) & = & (\mathsf{RD}_{entry}(\ell) \setminus \{(\mathtt{z},?)\}) \cup \{(\mathtt{z},\ell)\} \\
\mathsf{RD}_{exit}(\ell') & = & \mathsf{RD}_{entry}(\ell') \\
\mathsf{RD}_{exit}(\ell'') & = & \mathsf{RD}_{entry}(\ell'')
\end{array}
$$

$$\mathsf{RD}_{entry}(\ell') = \{(\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},\ell)\} \cup \mathsf{RD}_{entry}(\ell')$$

Sometimes, when the Reaching Definitions Analysis is presented in the literature, one has $\mathsf{RD}_{entry}(init(S_\star)) = \emptyset$ rather than $\mathsf{RD}_{entry}(init(S_\star)) = \{(x, ?) \mid x \in FV(S_\star)\}$. This is correct only for programs that always assign variables before their first use; incorrect optimisations may result if this is not the case. The advantage of our formulation is that it is always semantically sound.

$[\texttt{x:=5}]^1; [\texttt{y:=1}]^2; \texttt{while } [\texttt{x>1}]^3 \texttt{ do } ([\texttt{y:=x*y}]^4; [\texttt{x:=x-1}]^5)$

ℓ	$kill_{\mathsf{RD}}(\ell)$	$gen_{\mathsf{RD}}(\ell)$
1	$\{(\texttt{x}, ?), (\texttt{x}, 1), (\texttt{x}, 5)\}$	$\{(\texttt{x}, 1)\}$
2	$\{(\texttt{y}, ?), (\texttt{y}, 2), (\texttt{y}, 4)\}$	$\{(\texttt{y}, 2)\}$
3	$\emptyset$	$\emptyset$
4	$\{(\texttt{y}, ?), (\texttt{y}, 2), (\texttt{y}, 4)\}$	$\{(\texttt{y}, 4)\}$
5	$\{(\texttt{x}, ?), (\texttt{x}, 1), (\texttt{x}, 5)\}$	$\{(\texttt{x}, 5)\}$

$$[\texttt{x:=5}]^1;[\texttt{y:=1}]^2;\texttt{while } [\texttt{x>1}]^3 \texttt{ do } ([\texttt{y:=x*y}]^4;[\texttt{x:=x-1}]^5)$$

$$\begin{aligned}
\mathsf{RD}_{entry}(1) &= \{(\texttt{x},?),(\texttt{y},?)\} \\
\mathsf{RD}_{entry}(2) &= \mathsf{RD}_{exit}(1) \\
\mathsf{RD}_{entry}(3) &= \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(5) \\
\mathsf{RD}_{entry}(4) &= \mathsf{RD}_{exit}(3) \\
\mathsf{RD}_{entry}(5) &= \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{exit}(1) &= (\mathsf{RD}_{entry}(1) \setminus \{(\texttt{x},?),(\texttt{x},1),(\texttt{x},5)\}) \cup \{(\texttt{x},1)\} \\
\mathsf{RD}_{exit}(2) &= (\mathsf{RD}_{entry}(2) \setminus \{(\texttt{y},?),(\texttt{y},2),(\texttt{y},4)\}) \cup \{(\texttt{y},2)\} \\
\mathsf{RD}_{exit}(3) &= \mathsf{RD}_{entry}(3) \\
\mathsf{RD}_{exit}(4) &= (\mathsf{RD}_{entry}(4) \setminus \{(\texttt{y},?),(\texttt{y},2),(\texttt{y},4)\}) \cup \{(\texttt{y},4)\} \\
\mathsf{RD}_{exit}(5) &= (\mathsf{RD}_{entry}(5) \setminus \{(\texttt{x},?),(\texttt{x},1),(\texttt{x},5)\}) \cup \{(\texttt{x},5)\}
\end{aligned}$$

ℓ	$\mathsf{RD}_{entry}(\ell)$	$\mathsf{RD}_{exit}(\ell)$
1	$\{(\texttt{x},?),(\texttt{y},?)\}$	$\{(\texttt{y},?),(\texttt{x},1)\}$
2	$\{(\texttt{y},?),(\texttt{x},1)\}$	$\{(\texttt{x},1),(\texttt{y},2)\}$
3	$\{(\texttt{x},1),(\texttt{y},2),(\texttt{y},4),(\texttt{x},5)\}$	$\{(\texttt{x},1),(\texttt{y},2),(\texttt{y},4),(\texttt{x},5)\}$
4	$\{(\texttt{x},1),(\texttt{y},2),(\texttt{y},4),(\texttt{x},5)\}$	$\{(\texttt{x},1),(\texttt{y},4),(\texttt{x},5)\}$
5	$\{(\texttt{x},1),(\texttt{y},4),(\texttt{x},5)\}$	$\{(\texttt{y},4),(\texttt{x},5)\}$

Very Busy Expressions Analysis. An expression is *very busy* at the exit from a label if, no matter what path is taken from the label, the expression must always be used before any of the variables occurring in it are redefined. The aim of the Very Busy Expressions Analysis is to determine:

> For each program point, which expressions *must* be very busy at the exit from the point.

A possible optimisation based on this information is to evaluate the expression at the block and store its value for later use; this optimisation is sometimes called hoisting the expression.

`if` $[$`a>b`$]^1$ `then` $([$`x:=b-a`$]^2; [$`y:=a-b`$]^3)$ `else` $([$`y:=b-a`$]^4; [$`x:=a-b`$]^5)$

$$kill_{\mathsf{VB}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

$$gen_{\mathsf{VB}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

$$\mathsf{VB}_{entry}, \mathsf{VB}_{exit} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{AExp}_\star)$$

The analysis is a *backward analysis* and we are interested in the *largest* sets satisfying the equation for VB_{exit}.

$$
\begin{array}{rcl}
kill_{\mathsf{VB}}([x := a]^\ell) & = & \{a' \in \mathbf{AExp}_\star \mid x \in FV(a')\} \\
kill_{\mathsf{VB}}([\mathtt{skip}]^\ell) & = & \emptyset \\
kill_{\mathsf{VB}}([b]^\ell) & = & \emptyset \\
gen_{\mathsf{VB}}([\mathtt{x} := a]^\ell) & = & \mathbf{AExp}(a) \\
gen_{\mathsf{VB}}([\mathtt{skip}]^\ell) & = & \emptyset \\
gen_{\mathsf{VB}}([b]^\ell) & = & \mathbf{AExp}(b)
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{VB}_{exit}(\ell) & = & \begin{cases} \emptyset, \text{if } \ell \in final(S_\star) \\ \bigcap\{\mathsf{VB}_{entry}(\ell') \mid (\ell', \ell) \in flow^R(S_\star)\}, \text{otherwise} \end{cases} \\
\mathsf{VB}_{entry}(\ell) & = & (\mathsf{VB}_{exit}(\ell) \setminus kill_{\mathsf{VB}}(B^\ell)) \cup gen_{\mathsf{VB}}(B^\ell) \\
 & & \text{where } B^\ell \in blocks(S_\star)
\end{array}
$$

$\mathtt{if}\ [\mathtt{a>b}]^1\ \mathtt{then}\ ([\mathtt{x:=b-a}]^2; [\mathtt{y:=a-b}]^3)\ \mathtt{else}\ ([\mathtt{y:=b-a}]^4; [\mathtt{x:=a-b}]^5)$

ℓ	$kill_{\mathsf{VB}}(\ell)$	$gen_{\mathsf{VB}}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	{b-a}
3	$\emptyset$	{a-b}
4	$\emptyset$	{b-a}
5	$\emptyset$	{a-b}

$\texttt{if}\ [\texttt{a>b}]^1\ \texttt{then}\ ([\texttt{x:=b-a}]^2; [\texttt{y:=a-b}]^3)\ \texttt{else}\ ([\texttt{y:=b-a}]^4; [\texttt{x:=a-b}]^5)$

$$
\begin{aligned}
\mathsf{VB}_{entry}(1) &= \mathsf{VB}_{exit}(1) \\
\mathsf{VB}_{entry}(2) &= \mathsf{VB}_{exit}(2) \cup \{\texttt{b-a}\} \\
\mathsf{VB}_{entry}(3) &= \{\texttt{a-b}\} \\
\mathsf{VB}_{entry}(4) &= \mathsf{VB}_{exit}(4) \cup \{\texttt{b-a}\} \\
\mathsf{VB}_{entry}(5) &= \{\texttt{a-b}\} \\
\mathsf{VB}_{exit}(1) &= \mathsf{VB}_{entry}(2) \cap \mathsf{VB}_{entry}(4) \\
\mathsf{VB}_{exit}(2) &= \mathsf{VB}_{entry}(3) \\
\mathsf{VB}_{exit}(3) &= \emptyset \\
\mathsf{VB}_{exit}(4) &= \mathsf{VB}_{entry}(5) \\
\mathsf{VB}_{exit}(5) &= \emptyset
\end{aligned}
$$

ℓ	$\mathsf{VB}_{entry}(\ell)$	$\mathsf{VB}_{exit}(\ell)$
1	$\{\texttt{a-b}, \texttt{b-a}\}$	$\{\texttt{a-b}, \texttt{b-a}\}$
2	$\{\texttt{a-b}, \texttt{b-a}\}$	$\{\texttt{a-b}\}$
3	$\{\texttt{a-b}\}$	$\emptyset$
4	$\{\texttt{a-b}, \texttt{b-a}\}$	$\{\texttt{a-b}\}$
5	$\{\texttt{a-b}\}$	$\emptyset$

Live Variables Analysis. A variable is *live* at the exit from a label if there exists a path from the label to a use of the variable that does not re-define the variable. The Live Variables Analysis will determine:

> For each program point, which variables *may* be live at the exit from the point.

This analysis might be used as the basis for Dead Code Elimination. If the variable is not live at the exit from a label then, if the elementary block is an assignment to the variable, the elementary block can be eliminated.

> $[\texttt{x:=2}]^1; [\texttt{y:=4}]^2; [\texttt{x:=1}]^3;$
> $(\texttt{if } [\texttt{y>x}]^4 \texttt{ then } [\texttt{z:=y}]^5 \texttt{ else } ;[\texttt{z:=y*y}]^6); [\texttt{x:=z}]^7$

$$kill_{\mathsf{LV}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{Var}_\star)$$

$$gen_{\mathsf{LV}} : \mathbf{Blocks}_\star \to \mathcal{P}(\mathbf{Var}_\star)$$

$$\mathsf{LV}_{exit}, \mathsf{LV}_{entry} : \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Var}_\star)$$

The analysis is a *backward analysis* and we are interested in the *smallest* sets satisfying the equation for LV_{exit}.

$$
\begin{aligned}
kill_{\mathsf{LV}}([x := a]^\ell) &= \{x\} \\
kill_{\mathsf{LV}}([\mathtt{skip}]^\ell) &= \emptyset \\
kill_{\mathsf{LV}}([b]^\ell) &= \emptyset \\
gen_{\mathsf{LV}}([x := a]^\ell) &= FV(a) \\
gen_{\mathsf{LV}}([\mathtt{skip}]^\ell) &= \emptyset \\
gen_{\mathsf{LV}}([b]^\ell) &= FV(b)
\end{aligned}
$$

$$
\mathsf{LV}_{exit}(\ell) = \begin{cases} \emptyset, \text{if } \ell \in final(S_\star) \\ \bigcup\{\mathsf{LV}_{entry}(\ell') \mid (\ell', \ell) \in flow^R(S_\star)\}, \text{otherwise} \end{cases}
$$

$$
\mathsf{LV}_{entry}(\ell) = (\mathsf{LV}_{exit}(\ell) \setminus kill_{\mathsf{LV}}(B^\ell)) \cup gen_{\mathsf{LV}}(B^\ell)
$$

where $B^\ell \in blocks(S_\star)$

$[\mathtt{x:=2}]^1; [\mathtt{y:=4}]^2; [\mathtt{x:=1}]^3;$

$(\mathtt{if}\ [\mathtt{y>x}]^4\ \mathtt{then}\ [\mathtt{z:=y}]^5\ \mathtt{else}\ ;[\mathtt{z:=y*y}]^6); [\mathtt{x:=z}]^7$

ℓ	$kill_{\mathsf{LV}}(\ell)$	$gen_{\mathsf{LV}}(\ell)$
1	{x}	$\emptyset$
2	{y}	$\emptyset$
3	{x}	$\emptyset$
4	$\emptyset$	{x, y}
5	{z}	{y}
6	{z}	{y}
7	{x}	{z}

$$
\begin{aligned}
\mathsf{LV}_{entry}(1) &= \mathsf{LV}_{exit}(1) \backslash \{\mathtt{x}\} \\
\mathsf{LV}_{entry}(2) &= \mathsf{LV}_{exit}(2) \backslash \{\mathtt{y}\} \\
\mathsf{LV}_{entry}(3) &= \mathsf{LV}_{exit}(3) \backslash \{\mathtt{x}\} \\
\mathsf{LV}_{entry}(4) &= \mathsf{LV}_{exit}(4) \cup \{\mathtt{x},\mathtt{y}\} \\
\mathsf{LV}_{entry}(5) &= (\mathsf{LV}_{exit}(5) \backslash \{\mathtt{z}\}) \cup \{\mathtt{y}\} \\
\mathsf{LV}_{entry}(6) &= (\mathsf{LV}_{exit}(6) \backslash \{\mathtt{z}\}) \cup \{\mathtt{y}\} \\
\mathsf{LV}_{entry}(7) &= \{\mathtt{z}\}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{LV}_{exit}(1) &= \mathsf{LV}_{entry}(2) \\
\mathsf{LV}_{exit}(2) &= \mathsf{LV}_{entry}(3) \\
\mathsf{LV}_{exit}(3) &= \mathsf{LV}_{entry}(4) \\
\mathsf{LV}_{exit}(4) &= \mathsf{LV}_{entry}(5) \cup \mathsf{LV}_{entry}(6) \\
\mathsf{LV}_{exit}(5) &= \mathsf{LV}_{entry}(7) \\
\mathsf{LV}_{exit}(6) &= \mathsf{LV}_{entry}(7) \\
\mathsf{LV}_{exit}(7) &= \emptyset
\end{aligned}
$$

ℓ	$\mathsf{LV}_{entry}(\ell)$	$\mathsf{LV}_{exit}(\ell)$
1	$\emptyset$	$\emptyset$
2	$\emptyset$	$\{\mathtt{y}\}$
3	$\{\mathtt{y}\}$	$\{\mathtt{x},\mathtt{y}\}$
4	$\{\mathtt{x},\mathtt{y}\}$	$\{\mathtt{y}\}$
5	$\{\mathtt{y}\}$	$\{\mathtt{z}\}$
6	$\{\mathtt{y}\}$	$\{\mathtt{z}\}$
7	$\{\mathtt{z}\}$	$\emptyset$

Some authors assume that the variables of interest are output at the end of the program; in that case $\mathsf{LV}_{exit}(7)$ should be $\{\mathtt{x},\mathtt{y},\mathtt{z}\}$ which means that $\mathsf{LV}_{entry}(7)$, $\mathsf{LV}_{exit}(5)$ and $\mathsf{LV}_{exit}(6)$ should all be $\{\mathtt{y},\mathtt{z}\}$.

Derived Data Flow Information.

It is often convenient to directly link labels of statements that produce values to the labels of statements that use them. Links that, for each use of a variable, associate all assignments that reach that use are called *Use-Definition chains* or *ud*-chains. Links that, for each assignment, associate all uses are called *Definition-Use chains* or *du*-chains.

Definition clear paths:

$$\begin{aligned} clear(x,\ell,\ell') \quad = \quad & \exists \ell_1,\cdots,\ell_n : \\ & (\ell_1 = \ell) \wedge (\ell_n = \ell') \wedge (n > 0) \wedge \\ & (\forall i \in \{1,\cdots,n-1\} : (\ell_i,\ell_{i+1}) \in flow(S_\star)) \wedge \\ & (\forall i \in \{1,\cdots,n-1\} : \neg def(x,\ell_i)) \wedge use(x,\ell_n) \end{aligned}$$

$$use(x,\ell) = (\exists B : [B]^\ell \in blocks(S_\star) \wedge x \in gen_{\mathsf{LV}}([B]^\ell))$$

$$def(x,\ell) = (\exists B : [B]^\ell \in blocks(S_\star) \wedge x \in kill_{\mathsf{LV}}([B]^\ell))$$

$$ud, du : \mathbf{Var}_\star \times \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Lab}_\star)$$

$$\begin{aligned} ud(x,\ell') \quad = \quad & \{\ell \mid def(x,\ell) \wedge \exists \ell'' : (\ell,\ell'') \in flow(S_\star) \\ & \wedge clear(x,\ell'',\ell')\} \\ \cup \quad & \{? \mid clear(x, init(S_\star), \ell')\} \end{aligned}$$

$$du(x,\ell) = \begin{cases} \{\ell' \mid def(x,\ell) \wedge \\ \exists \ell'' : (\ell,\ell'') \in flow(S_\star) \wedge clear(x,\ell'',\ell')\}, \\ \qquad\qquad \text{if } \ell \neq\ ? \\ \{\ell' \mid clear(x, init(S_\star), \ell')\}, \text{if } \ell =\ ? \end{cases}$$

It turns out that:

$$du(x,\ell) = \{\ell' \mid \ell \in ud(x,\ell')\}$$

$[\texttt{x:=0}]^1; [\texttt{x:=3}]^2; (\texttt{if } [\texttt{z=x}]^3 \texttt{ then } [\texttt{z:=0}]^4 \texttt{ else } [\texttt{z:=x}]^5);$
$[\texttt{y:=x}]^6; [\texttt{x:=y+z}]^7$

$ud(x,\ell)$	x	y	z
1	$\emptyset$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\emptyset$	$\emptyset$
3	$\{2\}$	$\emptyset$	$\{?\}$
4	$\emptyset$	$\emptyset$	$\emptyset$
5	$\{2\}$	$\emptyset$	$\emptyset$
6	$\{2\}$	$\emptyset$	$\emptyset$
7	$\emptyset$	$\{6\}$	$\{4,5\}$

$du(x,\ell)$	x	y	z
1	$\emptyset$	$\emptyset$	$\emptyset$
2	$\{3,5,6\}$	$\emptyset$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$
4	$\emptyset$	$\emptyset$	$\{7\}$
5	$\emptyset$	$\emptyset$	$\{7\}$
6	$\emptyset$	$\{7\}$	$\emptyset$
7	$\emptyset$	$\emptyset$	$\emptyset$
?	$\emptyset$	$\emptyset$	$\{3\}$

One application of *ud*- and *du*-chains is for *Dead Code Elimination*; for the program on the previous slide we may e.g. remove the block labelled 1 because there will be no use of the value assigned to `x` before it is reassigned in the next block. Another application is in *Code Motion*; in the example program the block labelled 6 can be moved to just in front of the conditional because it only uses variables assigned in earlier blocks and the conditional does not use the variable assigned in block 6.

Constructive definitions. In order to define *ud*-chains we can use RD_{entry}, which records the assignments reaching a block and define

$$\mathsf{UD} : \mathbf{Var}_\star \times \mathbf{Lab}_\star \to \mathcal{P}(\mathbf{Lab}_\star)$$

$$\mathsf{UD}(x,\ell) = \begin{cases} \{\ell' \mid (x,\ell') \in \mathsf{RD}_{entry}(\ell)\} & \text{if } x \in gen_{\mathsf{LV}}(B^\ell) \\ \emptyset & \text{otherwise} \end{cases}$$

Monotone Frameworks. Despite the differences between the analyses we have studied there are sufficient similarities to make it plausible that there might be an underlying framework. The advantages that accrue from identifying such a framework include the possibility of designing generic algorithms for solving the data flow equations.

Each of the four classical analyses considers equations for a label consistent program $S_\star$ and they take the form:

$$\begin{array}{lcl} \mathit{Analysis}_\circ(\ell) & = & \begin{cases} \iota, \text{if } \ell \in E \\ \bigsqcup\{\mathit{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F\}, \text{otherwise} \end{cases} \\ \mathit{Analysis}_\bullet(\ell) & = & f_\ell(\mathit{Analysis}_\circ(\ell)) \end{array}$$

- $\bigsqcup$ is $\bigcap$ or $\bigcup$ (and $\sqcup$ is $\cup$ or $\cap$),
- F is either $\mathit{flow}(S_\star)$ or $\mathit{flow}^R(S_\star)$,
- E is $\{\mathit{init}(S_\star)\}$ or $\mathit{final}(S_\star)$,
- ι specifies the initial or final analysis information, and
- f_ℓ is the transfer function associated with $B^\ell \in \mathit{blocks}(S_\star)$.

- The forward analyses have F to be $\mathit{flow}(S_\star)$ and then $\mathit{Analysis}_\circ$ concerns entry conditions and $\mathit{Analysis}_\bullet$ concerns exit conditions; also the equation system presupposes that $S_\star$ has isolated entries.
- The backward analyses have F to be $\mathit{flow}^R(S_\star)$ and then $\mathit{Analysis}_\circ$ concerns exit conditions and $\mathit{Analysis}_\bullet$ concerns entry conditions; also the equation system presupposes that $S_\star$ has isolated exits.

- When $\bigsqcup$ is $\bigcap$ we require the *greatest* sets that solve the equations and we are able to detect properties satisfied by *all* paths of execution reaching (or leaving) the entry (or exit) of a label; these analyses are often called *must analyses*.
- When $\bigsqcup$ is $\bigcup$ we require the *least* sets that solve the equations and we are able to detect properties satisfied by *at least one* execution path to (or from) the entry (or exit) of a label; these analyses are often called *may analyses*.

It is occasionally awkward to have to assume that forward analyses have isolated entries and that backward analyses have isolated exits. This motivates reformulating the above equations to be of the form:

$$\begin{aligned} \mathit{Analysis}_\circ(\ell) &= \bigsqcup\{\mathit{Analysis}_\bullet(\ell') \mid (\ell', \ell) \in F\} \sqcup \iota_E^\ell \\ &\qquad \text{where } \iota_E^\ell = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \\ \mathit{Analysis}_\bullet(\ell) &= f_\ell(\mathit{Analysis}_\circ(\ell)) \end{aligned}$$

where $\bot$ satisfies $l \sqcup \bot = l$ (hence $\bot$ is not really there).

The view that we take here is that a program is a *transition system*; the nodes represent blocks and each block has a *transfer function* associated with it that specifies how the block acts on the "input" state. (Note that for forward analyses, the input state is the entry state, and for backward analyses, it is the exit state.)

A *Monotone Framework* consists of:

- a complete lattice, L, that satisfies the Ascending Chain Condition, and we write $\bigsqcup$ for the least upper bound operator; and
- a set $\mathcal{F}$ of monotone functions from L to L that contains the identity function and that is closed under function composition.

A *Distributive Framework* is a Monotone Framework where additionally all functions f in $\mathcal{F}$ are required to be distributive:

$$f(l_1 \sqcup l_2) = f(l_1) \sqcup f(l_2)$$

An *instance*, Analysis, of a Monotone (or Distributive) Framework to consists of:

- the complete lattice, L, of the framework;
- the space of functions, $\mathcal{F}$, of the framework;
- a finite *flow*, F, that typically is $\mathit{flow}(S_\star)$ or $\mathit{flow}^R(S_\star)$;
- a finite set of so-called *extremal labels*, E, that typically is $\{\mathit{init}(S_\star)\}$ or $\mathit{final}(S_\star)$;
- an *extremal value*, $\iota \in L$, for the extremal labels; and
- a mapping, $f_\cdot$, from the labels $\mathbf{Lab}_\star$ of F to transfer functions in $\mathcal{F}$.

An instance gives rise to a *set of equations*, $\mathsf{Analysis}^{=}$, of the form considered earlier:

$$\begin{aligned} Analysis_\circ(\ell) &= \bigsqcup\{Analysis_\bullet(\ell') \mid (\ell',\ell) \in F\} \sqcup \iota^\ell_E \\ &\qquad \text{where } \iota^\ell_E = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \\ Analysis_\bullet(\ell) &= f_\ell(Analysis_\circ(\ell)) \end{aligned}$$

It also gives rise to a *set of constraints*, $\mathsf{Analysis}^{\sqsubseteq}$, defined by:

$$\begin{aligned} Analysis_\circ(\ell) &\sqsupseteq \bigsqcup\{Analysis_\bullet(\ell') \mid (\ell',\ell) \in F\} \sqcup \iota^\ell_E \\ &\qquad \text{where } \iota^\ell_E = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases} \\ Analysis_\bullet(\ell) &= f_\ell(Analysis_\circ(\ell)) \end{aligned}$$

Available Expressions	Reaching Definitions	Very Busy Expressions	Live Variables
$\mathcal{P}(\mathbf{AExp}_\star)$ $\supseteq$ $\bigcap$ $\mathbf{AExp}_\star$	$\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star)$ $\subseteq$ $\bigcup$ $\emptyset$	$\mathcal{P}(\mathbf{AExp}_\star)$ $\supseteq$ $\bigcap$ $\mathbf{AExp}_\star$	$\mathcal{P}(\mathbf{Var}_\star)$ $\subseteq$ $\bigcup$ $\emptyset$
$\emptyset$ $\{init(S_\star)\}$ $flow(S_\star)$	$\{(x,?) \mid x \in FV(S_\star)\}$ $\{init(S_\star)\}$ $flow(S_\star)$	$\emptyset$ $final(S_\star)$ $flow^R(S_\star)$	$\emptyset$ $final(S_\star)$ $flow^R(S_\star)$
$\{f : L \to L \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g\}$ $f_\ell(l) = (l \setminus kill([B]^\ell)) \cup gen([B]^\ell)$ where $[B]^\ell \in blocks(S_\star)$			

Lemma: Each of the four classical data flow analyses is a Monotone Framework as well as a Distributive Framework.

It is worth pointing out that in order to get this result we have made the frameworks dependent upon the actual program – this is needed to enforce that the Ascending Chain Condition is fulfilled.

A Non-distributive Example. The *Constant Propagation Analysis* will determine:

> For each program point, whether or not a variable has a constant value whenever execution reaches that point.

Such information can be used as the basis for an optimisation known as *Constant Folding*: all uses of the variable may be replaced by the constant value.

$$\widehat{\mathbf{State}}_{\mathsf{CP}} = ((\mathbf{Var}_\star \to \mathbf{Z}^\top)_\bot, \sqsubseteq, \sqcup, \sqcap, \bot, \lambda x.\top)$$

where $\mathbf{Var}_\star$ is the set of variables appearing in the program and $\mathbf{Z}^\top = \mathbf{Z} \cup \{\top\}$ is partially ordered as follows:

$$\forall z \in \mathbf{Z}^\top : z \sqsubseteq \top$$

$$\forall z_1, z_2 \in \mathbf{Z} : (z_1 \sqsubseteq z_2) \Leftrightarrow (z_1 = z_2)$$

To capture the case where no information is available we extend $\mathbf{Var}_\star \to \mathbf{Z}^\top$ with a least element $\bot$, written $(\mathbf{Var}_\star \to \mathbf{Z}^\top)_\bot$. The partial ordering $\sqsubseteq$ on $\widehat{\mathbf{State}}_{\mathsf{CP}} = (\mathbf{Var}_\star \to \mathbf{Z}^\top)_\bot$ is defined by

$$\forall \widehat{\sigma} \in (\mathbf{Var}_\star \to \mathbf{Z}^\top)_\bot : \quad \bot \sqsubseteq \widehat{\sigma}$$

$$\forall \widehat{\sigma}_1, \widehat{\sigma}_2 \in \mathbf{Var}_\star \to \mathbf{Z}^\top : \quad \widehat{\sigma}_1 \sqsubseteq \widehat{\sigma}_2 \text{ iff } \forall x : \widehat{\sigma}_1(x) \sqsubseteq \widehat{\sigma}_2(x)$$

and the binary least upper bound operation is then:

$$\forall \widehat{\sigma} \in (\mathbf{Var}_\star \to \mathbf{Z}^\top)_\bot : \quad \widehat{\sigma} \sqcup \bot = \widehat{\sigma} = \bot \sqcup \widehat{\sigma}$$

$$\forall \widehat{\sigma}_1, \widehat{\sigma}_2 \in \mathbf{Var}_\star \to \mathbf{Z}^\top : \quad \forall x : (\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)(x) = \widehat{\sigma}_1(x) \sqcup \widehat{\sigma}_2(x)$$

$$\mathcal{A}_{\mathsf{CP}} \; : \; \mathbf{AExp} \to (\widehat{\mathbf{State}}_{\mathsf{CP}} \to \mathbf{Z}^\top_\bot)$$

$$\mathcal{A}_{\mathsf{CP}}[\![x]\!]\widehat{\sigma} \;=\; \begin{cases} \bot & \text{if } \widehat{\sigma} = \bot \\ \widehat{\sigma}(x) & \text{otherwise} \end{cases}$$

$$\mathcal{A}_{\mathsf{CP}}[\![n]\!]\widehat{\sigma} \;=\; \begin{cases} \bot & \text{if } \widehat{\sigma} = \bot \\ n & \text{otherwise} \end{cases}$$

$$\mathcal{A}_{\mathsf{CP}}[\![a_1 \; op_a \; a_2]\!]\widehat{\sigma} \;=\; \mathcal{A}_{\mathsf{CP}}[\![a_1]\!]\widehat{\sigma} \; \widehat{\mathsf{op}}_a \; \mathcal{A}_{\mathsf{CP}}[\![a_2]\!]\widehat{\sigma}$$

The operations on $\mathbf{Z}$ are lifted to $\mathbf{Z}^\top_\bot = \mathbf{Z} \cup \{\bot, \top\}$ by taking $z_1 \; \widehat{\mathsf{op}}_a \; z_2 = z_1 \; \mathbf{op}_a \; z_2$ if $z_1, z_2 \in \mathbf{Z}$ (and where $\mathbf{op}_a$ is the corresponding arithmetic operation on $\mathbf{Z}$), $z_1 \; \widehat{\mathsf{op}}_a \; z_2 = \bot$ if $z_1 = \bot$ or $z_2 = \bot$ and $z_1 \; \widehat{\mathsf{op}}_a \; z_2 = \top$ otherwise.

$$\mathcal{F}_{\mathsf{CP}} = \{f \mid f \text{ is a monotone function on } \widehat{\mathbf{State}}_{\mathsf{CP}}\}$$

$$[x := a]^\ell: \quad f_\ell^{\mathsf{CP}}(\widehat{\sigma}) = \begin{cases} \bot & \text{if } \widehat{\sigma} = \bot \\ \widehat{\sigma}[x \mapsto \mathcal{A}_{\mathsf{CP}}[\![a]\!]\widehat{\sigma}] & \text{otherwise} \end{cases}$$

$$[\mathtt{skip}]^\ell: \quad f_\ell^{\mathsf{CP}}(\widehat{\sigma}) = \widehat{\sigma}$$

$$[b]^\ell: \quad f_\ell^{\mathsf{CP}}(\widehat{\sigma}) = \widehat{\sigma}$$

Constant Propagation is a forward analysis, so for the program $S_\star$ we take the flow, F, to be $\mathit{flow}(S_\star)$, the extremal labels, E, to be $\{\mathit{init}(S_\star)\}$, the extremal value, ι_{CP}, to be $\lambda x.\top$.

Lemma Constant Propagation is a Monotone Framework that is *not* a Distributive Framework.

To show that it is not a Distributive Framework consider the transfer function f_ℓ^{CP} for $[\mathtt{y:=x*x}]^\ell$ and let $\widehat{\sigma}_1$ and $\widehat{\sigma}_2$ be such that $\widehat{\sigma}_1(\mathtt{x}) = 1$ and $\widehat{\sigma}_2(\mathtt{x}) = -1$. Then $\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2$ maps $\mathtt{x}$ to $\top$ and thus $f_\ell^{\mathsf{CP}}(\widehat{\sigma}_1 \sqcup \widehat{\sigma}_2)$ maps $\mathtt{y}$ to $\top$ and hence fails to record that $\mathtt{y}$ has the constant value 1. However, both $f_\ell^{\mathsf{CP}}(\widehat{\sigma}_1)$ and $f_\ell^{\mathsf{CP}}(\widehat{\sigma}_2)$ map $\mathtt{y}$ to 1 and so does $f_\ell^{\mathsf{CP}}(\widehat{\sigma}_1) \sqcup f_\ell^{\mathsf{CP}}(\widehat{\sigma}_2)$.

The MFP solution.

INPUT: An instance of a Monotone Framework: $(L, \mathcal{F}, F, E, \iota, f.)$

OUTPUT: $MFP_\circ, MFP_\bullet$

METHOD: Step 1: Initialisation (of W and Analysis)

```
W := nil;
for all (ℓ, ℓ') in F do
    W := cons((ℓ, ℓ'),W);
for all ℓ in F or E do
    if ℓ ∈ E then Analysis[ℓ] := ι
             else Analysis[ℓ] := ⊥_L;
```


Step 2: Iteration (updating W and Analysis)

```
while W ≠ nil do
    ℓ := fst(head(W)); ℓ' = snd(head(W));
    W := tail(W);
    if f_ℓ(Analysis[ℓ]) ⋢ Analysis[ℓ'] then
       Analysis[ℓ'] := Analysis[ℓ'] ⊔ f_ℓ(Analysis[ℓ]);
       for all (ℓ', ℓ'') in F do W := cons((ℓ', ℓ''),W);
```

Step 3: Presenting the result ($MFP_\circ$ and $MFP_\bullet$)

```
for all ℓ in F or E do
    MFP_∘(ℓ) := Analysis[ℓ];
    MFP_•(ℓ) := f_ℓ(Analysis[ℓ])
```

Lemma The worklist algorithm always terminates and it computes the least (or MFP) solution to the instance of the framework given as input.

Complexity. Assume that the flow F is represented in such a way that all (ℓ', ℓ'') emanating from ℓ' can be found in time proportional to their number. Suppose that E and F contain at most $b \geq 1$ distinct labels, that F contains at most $e \geq b$ pairs, and that L has finite height at most $h \geq 1$. Then steps 1 and 3 perform at most $O(b + e)$ basic operations.
Concerning step 2 a pair is placed on the worklist at most $O(h)$ times, and each time it takes only a constant number of basic steps to process it; this yields at most $O(e \cdot h)$ basic operations for step 2. Since $h \geq 1$ and $e \geq b$ this gives at most $O(e \cdot h)$ basic operations for the algorithm.

Consider the Reaching Definitions Analysis and suppose that there are at most $v \geq 1$ variables and $b \geq 1$ labels in the program, $S_\star$, being analysed. Since $L = \mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star)$, it follows that $h \leq v \cdot b$ and thus we have an $O(v \cdot b^3)$ upper bound on the number of basic operations.

Actually we can do better. If $S_\star$ is label consistent then the variable of the pairs (x, ℓ) of $\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star)$ will always be uniquely determined by the label ℓ so we get an $O(b^3)$ upper bound on the number of basic operations. Furthermore, F is $\mathit{flow}(S_\star)$ and inspection of the equations for $\mathit{flow}(S_\star)$ shows that for each label ℓ we construct at most two pairs with ℓ in the first component. This means that $e \leq 2 \cdot b$ and we get an $O(b^2)$ upper bound on the number of basic operations.

The MOP Solution. Consider an instance $(L, \mathcal{F}, F, E, \iota, f.)$ of a Monotone Framework. We shall use the notation $\vec{\ell} = [\ell_1, \cdots, \ell_n]$ for a sequence of $n \geq 0$ labels. The paths up to *but not* including ℓ are

$$path_\circ(\ell) = \{[\ell_1, \cdots, \ell_{n-1}] \mid n \geq 1 \wedge \forall i < n : (\ell_i, \ell_{i+1}) \in F \wedge \ell_n = \ell \wedge \ell_1 \in E\}$$

and the paths up to *and* including ℓ are:

$$path_\bullet(\ell) = \{[\ell_1, \cdots, \ell_n] \mid n \geq 1 \wedge \forall i < n : (\ell_i, \ell_{i+1}) \in F \wedge \ell_n = \ell \wedge \ell_1 \in E\}$$

For a path $\vec{\ell} = [\ell_1, \cdots, \ell_n]$ we define the transfer function

$$f_{\vec{\ell}} = f_{\ell_n} \circ \cdots \circ f_{\ell_1} \circ id$$

so that for the empty path we have $f_{[\,]} = id$ where id is the identity function.

$$MOP_\circ(\ell) = \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\circ(\ell)\}$$

$$MOP_\bullet(\ell) = \bigsqcup\{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in path_\bullet(\ell)\}$$

Unfortunately, the MOP solution is sometimes uncomputable (meaning that it is undecidable) even though the MFP solution is always easily computable (because of the property space satisfying the Ascending Chain Condition); the following result establishes one such result:

Lemma The MOP solution for Constant Propagation is undecidable.

Lemma Consider the MFP and MOP solutions to an instance $(L, \mathcal{F}, F, B, \iota, f.)$ of a Monotone Framework; then:

$$MFP_\circ \sqsupseteq MOP_\circ \text{ and } MFP_\bullet \sqsupseteq MOP_\bullet$$

If the framework is distributive and if $path_\circ(\ell) \neq \emptyset$ for all ℓ in E and F then:

$$MFP_\circ = MOP_\circ \text{ and } MFP_\bullet = MOP_\bullet$$

It is always possible to formulate the MOP solution as an MFP solution over a different property space (like $\mathcal{P}(L)$) and therefore little is lost by focusing on the fixed point approach to Monotone Frameworks.

Interprocedural Analysis.

A program, $P_\star$, in the extended WHILE-language has the form

$$\texttt{begin } D_\star \; S_\star \texttt{ end}$$

where $D_\star$ is a sequence of procedure declarations:

$$D ::= \texttt{proc } p(\texttt{val } x, \texttt{res } y) \texttt{ is}^{\ell_n} \; S \texttt{ end}^{\ell_x} \mid D \; D$$

$$S ::= \cdots \mid [\texttt{call } p(a, z)]^{\ell_c}_{\ell_r}$$

```
begin  proc fib(val z, u, res v) is[1]
           if [z<3][2] then [v:=u+1][3]
           else ([call fib(z-1,u,v)][4][5]; [call fib(z-2,v,v)][6][7])
       end[8];
       [call fib(x,0,y)][9][10]
end
```


$$
\begin{aligned}
\mathit{init}([\texttt{call } p(a,z)]^{\ell_c}_{\ell_r}) &= \ell_c \\
\mathit{final}([\texttt{call } p(a,z)]^{\ell_c}_{\ell_r}) &= \{\ell_r\} \\
\mathit{blocks}([\texttt{call } p(a,z)]^{\ell_c}_{\ell_r}) &= \{[\texttt{call } p(a,z)]^{\ell_c}_{\ell_r}\} \\
\mathit{labels}([\texttt{call } p(a,z)]^{\ell_c}_{\ell_r}) &= \{\ell_c, \ell_r\} \\
\mathit{flow}([\texttt{call } p(a,z)]^{\ell_c}_{\ell_r}) &= \{(\ell_c;\ell_n),(\ell_x;\ell_r)\} \\
& \quad \text{if} \\
& \quad \texttt{proc } p(\texttt{val } x, \texttt{res } y) \texttt{ is}^{\ell_n} \ S \ \texttt{end}^{\ell_x} \\
& \quad \text{is in } D_\star
\end{aligned}
$$

$(\ell_c; \ell_n)$ and $(\ell_x; \ell_r)$ are new kinds of flows:

- $(\ell_c; \ell_n)$ is the flow corresponding to *calling* a procedure at ℓ_c and with ℓ_n being the entry point for the procedure body, and
- $(\ell_x; \ell_r)$ is the flow corresponding to exiting a procedure body at ℓ_x and *returning* to the call at ℓ_r.

Next consider the program $P_\star$ of the form `begin` $D_\star$ $S_\star$ `end`. For each procedure declaration `proc` p(`val` x, `res` y) `is`$^{\ell_n}$ S `end`$^{\ell_x}$ we set

$$\begin{aligned}
\mathit{init}(p) &= \ell_n \\
\mathit{final}(p) &= \{\ell_x\} \\
\mathit{blocks}(p) &= \{\texttt{is}^{\ell_n}, \texttt{end}^{\ell_x}\} \cup \mathit{blocks}(S) \\
\mathit{labels}(p) &= \{\ell_n, \ell_x\} \cup \mathit{labels}(S) \\
\mathit{flow}(p) &= \{(\ell_n, \mathit{init}(S))\} \cup \mathit{flow}(S) \cup \{(\ell, \ell_x) \mid \ell \in \mathit{final}(S)\}
\end{aligned}$$

For the entire program $P_\star$ we set

$$\begin{array}{rcl}
\mathit{init}_\star & = & \mathit{init}(S_\star) \\
\mathit{final}_\star & = & \mathit{final}(S_\star) \\
\mathit{blocks}_\star & = & \bigcup\{\mathit{blocks}(p) \mid \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x} \\
 & & \text{is in } D_\star\} \cup\ \mathit{blocks}(S_\star) \\
\mathit{labels}_\star & = & \bigcup\{\mathit{labels}(p) \mid \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x} \\
 & & \text{is in } D_\star\} \cup\ \mathit{labels}(S_\star) \\
\mathit{flow}_\star & = & \bigcup\{\mathit{flow}(p) \mid \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x} \\
 & & \text{is in } D_\star\} \cup\ \mathit{flow}(S_\star)
\end{array}$$

as well as $\mathbf{Lab}_\star = \mathit{labels}_\star$.

We shall also need to define a notion of *interprocedural flow*

$$\begin{array}{rcl}
\mathit{inter\text{-}flow}_\star & = & \{(\ell_c, \ell_n, \ell_x, \ell_r) \mid P_\star \text{ contains } [\texttt{call}\ p(a, z)]^{\ell_c}_{\ell_r} \\
 & & \text{as well as } \texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}\}
\end{array}$$

that clearly indicates the relationship between the labels of a procedure call and the corresponding procedure body.

For the Fibonacci program we have

$$\begin{aligned} \mathit{flow}_\star &= \{(1,2),(2,3),(3,8),\\ &\quad (2,4),(4;1),(8;5),(5,6),(6;1),(8;7),(7,8),\\ &\quad (9;1),(8;10)\} \end{aligned}$$

$$\mathit{inter\text{-}flow}_\star = \{(9,1,8,10),(4,1,8,5),(6,1,8,7)\}$$

and $\mathit{init}_\star = 9$ and $\mathit{final}_\star = \{10\}$.

For a *forward analysis* we use $F = \mathit{flow}_\star$, $E = \{\mathit{init}_\star\}$ and $IF = \mathit{inter\text{-}flow}_\star$ whereas for a *backwards analysis* we use $F = \mathit{flow}_\star^R$, $E = \mathit{final}_\star$ and $IF = \mathit{inter\text{-}flow}_\star$.

Intraprocedural versus Interprocedural Analysis.

- for each procedure call $[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ we have two transfer functions f_{ℓ_c} and f_{ℓ_r} corresponding to calling the procedure and returning from the call, and
- for each procedure definition $\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$ we have two transfer functions f_{ℓ_n} and f_{ℓ_x} corresponding to entering and exiting the procedure body.

$$
\begin{aligned}
A_\bullet(\ell) &= f_\ell(A_\circ(\ell)) \\
A_\circ(\ell) &= \bigsqcup\{A_\bullet(\ell') \mid (\ell',\ell) \in F \text{ or } (\ell';\ell) \in F\} \sqcup \iota^\ell_E
\end{aligned}
$$

$$
\iota^\ell_E = \begin{cases} \iota & \text{if } \ell \in E \\ \bot & \text{if } \ell \notin E \end{cases}
$$

The MVP solution.

Complete paths.

$CP_{\ell_1,\ell_2} \longrightarrow \ell_1$ whenever $\ell_1 = \ell_2$

$CP_{\ell_1,\ell_3} \longrightarrow \ell_1, CP_{\ell_2,\ell_3}$ whenever $(\ell_1,\ell_2) \in F$;
for a forward analysis this means that $(\ell_1,\ell_2) \in \mathit{flow}_\star$

$CP_{\ell_c,\ell} \longrightarrow \ell_c, CP_{\ell_n,\ell_x}, CP_{\ell_r,\ell}$ whenever $(\ell_c,\ell_n,\ell_x,\ell_r) \in IF$;
for a forward analysis this means that $P_\star$ contains
$[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ and $\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$

Valid paths.

$VP_\star \longrightarrow VP_{\ell_1,\ell_2}$ whenever $\ell_1 \in E$ and $\ell_2 \in \mathbf{Lab}_\star$

$VP_{\ell_1,\ell_2} \longrightarrow \ell_1$ whenever $\ell_1 = \ell_2$

$VP_{\ell_1,\ell_3} \longrightarrow \ell_1, VP_{\ell_2,\ell_3}$ whenever $(\ell_1,\ell_2) \in F$

$VP_{\ell_c,\ell} \longrightarrow \ell_c, CP_{\ell_n,\ell_x}, VP_{\ell_r,\ell}$ whenever $(\ell_c,\ell_n,\ell_x,\ell_r) \in IF$

$VP_{\ell_c,\ell} \longrightarrow \ell_c, VP_{\ell_n,\ell}$ whenever $(\ell_c,\ell_n,\ell_x,\ell_r) \in IF$

$$
\begin{aligned}
vpath_\circ(\ell) &= \{[\ell_1, \cdots, \ell_{n-1}] \mid n \geq 1 \wedge \ell_n = \ell \\
&\quad \wedge [\ell_1, \cdots, \ell_n] \text{ is a valid path}\} \\
vpath_\bullet(\ell) &= \{[\ell_1, \cdots, \ell_n] \mid n \geq 1 \wedge \ell_n = \ell \\
&\quad \wedge [\ell_1, \cdots, \ell_n] \text{ is a valid path}\}
\end{aligned}
$$

$$
MVP_\circ(\ell) = \bigsqcup \{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in vpath_\circ(\ell)\}
$$

$$
MVP_\bullet(\ell) = \bigsqcup \{f_{\vec{\ell}}(\iota) \mid \vec{\ell} \in vpath_\bullet(\ell)\}
$$

Making Context Explicit.

$\delta \in \Delta$ context information

Consider an instance $(L, \mathcal{F}, F, E, \iota, f_\cdot)$ of a Monotone Framework. We shall now construct an instance

$$
(\widehat{L}, \widehat{\mathcal{F}}, F, E, \widehat{\iota}, \widehat{f}_\cdot)
$$

of an *embellished monotone framework* that takes context into account.

- $\widehat{L} = \Delta \to L$;
- the transfer functions in $\widehat{\mathcal{F}}$ are monotone; and
- each transfer function $\widehat{f}_\ell$ is given by $\widehat{f}_\ell(\widehat{l})(\delta) = f_\ell(\widehat{l}\,(\delta))$.

$$
\begin{array}{lcl}
A_\bullet(\ell) & = & \widehat{f}_\ell(A_\circ(\ell)) \\
 & & \text{for all labels that do not label a procedure call} \\
 & & \text{(i.e. that do not occur as first or fourth components} \\
 & & \text{of a tuple in } \mathit{IF}) \\
A_\circ(\ell) & = & \bigsqcup\{A_\bullet(\ell') \mid (\ell',\ell) \in F \text{ or } (\ell';\ell) \in F\} \sqcup \widehat{\iota^\ell_E} \\
 & & \text{for all labels (including procedure calls)}
\end{array}
$$

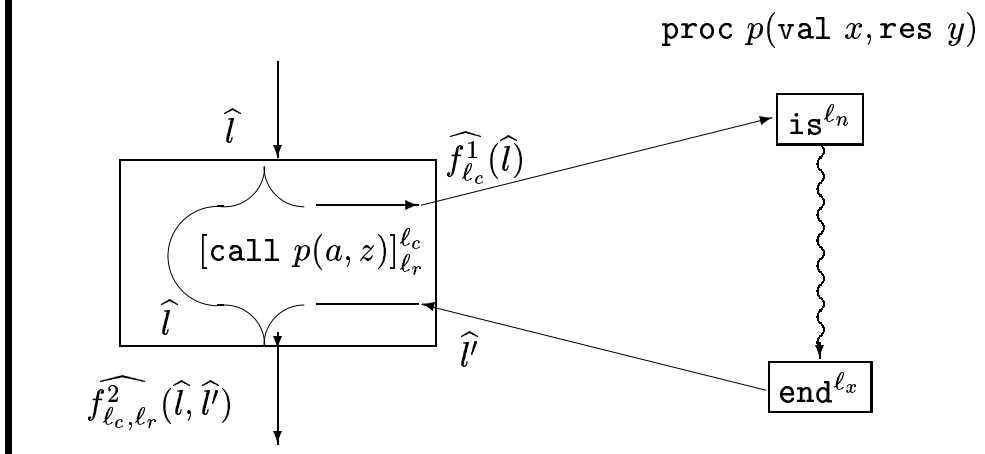
For a procedure definition $\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$ we have two transfer functions:

$$\widehat{f_{\ell_n}}, \widehat{f_{\ell_x}} : (\Delta \to L) \to (\Delta \to L)$$

In the case of our simple language we shall prefer to take both of these transfer functions to be the identity function; i.e.

$$
\begin{array}{lcl}
\widehat{f_{\ell_n}}(\widehat{l}) & = & \widehat{l} \\
\widehat{f_{\ell_x}}(\widehat{l}) & = & \widehat{l}
\end{array}
$$

for all $\widehat{l} \in \widehat{L}$.



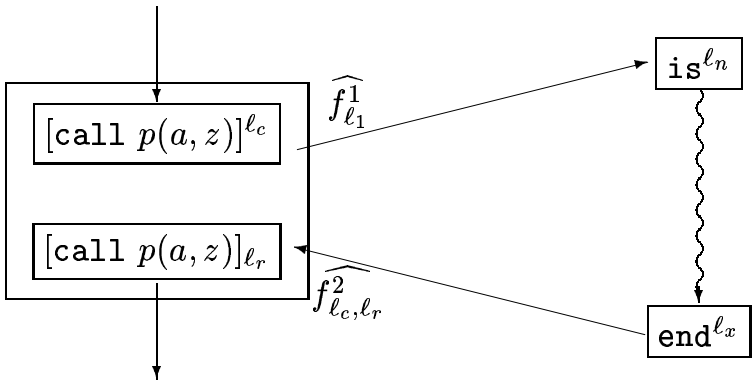
$$\widehat{f^1_{\ell_c}} : (\Delta \to L) \to (\Delta \to L)$$

$$A_\bullet(\ell_c) \quad = \quad \widehat{f^1_{\ell_c}}(A_\circ(\ell_c)) \quad \text{for all } (\ell_c, \ell_n, \ell_x, \ell_r) \in IF$$

$$\widehat{f^2_{\ell_c,\ell_r}} : (\Delta \to L) \times (\Delta \to L) \to (\Delta \to L)$$

$$A_\bullet(\ell_r) \quad = \quad \widehat{f^2_{\ell_c,\ell_r}}(A_\circ(\ell_c), A_\circ(\ell_r)) \quad \text{for all } (\ell_c, \ell_n, \ell_x, \ell_r) \in IF$$

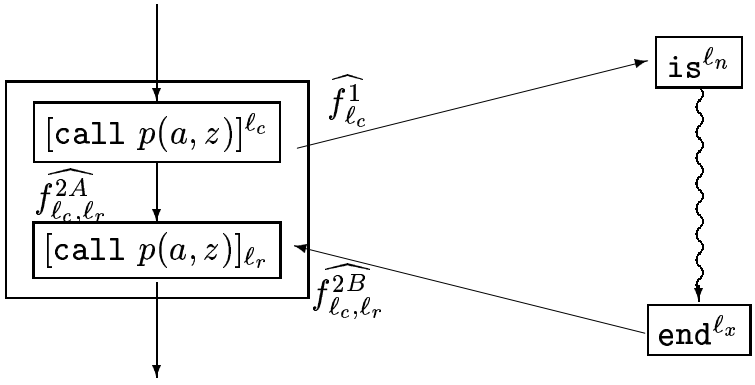
proc $p(\texttt{val}\ x, \texttt{res}\ y)$



$$\widehat{f^2_{\ell_c,\ell_r}}(\hat{l},\hat{l}') = \widehat{f^2_{\ell_r}}(\hat{l}')$$

$$\widehat{f^2_{\ell_c,\ell_r}}(\hat{l},\hat{l}') = \widehat{f^{2A}_{\ell_c,\ell_r}}(\hat{l}) \sqcup \widehat{f^{2B}_{\ell_c,\ell_r}}(\hat{l}')$$

proc $p(\texttt{val}\ x, \texttt{res}\ y)$



Call strings (unbounded) as context.

As the first possibility we simply encode the path taken; we shall only record flows of the form $(\ell_c; \ell_n)$ corresponding to a procedure call.

$$\Delta = \mathbf{Lab}^*$$

where the most recent label ℓ_c of a procedure call is at the right end (just as was the case for valid paths and paths); elements of Δ are called *call strings*. We then define

$$\widehat{\iota} = (\Lambda, \iota)$$

where Λ is the empty sequence corresponding to the fact that there are no pending procedure calls when the program starts execution.

For a procedure call $(\ell_c, \ell_n, \ell_x, \ell_r) \in IF$, amounting to $[\texttt{call}\ p(a,z)]^{\ell_c}_{\ell_r}$ in the case of a forward analysis, we define the transfer function $\widehat{f^1_{\ell_c}}$ such that $\widehat{f^1_{\ell_c}}(\widehat{l})([\delta, \ell_c]) = f^1_{\ell_c}(\widehat{l}(\delta))$ where $[\delta, \ell_c]$ denotes the path obtained by appending ℓ_c to δ and the function $f^1_{\ell_c} : L \to L$ describes how the property is modified. This is achieved by setting

$$\widehat{f^1_{\ell_c}}(\widehat{l})(\delta') = \begin{cases} f^1_{\ell_c}(\widehat{l}(\delta)) & \text{when } \delta' = [\delta, \ell_c] \\ \bot & \text{otherwise} \end{cases}$$

$$\widehat{f^2_{\ell_c,\ell_r}}(\widehat{l},\widehat{l'})(\delta) = f^2_{\ell_c,\ell_r}(\widehat{l}(\delta),\widehat{l'}([\delta,\ell_c]))$$

Here the information $\widehat{l}$ from the original call is combined with information $\widehat{l'}$ from the procedure exit using the function $f^2_{\ell_c,\ell_r} : L \times L \to L$. However, only information corresponding to the *same* contexts for call point ℓ_c is combined: this is ensured by the two occurrences of δ in the above formula.

Call strings (bounded) as context.

$$\Delta = \mathbf{Lab}^{\leq k}$$

and we still take the extremal value to be $\widehat{\iota} = (\Lambda, \iota)$. Note that in the case $k = 0$ we have $\Delta = \{\Lambda\}$ which is equivalent to having no context information.

The transfer function $\widehat{f^1_{\ell_c}}$ for procedure call is redefined by

$$\widehat{f^1_{\ell_c}}(\widehat{l})(\delta') = \bigsqcup\{f^1_{\ell_c}(\widehat{l}(\delta)) \mid \delta' = \lceil \delta, \ell_c \rceil_k\}$$

Similarly, the transfer function $\widehat{f^2_{\ell_c,\ell_r}}$ for procedure return is redefined by

$$\widehat{f^2_{\ell_c,\ell_r}}(\widehat{l},\widehat{l'}) = f^2_{\ell_c,\ell_r}(\widehat{l}(\delta), \widehat{l'}(\lceil \delta, \ell_c \rceil_k))$$

Flow-Sensitivity versus Flow-Insensitivity.

All of the data flow analyses we have considered so far have been *flow-sensitive*: this just means that in general we would expect the analysis of a program $S_1; S_2$ to differ from the analysis of the program $S_2; S_1$ where the statements come in a different order.

Sometimes one considers *flow-insensitive* analyses where the order of statements is of no importance for the analysis being performed. Clearly a flow-insensitive analysis may be much less precise than its flow-sensitive analogue but also it is likely to be much cheaper; since interprocedural data flow analyses tend to be very costly, it is therefore useful to have a repertoire of techniques for reducing the cost.

The set $IAV(S)$ of directly *assigned variables* gives for each statement S the set of variables that could be assigned in S – but ignoring the effect of procedure calls.

$$\begin{aligned}
IAV([\texttt{skip}]^\ell) &= \emptyset \\
IAV([x := a]^\ell) &= \{x\} \\
IAV(S_1; S_2) &= IAV(S_1) \cup IAV(S_2) \\
IAV(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2) &= IAV(S_1) \cup IAV(S_2) \\
IAV(\texttt{while } [b]^\ell \texttt{ do } S) &= IAV(S) \\
IAV([\texttt{call } p(a,z)]^{\ell_c}_{\ell_r}) &= \{z\}
\end{aligned}$$

The set $ICP(S)$ of immediately *called procedures* that gives for each statement S the set of procedure names that could be directly called in S – but ignoring the effect of procedure calls.

$$\begin{aligned}
ICP([\texttt{skip}]^\ell) &= \emptyset \\
ICP([x := a]^\ell) &= \emptyset \\
ICP(S_1; S_2) &= ICP(S_1) \cup ICP(S_2) \\
ICP(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2) &= ICP(S_1) \cup ICP(S_2) \\
ICP(\texttt{while } [b]^\ell \texttt{ do } S) &= ICP(S) \\
ICP([\texttt{call } p(a,z)]^{\ell_c}_{\ell_r}) &= \{p\}
\end{aligned}$$

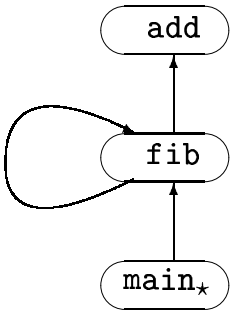
$$AV(p) = (IAV(S) \setminus \{x\}) \cup \bigcup \{AV(p') \mid p' \in ICP(S)\}$$

where $\texttt{proc}\ p(\texttt{val}\ x, \texttt{res}\ y)\ \texttt{is}^{\ell_n}\ S\ \texttt{end}^{\ell_x}$ is in $D_\star$


```
begin proc fib(val z) is
           if z<3 then call add(1)
           else (call fib(z-1); call fib(z-2))
      end;
      proc add(val u) is (y:=y+u; u:=0)
      end;
      y:=0; call fib(x)
end
```

$$AV(\texttt{fib}) = (\emptyset \setminus \{\texttt{z}\}) \cup AV(\texttt{fib}) \cup AV(\texttt{add})$$

$$AV(\texttt{add}) = \{\texttt{y}, \texttt{u}\} \setminus \{\texttt{u}\}$$



The least solution to the equation system is

$$AV(\texttt{fib}) = AV(\texttt{add}) = \{\texttt{y}\}$$

showing that only the variable y will be assigned by the procedure calls.