Exercise 1.6 Consider the Chaotic Iteration algorithm of Section 1.7 and suppose that
\[ \emptyset \subseteq R^D \subseteq F(R^D) \subseteq F^n(\emptyset) = F^{n+1}(\emptyset) \]
holds immediately before the assignment to \( R^D \); show that this also holds afterwards. (Hint: Write \( R^D_i \) for \( (R^D_1, \ldots, F_i(R^D_1, \ldots, R^D_{12}) \) and use the monotonicity of \( F \) and \( R^D \subseteq F(R^D) \) to establish that \( R^D \subseteq R^D_i \subseteq F(R^D) \subseteq F(R^D_{i+1}) \).)

Exercise 1.7 Use the Chaotic Iteration scheme of Section 1.7 to show that the information displayed in Table 1.1 is in fact the least fixed point of the function \( F \) defined in Section 1.3.

Exercise 1.8 Consider the following program
\[
[z := 1]^2; \text{while } [x > 0]^2 \text{ do } ([z := z \cdot y]^2; [x := x - 1]^4)
\]
computing the \( x \)-th power of the number stored in \( y \). Formulate a system of data flow equations in the manner of Section 1.3. Next use the Chaotic Iteration strategy of Section 1.7 to compute the least solution and present it in a table (like Table 1.1).

Exercise 1.9 Perform Constant Folding upon the program
\[
[x := 10]^2; [y := x + 10]^2; [z := y + x]^3
\]
so as to obtain
\[
[x := 10]^2; [y := 20]^2; [z := 30]^3
\]
How many ways of obtaining the result are there?

Exercise 1.10 The specification of Constant Folding in Section 1.8 only considers arithmetic expressions. Extend it to deal also with boolean expressions. Consider adding axioms like
\[
RD \vdash (\text{skip} ; S) \Rightarrow S
\]
\[
RD \vdash (\text{if } [\text{true} ] \text{ then } S_1 \text{ else } S_2 ) \Rightarrow S_1
\]
and discuss what complications arise.

Exercise 1.11 Consider adding the axiom
\[
RD \vdash [z := a]^c \Rightarrow [z := a[y \mapsto a]]^c
\]
if \( y \in FV(a) \land (y,?) \notin RD_{\text{entry}}(f) \land \forall (z,c') \in RD_{\text{entry}}(f) : (y = z \Rightarrow f \cdot c') = [y := a]^c \)
to the specification of Constant Folding given in Section 1.8 and discuss whether or not this is a good idea.

Chapter 2

Data Flow Analysis

In this chapter we introduce techniques for Data Flow Analysis. Data Flow Analysis is the traditional form of program analysis which is described in many textbooks on compiler writing. We will present analyses for the simple imperative language \textsc{while} that was introduced in Chapter 1. This includes a number of classical Data Flow Analyses: Available Expressions, Reaching Definitions, Very Busy Expressions and Live Variables. We introduce an operational semantics for \textsc{while} and demonstrate the correctness of the live variables analysis. We then present the notion of Monotone Frameworks and show how the examples may be recast as such frameworks. We continue by presenting a worklist algorithm for solving flow equations and study its termination and correctness properties. The chapter concludes with a presentation of some advanced topics, including Interprocedural Data Flow Analysis and Shape Analysis.

Throughout the chapter we will clarify the distinctions between intraprocedural and interprocedural analyses, between forward and backward analyses, between may and must analyses (or union and intersection analyses), between flow sensitive and flow insensitive analyses, and between context sensitive and context insensitive analyses.

2.1 Intraprocedural Analysis

In this section we present a number of example Data Flow Analyses for the \textsc{while} language. The analyses are each defined by pairs of functions that map labels to the appropriate sets; one function in each pair specifies information that is true on entry to the block, the second specifies information that is true at the exit.
2.1 Intraprocedural Analysis

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

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

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

**Flows and reverse flows.** We will need to operate on edges, or *flows*, between labels in a statement. We define a function

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

which maps statements to sets of flows:

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

Thus \( \text{labels}(S) \) and \( \text{flow}(S) \) will be a representation of the flow graph of \( S \).

**Example 2.1** Consider the following program, power, computing the \( x \)-th power of the number stored in \( y \):

\[ [x := 1]; \text{while } [x > 0] \rightarrow [x := x^y]; [x := x - 1] \]

We have \( \text{init}(\text{power}) = 1 \), \( \text{final}(\text{power}) = 2 \) and \( \text{labels}(\text{power}) = \{1, 2, 3, 4\} \).

The function \( \text{flow} \) produces the following set

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

which corresponds to the flow graph in Figure 2.1.

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

\[ \text{labels}(S) = \{\text{init}(S)\} \cup \{ \ell | (\ell, \ell') \in \text{flow}(S) \} \cup \{ \ell' | (\ell, \ell') \in \text{flow}(S) \} \]

and for composite statements (meaning those not simply of the form \([B]^\ell\)) the equation remains true when removing the \{\text{init}(S)\} component.
In order to formulate backward analyses we require a function that computes reverse flows:
\[ \text{flow}^{R} : \text{Stmt} \rightarrow \mathcal{P}(\text{Lab} \times \text{Lab}) \]

It is defined by:
\[ \text{flow}^{R}(S) = \{ (\ell, \ell') | (\ell, \ell') \in \text{flow}(S) \} \]

Example 2.2 For the power program of Example 2.1, flow\(^{R}\) produces
\[ \{ (2, 1), (2, 4), (3, 2), (4, 3) \} \]
which corresponds to a modified version of the flow graph in Figure 2.1 where the direction of the arcs has been reversed.

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
\[ \text{labels}(S) = \text{final}(S) \cup \{ \ell | (\ell, \ell') \in \text{flow}^{R}(S) \} \cup \{ \ell' | (\ell, \ell') \in \text{flow}^{R}(S) \} \]
and for composite statements the equation remains true when removing the final\((S)\) component.

The program of interest. We will use the notation \(S_{*}\) to represent the program that we are analysing (the “top-level” statement), Lab\(_{*}\) to represent the labels (labels\((S_{*})\)) appearing in \(S_{*}\), Var\(_{*}\) to represent the variables

(blocks\((S_{*})\)) occurring in \(S_{*}\), and AExp\(_{*}\) to represent the set of non-trivial arithmetic subexpressions in \(S_{*}\); an expression is trivial if it is a single variable or constant. We will also write AExp\((u)\) and AExp\((b)\) to refer to the set of non-trivial arithmetic subexpressions of a given arithmetic, respectively boolean, expression.

To simplify the presentation of the analyses, and to follow the traditions of the literature, we shall frequently assume that the program \(S_{*}\) has isolated entries; this means that:
\[ \forall \ell \in \text{Lab} : (\ell, \text{init}(S_{*})) \notin \text{flow}(S_{*}) \]
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:
\[ \forall \ell_{1} \in \text{final}(S_{*}) \forall \ell_{2} \in \text{Lab} : (\ell_{1}, \ell_{2}) \notin \text{flow}(S_{*}) \]

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 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 \(B\). We shall exploit this when defining the example analyses below.

Example 2.3 The power program of Example 2.1 has isolated entries but not isolated exits. It is clearly label consistent as well as uniquely labelled.

2.1.1 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.

Example 2.4 Consider the following program:
\[ [x := a + b]; [y := a + b]; \text{while } y > a + b \text{ do } ([a := a + 1]; [x := a + b]) \]
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
kill and gen functions
\[
\begin{align*}
\text{kill}_{AE}(x := a) &= \{ a' \in AExp, \ x \in FV(a') \} \\
\text{kill}_{AE}(\text{skip}) &= \emptyset \\
\text{kill}_{AE}(b) &= \emptyset \\
\text{gen}_{AE}(x := a) &= \{ a' \in AExp(a) \mid x \notin FV(a') \} \\
\text{gen}_{AE}(\text{skip}) &= \emptyset \\
\text{gen}_{AE}(b) &= AExp(b)
\end{align*}
\]

data flow equations: \( AE \)
\[
\begin{align*}
AE_{\text{entry}}(\ell) &= \emptyset \quad \text{if } \ell = \text{init}(S_a) \\
AE_{\text{entry}}(\ell) &= \{ \forall \ell', \ell \in \text{flow}(S_a) \} \cup \text{kill}_{AE}(B^t) \cup \text{gen}_{AE}(B^t) \\
\text{where } B^t \in \text{blocks}(S_a)
\end{align*}
\]

Table 2.1: Available Expressions Analysis.

The analysis is defined in Table 2.1 and explained below. An expression is killed in a block if any of the variables used in the expression are modified in the block; we use the function

\[
\text{kill}_{AE} : \text{Blocks} \rightarrow \mathcal{P}(AExp)
\]

to produce the set of non-trivial expressions killed in the block. Test and skip blocks do not kill any expressions and assignments kill any expression that uses the variable that appears in the left hand side of the assignment. Note that in the clause for \(x := a\) we have used the notation \(a' \in AExp\) to denote the fact that \(a'\) is a non-trivial arithmetic expression appearing in the program.

A generated expression is an expression that is evaluated in the block and where none of the variables used in the expression are later modified in the block. The set of non-trivial generated expressions is produced by the function:

\[
\text{gen}_{AE} : \text{Blocks} \rightarrow \mathcal{P}(AExp)
\]

The analysis itself is now defined by the functions \( AE_{\text{entry}} \) and \( AE_{\text{exit}} \) that each map labels to sets of expressions:

\[
AE_{\text{entry}}, AE_{\text{exit}} : \text{Lab} \rightarrow \mathcal{P}(AExp)
\]

For a label consistent program \( S_a \) (with isolated entries) the functions can be defined as in Table 2.1.

Figure 2.2: A schematic flow graph.

The analysis is a forward analysis and, as we shall see, we are interested in the largest sets satisfying the equation for \( AE_{\text{entry}} \) - an expression will be considered available if no path kills it. No expression is available at the start of the program. Subsequently, the expressions that are available at the entry to a block are any expressions that are available at all of the exits from blocks that flow to the block; if there are no such blocks the formula evaluates to \( AExp \). Given a set of expressions that are available at the entry, the expressions available at the exit of the block are computed by removing killed expressions and adding any new expression generated by the block.

To see why we require the largest solution, consider Figure 2.2 which shows the flow graph for a program in a schematic way. Such a flow graph might correspond to the following program:

\[
[z := x + y], \text{while true do [skip]}
\]

The set of expressions generated by the first assignment is \( \{x + y\} \); the other blocks do not generate expressions and no block kills any expressions. The equations for \( AE_{\text{entry}} \) and \( AE_{\text{exit}} \) are as follows:

\[
\begin{align*}
AE_{\text{entry}}(\ell) &= \emptyset \\
AE_{\text{entry}}(\ell') &= AE_{\text{entry}}(\ell') \cap AE_{\text{exit}}(\ell'') \\
AE_{\text{entry}}(\ell'') &= AE_{\text{exit}}(\ell') \\
AE_{\text{exit}}(\ell) &= AE_{\text{entry}}(\ell) \cup \{x + y\} \\
AE_{\text{exit}}(\ell') &= AE_{\text{entry}}(\ell') \\
AE_{\text{exit}}(\ell'') &= AE_{\text{entry}}(\ell'')
\end{align*}
\]
After some simplification, we find that:

\[ \text{AE}_{\text{entry}}(\ell') = (x+y) \cap \text{AE}_{\text{entry}}(\ell') \]

There are two solutions to this equation: \{x+y\} and \emptyset. Consideration of
the example and the definition of available expressions shows that the most
informative solution is \{x+y\} – the expression is available every time we enter
\( \ell' \). Thus we require the \textit{largest} solution to the equations.

Example 2.5 For the program

\[
[x := a + b]; [y := a + b]; \text{while } (y > a + b) \text{ do } ([a := a + 1]; [x := a + b])
\]

of Example 2.4, \textit{kill}_{AE} and \textit{gen}_{AE} are defined as follows:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>\textit{kill}_{AE}(\ell)</th>
<th>\textit{gen}_{AE}(\ell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\emptyset</td>
<td>{a+b}</td>
</tr>
<tr>
<td>2</td>
<td>\emptyset</td>
<td>{a+b}</td>
</tr>
<tr>
<td>3</td>
<td>\emptyset</td>
<td>{a+b}</td>
</tr>
<tr>
<td>4</td>
<td>{a+b, a+b, a+1}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>5</td>
<td>\emptyset</td>
<td>{a+b}</td>
</tr>
</tbody>
</table>

We get the following equations:

\[
\begin{align*}
\text{AE}_{\text{entry}}(1) &= \emptyset \\
\text{AE}_{\text{entry}}(2) &= \text{AE}_{\text{exit}}(1) \\
\text{AE}_{\text{entry}}(3) &= \text{AE}_{\text{exit}}(2) \cap \text{AE}_{\text{exit}}(5) \\
\text{AE}_{\text{entry}}(4) &= \text{AE}_{\text{exit}}(3) \\
\text{AE}_{\text{entry}}(5) &= \text{AE}_{\text{exit}}(4) \\
\text{AE}_{\text{exit}}(1) &= \text{AE}_{\text{entry}}(1) \cup \{a+b\} \\
\text{AE}_{\text{exit}}(2) &= \text{AE}_{\text{entry}}(2) \cup \{a+b\} \\
\text{AE}_{\text{exit}}(3) &= \text{AE}_{\text{entry}}(3) \cup \{a+b\} \\
\text{AE}_{\text{exit}}(4) &= \text{AE}_{\text{entry}}(4) \setminus \{a+b, a+b, a+1\} \\
\text{AE}_{\text{exit}}(5) &= \text{AE}_{\text{entry}}(5) \cup \{a+b\}
\end{align*}
\]

Using an analogue of the Chaotic Iteration discussed in Chapter 1 (starting
with \textit{AE}_{\text{exp}} rather than \emptyset) we can compute the following solution:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>\textit{AE}_{\text{entry}}(\ell)</th>
<th>\textit{AE}_{\text{exit}}(\ell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\emptyset</td>
<td>{a+b}</td>
</tr>
<tr>
<td>2</td>
<td>{a+b}</td>
<td>{a+b, a+b}</td>
</tr>
<tr>
<td>3</td>
<td>{a+b}</td>
<td>{a+b}</td>
</tr>
<tr>
<td>4</td>
<td>{a+b}</td>
<td>{a+b}</td>
</tr>
<tr>
<td>5</td>
<td>\emptyset</td>
<td>{a+b}</td>
</tr>
</tbody>
</table>

2.1 Intraprocedural Analysis

Note that, even though \( a \) is redefined in the loop, the expression \( a+b \)
evaluated in the loop and so it is always available on entry to the loop.
the other hand, \( a+b \) is available on the first entry to the loop but is
before the next iteration.

2.1.2 Reaching Definitions Analysis

As mentioned in Chapter 1, the \textit{Reaching Definitions Analysis} should
properly be called \textit{reaching assignments} but we will use the traditional
This analysis is analogous to the previous one except that we are inter-

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 construc-
tion of direct links between blocks that produce values and blocks that use
we shall return to this in Subsection 2.1.5.

Example 2.6 Consider the following program:

\[
[x := 0]; [y := 1]; \text{while } (z > 1) \text{ do } ([y := x + y]; [x := x - 1])
\]

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

The analysis is specified in Table 2.2. The function

\[
\text{kill}_{RD} : \text{Blocks} \rightarrow \mathcal{P}(\text{Var} \times \text{Lab}^2)
\]

produces the set of pairs of variables and labels of assignments that
are destroyed by the block. An assignment is destroyed if the block ass-
names a variable that has an initial value to the variable, i.e. the left hand side of the assignment. To
with uninitialised variables we shall, as in Chapter 1, use the special
\textit{?} and we set \textit{Lab}^2 = \textit{Lab} \cup \{?\}.

The function

\[
\text{gen}_{RD} : \text{Blocks} \rightarrow \mathcal{P}(\text{Var} \times \text{Lab}^2)
\]

produces the set of pairs of variables and labels of assignments generate
the block; only assignments generate definitions.

The analysis itself is now defined by the pair of functions \textit{RD}_{entry} and \textit{RD}
that mapping labels to sets of pairs of variables and labels (of assignment blocks)

\[
\textit{RD}_{entry} = \textit{RD}_{entry} \cup \textit{RD}_{exit} \cup \textit{RD}_{label}
\]
Once again, we concentrate on the entry of the block labelled \( \ell' \), \( R_{D\text{entry}}(\ell') \); after some simplification we get:

\[
R_{D\text{entry}}(\ell') = \{(x,?), (y,?), (z,\ell)\} \cup R_{D\text{entry}}(\ell')
\]

but this equation has many solutions: we can take \( R_{D\text{entry}}(\ell') \) to be any superset of \( \{(x,?), (y,?), (z,\ell)\} \). However, since \( \ell' \) does not generate any new definitions, the most precise solution is \( \{(x,?), (y,?), (z,\ell)\} \) - we require the smallest solution to the equations.

Sometimes, when the Reaching Definitions Analysis is presented in the literature, one has \( R_{D\text{entry}}(\text{init}(S_\ell)) = \emptyset \) rather than \( R_{D\text{entry}}(\text{init}(S_\ell)) = \{(x,?) | x \in FV(S_\ell)\} \). 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, as will emerge from Mini Project 2.8, is that it is always semantically sound.

**Example 2.7** The following table summarises the assignments killed and generated by each of the blocks in the program:

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

of Example 2.6:

<table>
<thead>
<tr>
<th>\ell</th>
<th>\text{kill}_{RD}(\ell)</th>
<th>\text{gen}_{RD}(\ell)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{(x,?), (y,?), (x,5)}</td>
<td>{(x,1)}</td>
</tr>
<tr>
<td>2</td>
<td>{(y,?), (y,2), (y,4)}</td>
<td>{(y,2)}</td>
</tr>
<tr>
<td>3</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>4</td>
<td>{(y,?), (y,2), (y,4)}</td>
<td>{(y,4)}</td>
</tr>
<tr>
<td>5</td>
<td>{(x,?), (x,1), (x,5)}</td>
<td>{(x,5)}</td>
</tr>
</tbody>
</table>

The analysis gives rise to the following equations:

\[
\begin{align*}
R_{D\text{entry}}(1) & = \{(x,?), (y,?)\} \\
R_{D\text{entry}}(2) & = R_{D\text{exit}}(1) \\
R_{D\text{entry}}(3) & = R_{D\text{exit}}(2) \cup R_{D\text{exit}}(5) \\
R_{D\text{entry}}(4) & = R_{D\text{exit}}(3) \\
R_{D\text{entry}}(5) & = R_{D\text{exit}}(4) \\
R_{D\text{exit}}(1) & = (R_{D\text{entry}}(1) \setminus \{(x,?), (x,1), (x,5)\}) \cup \{(x,1)\} \\
R_{D\text{exit}}(2) & = (R_{D\text{entry}}(2) \setminus \{(y,?), (y,2), (y,4)\}) \cup \{(y,2)\} \\
R_{D\text{exit}}(3) & = R_{D\text{exit}}(3) \\
R_{D\text{exit}}(4) & = (R_{D\text{entry}}(4) \setminus \{(y,?), (y,2), (y,4)\}) \cup \{(y,4)\} \\
R_{D\text{exit}}(5) & = (R_{D\text{entry}}(5) \setminus \{(x,?), (x,1), (x,5)\}) \cup \{(x,5)\}
\end{align*}
\]
Using Chaotic Iteration we may compute the solution:

\[
\begin{array}{c|c|c}
\ell & \text{RD}_{\text{entry}}(\ell) & \text{RD}_{\text{exit}}(\ell) \\
1 & \{(x,?), (y,?)\} & \{(y,?), (x,1)\} \\
2 & \{(y,?), (x,1)\} & \{(x,1), (y,2)\} \\
3 & \{(x,1), (y,2), (y,4), (x,5)\} & \{(x,1), (y,2), (y,4), (x,5)\} \\
4 & \{(x,1), (y,2), (y,4), (x,5)\} & \{(x,1), (y,4), (x,5)\} \\
5 & \{(x,1), (y,4), (x,5)\} & \{(y,4), (x,5)\} \\
\end{array}
\]

### 1.1.3 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.

**Example 2.8** Consider the program:

\[
\text{if } [a>b] \text{ then } [(x:=b-a)^2; (y:=a-b)^2] \text{ else } [(y:=b-a)^2; (x:=a-b)^2]
\]

The expressions \(a-b\) and \(b-a\) are both very busy at the start of the conditional; they can be hoisted to the start of the conditional resulting in a space saving in the size of the code generated for this program.

The analysis is specified in Table 2.3. We have already defined the notion of an expression being killed when we presented the Available Expressions analysis; we use an equivalent function here:

\[
\text{kil}_{\text{vb}} : \text{Blocks}_n \rightarrow \mathcal{P}(\text{AExp}_n)
\]

By analogy with the previous analyses, we also need to define how a block generates additional very busy expressions. For this we use:

\[
\text{gen}_{\text{vb}} : \text{Blocks}_n \rightarrow \mathcal{P}(\text{AExp}_n)
\]

All of the expressions that appear in a block are very busy at the entry to a block (unlike what was the case for Available Expressions).

### 2.1 Intraprocedural Analysis

#### kill and gen functions

\[
\begin{align*}
\text{kil}_{\text{vb}}([x := a]^\ell) & = \{a' \in \text{AExp}_n \mid x \in \text{FV}(a')\} \\
\text{kil}_{\text{vb}}([\text{skip}]^\ell) & = \emptyset \\
\text{kil}_{\text{vb}}([b]^\ell) & = \emptyset \\
\text{gen}_{\text{vb}}([x := a]^\ell) & = \text{AExp}(a) \\
\text{gen}_{\text{vb}}([\text{skip}]^\ell) & = \emptyset \\
\text{gen}_{\text{vb}}([b]^\ell) & = \text{AExp}(b)
\end{align*}
\]

#### data flow equations: \(\text{VB}^n\)

\[
\begin{align*}
\text{VB}_{\text{entry}}(\ell) & = \begin{cases} \\
\emptyset & \text{if } \ell \in \text{final}(S_n) \\
\cap \{\text{VB}_{\text{entry}}(\ell') \mid (\ell', \ell) \in \text{flow}^R(S_n)\} & \text{otherwise}
\end{cases} \\
\text{VB}_{\text{exit}}(\ell) & = \{\text{VB}_{\text{exit}}(\ell') \backslash \text{kil}_{\text{vb}}(B_f^\ell)\} \cup \text{gen}_{\text{vb}}(B_f^\ell)
\end{align*}
\]

Table 2.3: Very Busy Expressions Analysis.

The analysis itself is defined by the pair of functions \(\text{VB}_{\text{entry}}\) and \(\text{VB}_{\text{exit}}\) mapping labels to sets of expressions:

\[
\text{VB}_{\text{entry}} : \text{Lab}_n \rightarrow \mathcal{P}(\text{AExp}_n)
\]

For a label consistent program \(S_n\) (with isolated exits) they are defined as in Table 2.3.

The analysis is a _backward analysis_ and, as we shall see, we are interested in the largest sets satisfying the equation for \(\text{VB}_{\text{exit}}\). The functions propagate information _against_ the flow of the program: an expression is very busy at the exit from a block if it is very busy at the entry to every block that follows; if there are none the formula evaluates to \(\text{AExp}_n\). However, no expressions are very busy at the exit from any final block.

To motivate the fact that we require the largest set, we consider the situation where we have a flow graph as shown in Figure 2.3; this flow graph might correspond to the program:

\[
(\text{while } [x>1]^\ell \text{ do } [\text{skip}]^\ell; [x:=x+1]^\ell)
\]

The equations for this program are

\[
\begin{align*}
\text{VB}_{\text{entry}}(\ell) & = \text{VB}_{\text{exit}}(\ell) \\
\text{VB}_{\text{entry}}(\ell') & = \text{VB}_{\text{exit}}(\ell') \\
\text{VB}_{\text{entry}}(\ell'') & = \emptyset
\end{align*}
\]
Figure 2.3: A schematic flow graph (in reverse).

\[ \text{VB}_{\text{exit}(k)} = \text{VB}_{\text{entry}(k')} \cap \text{VB}_{\text{entry}(k'')} \]
\[ \text{VB}_{\text{exit}(l)} = \text{VB}_{\text{exit}(l')} \]
\[ \text{VB}_{\text{exit}(l'')} = \emptyset \]

and, for the exit conditions of \( k \), we calculate:

\[ \text{VB}_{\text{exit}(k)} = \text{VB}_{\text{exit}(k')} \cap \{x+1\} \]

Any subset of \( \{x+1\} \) is a solution but \( \{x+1\} \) is the most informative. Hence we want the largest solution to the equations.

**Example 2.9** To analyse the program

if \( [a>b] \) then \( \{x:=b-a\}; \{y:=a-b\} \) else \( \{y:=a-b\}; \{x:=a-b\}\)

of Example 2.8, we calculate the following killed and generated sets:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{kill}_{\text{VB}}(\ell) )</th>
<th>( \text{gen}_{\text{VB}}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\emptyset</td>
<td>{b-a}</td>
</tr>
<tr>
<td>2</td>
<td>\emptyset</td>
<td>{a-b}</td>
</tr>
<tr>
<td>3</td>
<td>{a-b}</td>
<td>{a-b}</td>
</tr>
<tr>
<td>4</td>
<td>{a-b}</td>
<td>{a-b}</td>
</tr>
<tr>
<td>5</td>
<td>\emptyset</td>
<td>{a-b}</td>
</tr>
</tbody>
</table>

We get the following equations:

\[ \text{VB}_{\text{entry}}(1) = \text{VB}_{\text{exit}}(1) \]
\[ \text{VB}_{\text{entry}}(2) = \text{VB}_{\text{exit}}(2) \cup \{b-a\} \]
\[ \text{VB}_{\text{entry}}(3) = \{a-b\} \]
\[ \text{VB}_{\text{entry}}(4) = \text{VB}_{\text{exit}}(4) \cup \{b-a\} \]
\[ \text{VB}_{\text{entry}}(5) = \{a-b\} \]

We can then use an analogue of Chaotic Iteration (starting with \( \text{AExp} \), rather than \( \emptyset \)) to compute:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{VB}_{\text{entry}}(\ell) )</th>
<th>( \text{VB}_{\text{exit}}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{a-b,b-a}</td>
<td>{a-b,b-a}</td>
</tr>
<tr>
<td>2</td>
<td>{a-b,b-a}</td>
<td>{a-b}</td>
</tr>
<tr>
<td>3</td>
<td>{a-b}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>4</td>
<td>{a-b,b-a}</td>
<td>{a-b}</td>
</tr>
<tr>
<td>5</td>
<td>{a-b}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

### 2.1.4 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.

**Example 2.10** Consider the following expression:

\[ [x:=2]; [y:=4]; [x:=1]; \text{if} \ [y>x] \text{ then} [z:=y^2]; [x:=z]^7 \]

The variable \( x \) is not live at the exit from label 1; the first assignment of the program is redundant. Both \( x \) and \( y \) are live at the exit from label 3.

The analysis is defined in Table 2.4. The variable that appears on the left-hand side of an assignment is killed by the assignment; tests and \text{skip} statements do not kill variables. This is expressed by the function:

\[ \text{kill}_{\text{LV}} : \text{Blocks} \rightarrow \mathcal{P}(\text{Var}) \]

The function
2 Data Flow Analysis

2.1 Intraprocedural Analysis

\[
LV_{\text{exit}}(\ell) = LV_{\text{entry}}(\ell) \cup LV_{\text{entry}}(\ell')
\]
\[
LV_{\text{exit}}(\ell') = LV_{\text{entry}}(\ell)
\]
\[
LV_{\text{exit}}(\ell'') = \emptyset
\]

Suppose that we are interested in \(LV_{\text{exit}}(\ell)\); after some calculation we get:
\[
LV_{\text{exit}}(\ell) = LV_{\text{exit}}(\ell) \cup \{x\}
\]

Any superset of \(\{x\}\) is a solution. Optimisations based on this analysis are based on "dead" variables – the smaller the set of live variables, the more optimisations are possible. Hence we shall be interested in the smallest solution \(\{x\}\) to the equations. Correctness of the analysis will be established in Section 2.2.

Example 2.11 Returning to the program
\[
[x:=2];[y:=4];[x:=1];(\text{if } [y>x] \text{ then } [z:=y] \text{ else } [z:=y^2];[x:=z])
\]
of Example 2.10, we can compute \(kill_{LV}\) and \(gen_{LV}\) as:

<table>
<thead>
<tr>
<th>(\ell)</th>
<th>(kill_{LV}(\ell))</th>
<th>(gen_{LV}(\ell))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>({x})</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>2</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>3</td>
<td>({x})</td>
<td>({y})</td>
</tr>
<tr>
<td>4</td>
<td>(\emptyset)</td>
<td>({x,y})</td>
</tr>
<tr>
<td>5</td>
<td>({x})</td>
<td>({y})</td>
</tr>
<tr>
<td>6</td>
<td>({x})</td>
<td>({y})</td>
</tr>
<tr>
<td>7</td>
<td>({x})</td>
<td>({x})</td>
</tr>
</tbody>
</table>

We get the following equations:
\[
LV_{\text{entry}}(1) = LV_{\text{exit}}(1) \setminus \{x\}
\]
\[
LV_{\text{entry}}(2) = LV_{\text{exit}}(2) \setminus \{y\}
\]
\[
LV_{\text{entry}}(3) = LV_{\text{exit}}(3) \setminus \{x\}
\]
\[
LV_{\text{entry}}(4) = LV_{\text{exit}}(4) \setminus \{x,y\}
\]
\[
LV_{\text{entry}}(5) = (LV_{\text{exit}}(5) \setminus \{z\}) \cup \{y\}
\]
\[
LV_{\text{entry}}(6) = (LV_{\text{exit}}(6) \setminus \{z\}) \cup \{y\}
\]
\[
LV_{\text{entry}}(7) = \{z\}
\]
\[
LV_{\text{exit}}(1) = LV_{\text{entry}}(2)
\]
\[
LV_{\text{exit}}(2) = LV_{\text{entry}}(3)
\]
\[
LV_{\text{exit}}(3) = LV_{\text{entry}}(4)
\]
\[
LV_{\text{exit}}(4) = LV_{\text{entry}}(5) \cup LV_{\text{entry}}(6)
\]
\[
LV_{\text{exit}}(5) = LV_{\text{entry}}(7)
\]
\[
LV_{\text{exit}}(6) = LV_{\text{entry}}(7)
\]
\[
LV_{\text{exit}}(7) = \emptyset
\]
We can then use Chaotic Iteration to compute the solution:

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>( \text{LV}_{\text{entry}}(\ell) )</th>
<th>( \text{LV}_{\text{exit}}(\ell) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>( {y} )</td>
<td>( {y} )</td>
</tr>
<tr>
<td>3</td>
<td>( {y} )</td>
<td>( {x,y} )</td>
</tr>
<tr>
<td>4</td>
<td>( {x,y} )</td>
<td>( {y} )</td>
</tr>
<tr>
<td>5</td>
<td>( {y} )</td>
<td>( {x} )</td>
</tr>
<tr>
<td>6</td>
<td>( {y} )</td>
<td>( {x} )</td>
</tr>
<tr>
<td>7</td>
<td>( {x} )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

Note that we have assumed that all variables are dead at the end of the program. Some authors assume that the variables of interest are output at the end of the program; in that case \( \text{LV}_{\text{entry}}(\ell) \) should be \( \{x, y, z\} \) which means that \( \text{LV}_{\text{entry}}(3), \text{LV}_{\text{exit}}(5) \) and \( \text{LV}_{\text{exit}}(6) \) should all be \( \{y, z\} \).

### 2.1 Intraprocedural Analysis

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.

In order to make these definitions more precise, we will use the notion of a definition clear path with respect to some variable. The idea is that \( \ell_1, \ldots, \ell_n \) is a definition clear path for \( x \) if none of the blocks labelled \( \ell_1, \ldots, \ell_n \) assigns a value to \( x \) and if \( \ell_n \) uses \( x \). Formally, for a label consistent program \( S \), we define the predicate clear:

\[
\text{clear}(x, \ell, \ell') = \exists_{\ell_1, \ldots, \ell_n}:
\begin{align*}
(\ell_1 = \ell) \land (\ell_n = \ell') \land (n > 0) \land \\
(\forall i \in \{1, \ldots, n-1\} : (\ell_i, \ell_{i+1}) \in \text{flow}(S_n)) \land \\
(\forall i \in \{1, \ldots, n-1\} : \lnot \text{def}(\ell_i, x)) \land \text{use}(x, \ell_n)
\end{align*}
\]

Here the predicate \( \text{use}(x, \ell) = (\exists B : [B]^\ell \in \text{blocks}(S_n) \land x \in \text{gen}_{\text{LV}}([B]^\ell)) \)

and the predicate \( \text{def}(x, \ell) = (\exists B : [B]^\ell \in \text{blocks}(S_n) \land x \in \text{kil}_{\text{LV}}([B]^\ell)) \)

Armed with these definitions, we can define the functions

\[
\text{ud, du} : \text{Var} \times \text{Lab} \rightarrow \mathcal{P}(\text{Lab})
\]

as follows:

\[
\text{ud}(x, \ell) = \{ \ell' | \text{def}(x, \ell) \land \exists \ell' : (\ell, \ell') \in \text{flow}(S_n) \land \text{clear}(x, \ell', \ell) \} \\
\cup \{ ? | \text{clear}(x, \text{init}(S_n), \ell) \}
\]

\[
\text{du}(x, \ell) = \begin{cases} 
\{ \ell' | \text{def}(x, \ell) \land \exists \ell' : (\ell, \ell') \in \text{flow}(S_n) \land \text{clear}(x, \ell', \ell) \} \\
\text{if } \ell \neq ? \\
\{ ? | \text{clear}(x, \text{init}(S_n), \ell) \} \\
\text{if } \ell = ?
\end{cases}
\]

So \( \text{ud}(x, \ell) \) will return the labels where an occurrence of \( x \) at \( \ell' \) might have obtained its value; this may be at a label \( \ell \in S_n \) or \( x \) may be uninitialised as indicated by the occurrence of \( ? \). And \( \text{du}(x, \ell) \) will return the labels where the value assigned to \( x \) at \( \ell \) might be used; again we distinguish between the case where \( x \) gets its value within the program and the case where it is uninitialised. It turns out that:

\[
\text{du}(x, \ell) = \{ \ell' | \ell \in \text{ud}(x, \ell) \}
\]

Before showing how \( \text{ud} \)- and \( \text{du} \)-chains can be used, we illustrate the functions by a simple example.

**Example 2.12** Consider the program:

\[
x := 0; \{ x := 0 \} ; \{ \text{if } [x := x] \text{ then } [x := 0] \text{ else } [x := x] \}; [y := x] ; [x := x + z]
\]

Then we get:

\[
\begin{array}{cccc}
\text{ud}(x, \ell) & x & y & z \\
1 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 \\
3 & 0 & 0 & \{?\} \\
4 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 \\
7 & 0 & 0 & \{4,5\} \\
\end{array}
\]

\[
\begin{array}{cccc}
\text{du}(x, \ell) & x & y & z \\
1 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 \\
3 & 0 & 0 & \{3,5,6\} \\
4 & 0 & 0 & 0 \\
5 & 0 & 0 & \{7\} \\
6 & 0 & 0 & \{7\} \\
7 & 0 & 0 & \{3\} \\
\end{array}
\]

The table for \( \text{ud} \) shows that the occurrence of \( x \) in block 3 will get its value in block 2 and the table for \( \text{du} \) shows that the value assigned to \( x \) in block 2 may be used in block 3, 5 and 6.

One application of \( \text{ud} \)- and \( \text{du} \)-chains is for Dead Code Elimination; for the program of Example 2.12 we may remove the block labelled 1 for example 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
2.2 Theoretical Properties

In this section we will show that the Live Variables Analysis of Subsection 2.1.4 is indeed correct; the correctness of the Reaching Definitions Analysis is the topic of Mini Project 2.2. We shall begin by presenting a formal semantics for WHILE.

The material of this section may be skimmed through on a first reading; however, it is frequently when conducting the correctness proof that the final and subtle errors in the analysis are found and corrected. In other words, proving the semantic correctness of the analysis should not be considered a dispensable development that is merely of interest for theoreticians.

2.2.1 Structural Operational Semantics

We choose to use a (so-called small step) Structural Operational Semantics because it allows us to reason about intermediate stages in a program execution and it also allows us to deal with non-terminating programs.

Configurations and transitions. First define a state as a mapping from variables to integers:

\[ \sigma \in \text{State} = \text{Var} \rightarrow \mathbb{Z} \]

A configuration of the semantics is either a pair consisting of a statement and a state or it is a state; a terminal configuration is a configuration that simply is a state. The transitions of the semantics are of the form

\[
\begin{align*}
A &: \text{AExp} \rightarrow (\text{State} \rightarrow \mathbb{Z}) \\
A[x] \sigma &= \sigma(x) \\
A[n] \sigma &= N[n] \\
A[a_1 \text{op}_a a_2] \sigma &= A[a_1] \sigma \text{ op}_a A[a_2] \sigma \\
B &: \text{BExp} \rightarrow (\text{State} \rightarrow \mathbb{T}) \\
B[\text{not } b] \sigma &= \neg B[b] \sigma \\
B[b_1 \text{ op}_b b_2] \sigma &= B[b_1] \sigma \text{ op}_b B[b_2] \sigma \\
B[a_1 \text{ op}_b a_2] \sigma &= A[a_1] \sigma \text{ op}_b A[a_2] \sigma
\end{align*}
\]

Table 2.5: Semantics of expressions in WHILE.

\((S, \sigma) \rightarrow \sigma'\) and \((S, \sigma) \rightarrow (S', \sigma')\)

and express how the configuration is changed by one step of computation. So in the configuration \((S, \sigma)\) one of two things may happen:

- the execution terminates after one step and we record that by giving the resulting state \(\sigma'\), or
- the execution does not terminate after one step and we record that by a new configuration \((S', \sigma')\) where \(S'\) is the rest of the program and \(\sigma'\) is the updated state.

To deal with arithmetic and boolean expressions we require the semantic functions

\[
\begin{align*}
A &: \text{AExp} \rightarrow (\text{State} \rightarrow \mathbb{Z}) \\
B &: \text{BExp} \rightarrow (\text{State} \rightarrow \mathbb{T})
\end{align*}
\]

whose definition are given in Table 2.5. Here we assume that \text{op}_n, \text{op}_b \text{ and op}_p are the semantic counterparts of the corresponding syntax. We have also assumed the existence of \(N : \text{Num} \rightarrow \mathbb{Z}\) which defines the semantics of numerals. For simplicity we have assumed that no errors can occur; this means that division by 0 will have to produce an integer for example. One can modify the definition so as to allow errors but this will complicate the correctness proof to be performed below. Note that the value of an expression is only affected by the variables appearing in it, that is:

- if \(\forall x \in \text{FV}(a) : \sigma_1(x) = \sigma_2(x)\) then \(A[a]_{\sigma_1} = A[a]_{\sigma_2}\)
- if \(\forall x \in \text{FV}(b) : \sigma_1(x) = \sigma_2(x)\) then \(B[b]_{\sigma_1} = B[b]_{\sigma_2}\)

These results can easily be proved by structural induction on expressions (or by mathematical induction on their size); see Appendix B for a brief introduction to these proof principles.
to true then the first step is to unroll the loop and the second axiom expresses that the execution terminates if the boolean expression evaluates to false.

Derivation sequences. A derivation sequence for a statement $S$ and a state $s_0$, can take one of two forms:

- It is a finite sequence of configurations $(s_1, s_2, \ldots, s_n, s_{n+1})$, satisfying $(s_i, s_{i+1})$ for $i = 1, \ldots, n-1$ and $(s_n, s_{n+1})$ corresponds to a terminating computation.
- It is an infinite sequence of configurations $(s_1, s_2, \ldots, s_i, s_{i+1})$ for all $i \geq 1$; this corresponds to a looping computation.

Example 2.13 We illustrate the semantics by showing an execution of the factorial program of Example 1.1. In the following, we assume that the state $s_{a, a, a}$ maps $x$ to $n_y$, $y$ to $n_y$, and $z$ to $n_z$. We then get the following finite derivation sequence:

$(x := 0; y := 0; x := x + 1; y := y \times x) \rightarrow \ldots \rightarrow (y := 0, s_{300})$

Note that labels have no impact on the semantics: they are merely carried along and never inspected.

Properties of the semantics. We shall first establish a number of properties of the operations on programs and labels that we have used in the formulation of the analyses. In the course of the computation the set of
flows, the set of final labels and the set of elementary blocks of the statements of the configurations will be modified; Lemma 2.14 shows that the sets will decrease:

Lemma 2.14

(i) If \( \langle S, \sigma \rangle \rightarrow \sigma' \) then \( \text{final}(S) = \{ \text{init}(S) \} \).

(ii) If \( \langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle \) then \( \text{final}(S) \supseteq \text{final}(S') \).

(iii) If \( \langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle \) then \( \text{flow}(S) \supseteq \text{flow}(S') \).

(iv) If \( \langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle \) then \( \text{blocks}(S) \supseteq \text{blocks}(S') \) and if \( S \) is label consistent then so is \( S' \).

Proof. The proof of (i) is by induction on the shape of the inference tree used to establish \( \langle S, \sigma \rangle \rightarrow \sigma' \); we refer to Appendix B for a brief introduction to the proof principle. Consulting Table 2.6 we see that there are three non-vacuous cases:

The case [aa]. Then \( \langle x := a\rangle, \sigma \rangle \rightarrow \sigma [x := a[\sigma] \rangle \) and we get:

\[
\text{final}[x := a]\rangle = \{ \ell \} = \{ \text{init} \}
\]

The case [skip]. Then \( \langle \text{skip} \rangle, \sigma \rangle \rightarrow \sigma \) and we get:

\[
\text{final}[\text{skip}] = \{ \ell \} = \{ \text{init} \}
\]

The case [while]. Then \( \langle \text{while } [b] \text{ do } S, \sigma \rangle \rightarrow \sigma \) because \( B[b] \sigma \text{ is false } \) and we get:

\[
\text{final}[\text{while } [b] \text{ do } S] = \{ \ell \} = \{ \text{init} \}
\]

This completes the proof of (i).

The proof of (ii) is by induction on the shape of the inference tree used to establish \( \langle S, \sigma \rangle \rightarrow \langle S', \sigma' \rangle \). There are five non-vacuous cases:

The case [seq1]. Then \( \langle S_1; S_2, \sigma \rangle \rightarrow \langle S_1; S_2, \sigma' \rangle \) because \( \langle S_1, \sigma \rangle \rightarrow \langle S_1', \sigma' \rangle \) and we get:

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

The case [seq2]. Then \( \langle S_1; S_2, \sigma \rangle \rightarrow \langle S_2, \sigma' \rangle \) because \( \langle S_1, \sigma \rangle \rightarrow \sigma' \) and we get:

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

The case [if]. Then \( \langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle \) because \( B[b] \) is true and we get:

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

The case [if2] is similar to the previous case.

The case [while]. Then \( \langle \text{while } [b] \text{ do } S, \sigma \rangle \rightarrow \langle S; \text{while } [b] \text{ do } S, \sigma \rangle \) because \( B[b] \sigma \) is true and we get:

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

This completes the proof of (ii).

The proof of (iii) is by induction on the shape of the inference tree used to establish \( \langle S, \sigma \rangle \rightarrow \sigma' \). There are five non-vacuous cases:

The case [seq1]. Then \( \langle S_1; S_2, \sigma \rangle \rightarrow \langle S_1; S_2, \sigma' \rangle \) because \( \langle S_1, \sigma \rangle \rightarrow \langle S_1', \sigma' \rangle \) and we get:

\[
\text{flow}(S_1; S_2) = \text{flow}(S_1) \cup \text{flow}(S_2) \cup \{ \ell \mid \text{init}(S_1) \} \quad \text{if } \ell \in \text{final}(S_1)
\]

\[
\text{flow}(S_1') \cup \text{flow}(S_2) \cup \{ \ell \mid \text{init}(S_1) \} \quad \text{if } \ell \in \text{final}(S_1)
\]

\[
\text{flow}(S_1'; S_2) = \text{flow}(S_1') \cup \text{flow}(S_2)
\]

where we have used the induction hypothesis and (ii).

The case [seq2]. Then \( \langle S_1; S_2, \sigma \rangle \rightarrow \langle S_2, \sigma' \rangle \) because \( \langle S_1, \sigma \rangle \rightarrow \sigma' \) and we get:

\[
\text{flow}(S_1; S_2) = \text{flow}(S_1) \cup \text{flow}(S_2) \cup \{ \ell \mid \text{init}(S_1) \} \quad \text{if } \ell \in \text{final}(S_1)
\]

\[
\text{flow}(S_2)
\]

The case [if1]. Then \( \langle \text{if } b \text{ then } S_1 \text{ else } S_2, \sigma \rangle \rightarrow \langle S_1, \sigma \rangle \) because \( B[b] \sigma = \text{true} \) and we get:

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

\[
\cup \{ \ell \mid \text{init}(S_1), \text{init}(S_2) \}
\]

\[
\cup \{ \ell \mid \text{init}(S_1), \text{init}(S_2) \}
\]

\[
\cup \{ \ell \mid \text{init}(S_1), \text{init}(S_2) \}
\]

\[
\text{flow}(S_1)
\]

The case [if2] is similar to the previous case.

The case [while]. Then \( \langle \text{while } [b] \text{ do } S, \sigma \rangle \rightarrow \langle S; \text{while } [b] \text{ do } S, \sigma \rangle \) because \( B[b] \sigma = \text{true} \) and we get:

\[
\text{flow}(S; \text{while } [b] \text{ do } S) = \text{flow}(S) \cup \text{flow}(S) \cup \{ \ell \mid \text{init}(S) \}
\]

\[
\cup \{ \ell \mid \text{init}(S) \}
\]

\[
\cup \{ \ell \mid \text{init}(S) \}
\]

\[
\text{flow}(S)
\]

This completes the proof of (iii).

The proof of (iv) is similar to that of (iii) and we omit the details.

2.2.2 Correctness of Live Variables Analysis

Preservation of solutions. Subsection 2.1.4 shows how to define an equation system for a label consistent program \( S_\ast \).
2 Data Flow Analysis

refer to this system as \( \mathbb{L}V^\mathcal{E}(S_a) \). The construction of \( \mathbb{L}V^\mathcal{E}(S_a) \) can be modified to give a constraint system \( \mathbb{L}V^\mathcal{E}(S_a) \) of the form studied in Subsection 1.3.2:

\[
\mathbb{L}V_{\text{exit}}(\ell) \supseteq \begin{cases} 
0 & \text{if } \ell \in \text{final}(S_a) \\
\bigcup \mathbb{L}V_{\text{entry}}(\ell') & \ell' \in \text{flow}^a(S_a) \end{cases}
\]

\[
\mathbb{L}V_{\text{entry}}(\ell) \supseteq \left( \mathbb{L}V_{\text{exit}}(\ell) \setminus \text{kill}_{\mathbb{L}}(B'f) \right) \cup \text{gen}_{\mathbb{L}}(B'f)
\]

where \( B'f \in \text{blocks}(S_a) \).

We make this definition because in the correctness proof we want to use the same solution for all statements derived from \( S_a \); this will be possible for \( \mathbb{L}V^\mathcal{E}(S_a) \) but not for \( \mathbb{L}V^a(S_a) \).

Now consider a collection \( \text{live} \) of functions:

\[
\text{live}_{\text{entry}}, \text{live}_{\text{exit}} : \mathcal{L}_{\mathbb{L}} \to \mathcal{P}(\mathcal{V}_{\mathbb{L}})
\]

We say that \( \text{live} \) solves \( \mathbb{L}V^a(S_a) \) and write

\[
\text{live} \models \mathbb{L}V^\mathcal{E}(S_a)
\]

if the functions satisfy the equations; similarly we write

\[
\text{live} \models \mathbb{L}V^\mathcal{E}(S_a)
\]

if \( \text{live} \) solves \( \mathbb{L}V^\mathcal{E}(S_a) \). The following result shows that any solution of the equation system is also a solution of the constraint system and that the least solutions of the two systems coincide.

Lemma 2.15 Consider a label consistent program \( S_a \). If \( \text{live} \models \mathbb{L}V^\mathcal{E}(S_a) \) then \( \text{live} \models \mathbb{L}V^\mathcal{E}(S_a) \). The least solution of \( \mathbb{L}V^\mathcal{E}(S_a) \) coincides with the least solution of \( \mathbb{L}V^\mathcal{E}(S_a) \).

Proof If \( \text{live} \models \mathbb{L}V^\mathcal{E}(S_a) \) then clearly \( \text{live} \models \mathbb{L}V^\mathcal{E}(S_a) \) because \( \gg \) includes the case of \( = \).

Next let us prove that \( \mathbb{L}V^\mathcal{E}(S_a) \) and \( \mathbb{L}V^\mathcal{E}(S_a) \) have the same least solution. We gave a constructive proof of a related result in Chapter I (under some assumptions about finiteness) so let us here give a more abstract proof using more advanced fixed point theory (as covered in Appendix A). In the manner of Chapter I we construct a function \( F^\mathbb{L}_\mathbb{L}(\text{live}) \) such that:

\[
\text{live} \models \mathbb{L}V^\mathcal{E}(S_a) \iff \text{live} \supseteq F^\mathbb{L}_\mathbb{L}(\text{live})
\]

\[
\text{live} \models \mathbb{L}V^\mathcal{E}(S_a) \iff \text{live} = F^\mathbb{L}_\mathbb{L}(\text{live})
\]

Using Tarski's Fixed Point Theorem (Proposition A.19) we now have that \( F^\mathbb{L}_\mathbb{L}(\text{live}) \) has a least fixed point \( lfp(F^\mathbb{L}_\mathbb{L}(\text{live})) \) such that

\[
lfp(F^\mathbb{L}_\mathbb{L}(\text{live})) = \bigcap \{ \text{live} \mid \text{live} \supseteq F^\mathbb{L}_\mathbb{L}(\text{live}) \} = \bigcap \{ \text{live} \mid \text{live} = F^\mathbb{L}_\mathbb{L}(\text{live}) \}
\]

We now have the following corollary expressing that the solution to the constraints of \( \mathbb{L}V^\mathcal{E} \) is preserved during computation; this is illustrated in Figure 2.4 for finite computations.

Corollary 2.17 If \( \text{live} \models \mathbb{L}V^\mathcal{E}(S_a) \) (for \( S_a \) being label consistent) and if \( (S_a, \sigma) \rightarrow (S'_a, \sigma') \) then also \( \text{live} \models \mathbb{L}V^\mathcal{E}(S'_a) \).

Proof Follows from Lemma 2.14 and 2.15.

We also have an easy result relating entry and exit components of a solution.

Lemma 2.18 If \( \text{live} \models \mathbb{L}V^\mathcal{E}(S_a) \) (with \( S_a \) being label consistent) then for all \( (\ell, \ell') \in \text{flow}(S_a) \) we have \( \text{live}_{\text{exit}}(\ell) \supseteq \text{live}_{\text{entry}}(\ell') \).

Proof The result follows immediately from the construction of \( \mathbb{L}V^\mathcal{E}(S_a) \).

Correctness relation. Intuitively, the correctness result for the Live Variables Analysis should express that the sets of live variables computed by
2.2 Theoretical Properties

Lemma 2.20 Assume \( \text{live} \models L^S(S) \) with \( S \) being label consistent. Then \( \sigma_1 \sim_{N(t)} \sigma_2 \) implies \( \sigma_1 \sim_{N(t)} \sigma_2 \) for all \( (t, t') \in \text{flow}(S) \).

Proof. Follows directly from Lemma 2.18 and the definition of \( \sim_{N(t)} \).

Correctness result. We are now ready for the main result. It states how semantically correct liveness information is preserved under each step of the execution: \( i \) in the case where we do not immediately terminate and \( ii \) in the case where we do immediately terminate.

Theorem 2.21
If \( \text{live} \models L^S(S) \) (with \( S \) being label consistent) then:

\( i \) if \( (S, \sigma_1) \rightarrow (S', \sigma_1') \) and \( \sigma_1 \sim_{N(\text{init}(S))} \sigma_2 \) then there exists \( \sigma_2' \) such that \( (S, \sigma_2) \rightarrow (S', \sigma_2') \) and \( \sigma_1' \sim_{N(\text{init}(S'))} \sigma_2' \). and

\( ii \) if \( (S, \sigma_1) \rightarrow \sigma_1' \) and \( \sigma_1 \sim_{N(\text{init}(S))} \sigma_2 \) then there exists \( \sigma_2' \) such that \( (S, \sigma_2) \rightarrow \sigma_2' \) and \( \sigma_1' \sim_{N(\text{init}(S))} \sigma_2' \).

Proof. The proof is by induction on the shape of the inference tree used to establish \( (S, \sigma_1) \rightarrow (S', \sigma_1') \) and \( (S, \sigma_1) \rightarrow \sigma_1' \), respectively.

The case \( \text{New} \). Then \( ([x] = a), \sigma_1) \rightarrow \sigma_1' [x] = \text{New}[a] \sigma_1 \) and from the specification of the constraint system we have

\[ N(t) = \text{live}_{\text{entry}}(t) = \left( \text{live}_{\text{entry}}(t), \{x\} \right) \cup \text{FV}(a) = (X(t), \{x\}) \cup \text{FV}(a) \]

and thus

\[ \sigma_1 \sim_{N(t)} \sigma_1 \text{ implies } \text{New}[a] \sigma_1 \sim_{N(t)} \sigma_2 \]

because the value of \( a \) is only affected by the variables occurring in it. Therefore, taking

\[ \sigma_2' = \sigma_2 | x = \text{New}(a) \sigma_1 \]

we have that \( \sigma_1'(x) = \sigma_2'(x) \), and thus \( \sigma_1' \sim_{N(t)} \sigma_2' \).

The case \( \text{Skip} \). Then \( ([\text{skip}], \sigma_1) \rightarrow \sigma_1' \), and from the specification of the constraint system

\[ N(t) = \text{live}_{\text{entry}}(t) = \left( \text{live}_{\text{entry}}(t), \emptyset \right) \cup \emptyset = \text{live}_{\text{entry}}(t) = X(t) \]

and we take \( \sigma_2' \) to be \( \sigma_2' \).

The case \( \text{Reg} \). Then \( (S_1, S_2, \sigma_1) \rightarrow (S_1', S_2, \sigma_1') \) because \( (S_1, \sigma_1) \rightarrow (S_1', \sigma_1') \). By construction we have

\[ \text{flow}(S_1; S_2) \supseteq \text{flow}(S_1) \] and also \( \text{blocks}(S_1; S_2) \supseteq \text{blocks}(S_1) \).

Thus by Lemma 2.16, \( \text{live} \) is a solution to \( L^S(S_1) \) and thus by the induction hypothesis there exists \( \sigma_2' \) such that

\[ (S_1, \sigma_2) \rightarrow (S_1', \sigma_2') \text{ and } \sigma_1' \sim_{N(\text{init}(S_1))} \sigma_2' \]

and the result follows.
2.3 Monotone Frameworks

Despite the differences between the analyses presented in Section 2.1, 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, as we will see in Section 2.4.

The overall pattern. Each of the four classical analyses (presented in Subsection 2.1.1 to 2.1.4) considers equations for a label consistent program $S$, and they take the form

$$Analysis_i(l) = \begin{cases} \bigcup \{Analysis_i(l') \mid (l', l) \in F\} & \text{if } l \in E \\ I(l) & \text{otherwise} \end{cases}$$

where

- $I$ is $\bigcap$ or $\bigcup$ (and $l$ is $U$ or $c$),
- $F$ is either $flow(S)$ or $flow^N(S)$,
- $E$ is $\{init(S)\}$ or $final(S)$,
- $l$ specifies the initial or final analysis information, and
- $f_l$ is the transfer function associated with $B_l \in blocks(S)$.

We now have the following characterisation:
2.3 Monotone Frameworks

2.3.1 Basic Definitions

Property spaces. One important ingredient in the framework is the property space, \( L \), used to represent the data flow information as well as the combination operator, \( \sqcup : \mathcal{P}(L) \to L \), that combines information from different paths; as usual \( \sqcup : L \times L \to L \) is defined by \( L \sqcup L' \equiv \{ \{L \} \} \) and we write \( \sqcup \) for \( \sqcup \emptyset \). It is customary to demand that this property space is in fact a complete lattice; as discussed in Appendix A this just means that it is a partially ordered set, \( (L, \sqsubseteq) \), such that each subset, \( Y \), has a least upper bound, \( \sqcup Y \). Looking ahead to the task of implementing the analysis one often requires that \( L \) satisfies the *Ascending Chain Condition*; as discussed in Appendix A this means that each ascending chain, \( (l_n)_n \), i.e., \( l_n \subseteq l_{n+1} \subseteq \cdots \), eventually stabilises, i.e., \( \exists n : l_n = l_{n+1} = \cdots \).

Example 2.23 For Reaching Definitions we have \( L = \mathcal{P}(\text{Var} \times \text{Lab}_\ast) \) and it is partially ordered by subset inclusion, i.e., \( \sqsubseteq \sim \sqsubseteq \). Similarly, \( \sqcup Y \) is \( \sqcup Y \), \( l \sqcup l' \equiv l \sqcup l' \), and \( \sqsubseteq = \emptyset \). That \( L \) satisfies the Ascending Chain Condition, i.e., that \( l_n \subseteq l_n \subseteq \cdots \), implies \( \exists n : l_n = l_{n+1} = \cdots \), follows because \( \text{Var} \times \text{Lab}_\ast \) is finite (unlike \( \text{Var} \times \text{Lab} \)).

Example 2.24 For AExp expressions we have \( L = \mathcal{P}(\text{AExp}_\ast) \) and it is partially ordered by subset inclusion, i.e., \( \sqsubseteq \sim \sqsubseteq \). Similarly, \( \sqcup Y \) is \( \sqcup Y \), \( l \sqcup l' \equiv l \sqcup l' \), and \( \sqsubseteq = \emptyset \). That \( L \) satisfies the Ascending Chain Condition, i.e., that \( l_n \sqsubseteq l_n \sqsubseteq \cdots \), implies \( \exists n : l_n = l_{n+1} = \cdots \), follows because \( \text{AExp}_\ast \) is finite (unlike \( \text{AExp} \)).

Remark. Historically, the demands on the property space, \( L \), have often been expressed in a different way. A join semi-lattice is a non-empty set, \( L \), with a binary join operation, \( \sqcup \), which is idempotent, commutative and associative, i.e., \( l \sqcup l = l \), \( l \sqcup l' = l' \sqcup l \), and \( (l \sqcup l') \sqcup l' = l \sqcup (l' \sqcup l') \). The commutativity and associativity of the operation mean that it does not matter in which order we combine information from different paths. The join operation induces a partial ordering, \( \sqsubseteq \), on the elements by taking \( l \sqsubseteq l' \) if and only if \( l \sqcup l' = l \). It is not hard to show that this in fact defines a partial ordering and that \( l \sqcup l' \) is the least upper bound (with respect to \( \sqsubseteq \)). A unit for the join operation is an element, \( \bot \), such that \( \bot \sqcup l = l \). It is
not hard to show that the unit is in fact the least element (with respect to \( \sqsubseteq \)). It has been customary to demand that the property space, \( L \), is a join semi-lattice with a unit and that it satisfies the Ascending Chain Condition. As proved in Lemma A.8 of Appendix A this is equivalent to our assumption that the property space, \( L \), is a complete lattice satisfying the Ascending Chain Condition.

Some formulations of Monotone Frameworks are expressed in terms of property spaces satisfying a Descending Chain Condition and using a combination operator \([\cdot]\). It follows from the principle of lattice duality (see the Concluding Remarks of Chapter 4) that this does not change the notion of Monotone Framework.

Transfer functions. Another important ingredient in the framework is the set of transfer functions, \( f_\ell : L \rightarrow L \) for \( \ell \in \text{Lab}_b \). It is natural to demand that each transfer function is monotone, i.e. \( l \sqsubseteq l' \) implies \( f_\ell(l) \sqsubseteq f_\ell(l') \). Intuitively, this says that an increase in our knowledge about the input must give rise to an increase in our knowledge about the output (or at least that we know the same as before). Formally, we shall see that monotonicity is of importance for the algorithms we develop. To control the set of transfer functions we demand that there is a set \( \mathcal{F} \) of monotone functions over \( L \), fulfilling the following conditions:

- \( \mathcal{F} \) contains all the transfer functions \( f_\ell \) in question,
- \( \mathcal{F} \) contains the identity function \( id \), and
- \( \mathcal{F} \) is closed under composition of functions.

The condition on the identity function is natural because of the skip statement and the condition on composition of functions is natural because of the sequencing of statements. Clearly one can take \( \mathcal{F} \) to be the space of monotone functions over \( L \) but it is occasionally advantageous to consider a smaller set because it makes it easier to find compact representations of the functions.

Some formulations of Monotone Frameworks associate transfer functions with edges (or flows) rather than nodes (or labels). A similar effect can be obtained using the approach of Exercise 2.11.

Frameworks. In summary, a Monotone Framework consists of:

- a complete lattice, \( L \), that satisfies the Ascending Chain Condition, and we write \( \sqcup \) 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.

Note that we do not demand that \( \mathcal{F} \) is a complete lattice or even a partially ordered set although this is the case for the set of all monotone functions from \( L \) to \( L \) (see Appendix A).

A somewhat stronger concept is that of a Distributive Framework. This is a Monotone Framework where additionally all functions \( f \) in \( \mathcal{F} \) are required to be distributive:

\[
(f_1 \sqcup f_2) = f((f_1 \sqcup f_2))
\]

Since \( f((f_1 \sqcup f_2)) \sqsubseteq f_1 \sqcup f_2 \), follows from monotonicity, the only additional demand is that \( f((f_1 \sqcup f_2)) \subseteq f_1 \sqcup f_2 \). When this condition is fulfilled it is sometimes possible to get more efficient algorithms.

Instances. The data flow equations make it clear that more than just a Monotone (or Distributive) Framework is needed in order to specify an analysis. To this end we define an instance, Analysis, of a Monotone (or Distributive) Framework to consist of:

- the complete lattice, \( L \), of the framework;
- the space of functions, \( \mathcal{F} \), of the framework;
- a finite flow, \( F \), that typically is \( \text{flow}(S) \) or \( \text{flow}^P(S) \);
- a finite set of so-called extremal labels, \( E \), that typically is \( \text{init}(S) \) or \( \text{find}(S) \);
- an extremal value, \( v \in L \), for the extremal labels; and
- a mapping, \( f \), from the labels \( \text{Lab}_b \) of \( F \) and \( E \) to transfer functions in \( \mathcal{F} \).

The instance then gives rise to a set of equations, \( \text{Analysis}_t \), of the form considered earlier:

\[
\text{Analysis}_t(\ell) \downarrow \{(\text{Analysis}_t(\ell'), (\ell', \ell) \in F) \cup t'_E
\]

where \( t'_E = \{ * \text{ if } \ell \in E \}

\[
\text{Analysis}(\ell) = f_t(\text{Analysis}_t(\ell))
\]

It also gives rise to a set of constraints, \( \text{Analysis}_E \), defined by:

\[
\text{Analysis}_E(\ell) \downarrow \{(\text{Analysis}_E(\ell'), (\ell', \ell) \in F) \cup t'_E
\]

where \( t'_E = \{ * \text{ if } \ell \in E \}

\[
\text{Analysis}_E(\ell) = f_t(\text{Analysis}_E(\ell))
\]
2.3 Monotone Frameworks

Lemma 2.25 Each of the four data flow analyses in Figure 2.6 is a Monotone Framework as well as a Distributive Framework.

Proof. To prove that the analyses are Monotone Frameworks we just have to confirm that \( F \) has the necessary properties.

The functions of \( F \) are monotone. Assume that \( l \leq l' \). Then \((l \setminus l) \subseteq (l' \setminus l)\) and, therefore \((l \setminus l) \cup l) \subseteq (l' \setminus l) \cup l_2\) and thus \(f(l) \leq f(l')\) as required.

Note that this calculation is valid regardless of whether \( \subseteq \) is \( \leq \) or \( \geq \).

The identity function is in \( F \): It is obtained by taking both \( l_1 \) and \( l_2 \) to be \( \emptyset \). The functions of \( F \) are closed under composition: Suppose \( f(l) = (l \setminus l) \cup l_2 \) and \( f'(l) = (l' \setminus l) \cup l_2'\). Then we calculate:

\[
(f \circ f')(l) = (((l \setminus l) \cup l_2) \setminus l) \cup l_2
\]

So \((f \circ f')(l) = (l \setminus l') \cup l_2'\) where \(l_2' = l_2 \setminus l_1\) and \(l_2' = (l_2 \setminus l_1) \cup l_2\). This completes the proof of the last part of the lemma.

To prove that the analyses are Distributive Frameworks consider \( f \in F \) given by \( f(l) = (l \setminus l) \cup l_2 \). Then we have:

\[
f(l \cup f') = (l \cup f') \setminus l_2 \cup l_2
\]

Note that the above calculation is valid regardless of whether \( \cup \) is \( \cup \) or \( \sqcup \). This completes the proof.

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.

Example 2.26 Let us return to the Available Expressions Analysis of the program

\[ [x := a+b]; [y := a+b]^2; \text{while } [y > a+b]^3 \text{ do } ([a := a+1]; [x := a+b]^2) \]

of Examples 2.4 and 2.5 and let us specify it as an instance of the associated Monotone Framework. The complete lattice of interest is

\[ \mathcal{P}\{a+b, a+b, a+1\}, \supseteq \]

with least element \( \{a+b, a+b, a+1\} \). The set of transfer functions has the form shown in Figure 2.6.

The instance of the framework additionally has the flow \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 3)\} and the set of external labels is \{1\}. The extremal value is \( \emptyset \)
and the transfer functions associated with the labels are

\[
\begin{align*}
    f^1_{AE}(Y) & = Y \cup \{a+b\} \\
    f^2_{AE}(Y) & = Y \cup \{a+b\} \\
    f^3_{AE}(Y) & = Y \cup \{a\} \\
    f^4_{AE}(Y) & = Y \setminus \{a+b, a+1\} \\
    f^5_{AE}(Y) & = Y \cup \{a\}
\end{align*}
\]

for \( Y \subseteq \{a+b, a, a+1\} \).

2.3.3 A Non-distributive Example

Lest the reader should imagine that all Monotone Frameworks are Distributive Frameworks, here we present one that is not. 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.

The Constant Propagation framework. The complete lattice used for Constant Propagation Analysis of a program, \( S_1 \), is

\[
\text{State}_{CP} = (\text{Var}_*, Z^T, \sqsubseteq, \cup, \cap, \bot, \lambda, T)
\]

where \( \text{Var}_* \) is the set of variables appearing in the program and \( Z^T = Z \cup \{T\} \) is partially ordered as follows:

\[
\begin{align*}
    \forall z \in Z^T : z \subseteq T \\
    \forall z_1, z_2 \in Z : (z_1 \subseteq z_2) \Rightarrow (z_1 = z_2)
\end{align*}
\]

The top element of \( Z^T \) is used to indicate that a variable is non-constant and all other elements indicate that the value is that particular constant. The idea is that an element \( \bar{\delta} \in \text{Var}_* \to Z^T \) is a property state: for each variable \( x \), \( \bar{\delta}(x) \) will give information about whether or not \( x \) is a constant and in the latter case which constant.

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

\[
\begin{align*}
    \forall \bar{\delta} \in (\text{Var}_* \to Z^T) : \quad \bot \sqsubseteq \bar{\delta} \\
    \forall \bar{\delta}_1, \bar{\delta}_2 \in \text{Var}_* \to Z^T : \quad \bar{\delta}_1 \sqsubseteq \bar{\delta}_2 \text{ if } \forall x : \bar{\delta}_1(x) \sqsubseteq \bar{\delta}_2(x)
\end{align*}
\]

\[\text{ACP} : \text{AExp} \to (\text{State}_{CP} \to Z^T)\]

\[
\begin{align*}
    \text{ACP}[x][\bar{\delta}] & = \begin{cases} 
        \bot & \text{if } \bar{\delta} = \bot \\
        \bar{\delta}(x) & \text{otherwise}
    \end{cases} \\
    \text{ACP}[n][\bar{\delta}] & = \begin{cases} 
        \bot & \text{if } \bar{\delta} = \bot \\
        n & \text{otherwise}
    \end{cases} \\
    \text{ACP}[a_1 \ op_a a_2][\bar{\delta}] & = \text{ACP}[a_1][\bar{\delta}] \ op_a \text{ACP}[a_2][\bar{\delta}]
\end{align*}
\]

\[
\begin{align*}
    [x := a] : & \ f^0_{CP}(\bar{\delta}) = \begin{cases} 
        \bot & \text{if } \bar{\delta} = \bot \\
        \bar{\delta}[x \mapsto \text{ACP}[a][\bar{\delta}]] & \text{otherwise}
    \end{cases} \\
    [\text{skip}] : & \ f^1_{CP}(\bar{\delta}) = \bar{\delta} \\
    [0] : & \ f^2_{CP}(\bar{\delta}) = \bar{\delta}
\end{align*}
\]

Table 2.7: Constant Propagation Analysis.

and the binary least upper bound operation is then:

\[
\forall \bar{\delta}_1, \bar{\delta}_2 \in \text{Var}_* \to Z^T : \forall x : (\bar{\delta}_1 \sqcup \bar{\delta}_2)(x) = \bar{\delta}_1(x) \sqcup \bar{\delta}_2(x)
\]

In contrast to the earlier examples, we define the transfer functions as follows:

\[\mathcal{F}_{CP} = \{ f | f \text{ is a monotone function on State}_{CP} \}\]

It is easy to verify that \( \text{State}_{CP} \) and \( \mathcal{F}_{CP} \) satisfy the requirements of being a Monotone Framework (see Exercise 2.8).

Constant Propagation is a forward analysis, so for the program \( S_1 \), we take the flow, \( F_1 \), to be \( \text{flow}(S_1) \), the external labels, \( E_1 \), to be \( \{\text{init}(S_1)\} \), the extremal value, \( \lceil e \rceil \), to be \( \lambda, T \), and the mapping, \( f^0_{CP} \), of labels to transfer functions is given in Table 2.7. The specification of the transfer functions uses the function

\[\text{ACP} : \text{AExp} \to (\text{State}_{CP} \to Z^T)\]

for analysing expressions. Here the operations on \( Z \) are lifted to \( Z^T = Z \cup \{\bot, T\} \) by taking \( z_1 \ lop_a z_2 = z_1 \ op_a z_2 \) if \( z_1, z_2 \in Z \) (and where \( \text{op}_a \) is the corresponding arithmetic operation on \( Z \)), \( z_1 \ lop_a \bot = \bot \) if \( z_1 = \bot \) or \( z_2 = \bot \) and \( z_2 \ lop_a \bot = T \) otherwise.

Lemma 2.27 Constant Propagation is a Monotone Framework that is not a Distributive Framework.

Proof The proof that Constant Propagation is a Monotone Framework is left for Exercise 2.8. To show that it is not a Distributive Framework consider the transfer
2.4 Equation Solving

Having set up a framework, there remains the question of how to use the framework to obtain an analysis result. In this section we shall consider two approaches. One is an iterative algorithm in the spirit of Chaotic Iteration as presented in Section 1.7. The other more directly propagates analysis information along paths in the program.

2.4.1 The MFP Solution

We first present a general iterative algorithm for Monotone Frameworks that computes the least solution to the data flow equations. Historically, this is called the MFP solution (for Maximal Fixed Point) although it in fact computes the least fixed point; the reason is that the classical literature tends to focus on analyses where U is $\cap$ (and because the least fixed point with respect to $\subseteq$ or $\supseteq$ then equals the greatest fixed point with respect to $\subseteq$).

The algorithm, written in pseudo-code in Table 2.8, takes as input an instance of a Monotone Framework. It uses an array, Analysis, which contains the Analysis information for each elementary block; the array is indexed by labels. It also uses a worklist $W$ which is a list of pairs; each pair is an element of the flow relation $F$. The presence of a pair in the worklist indicates that the analysis has changed at the exit of (or entry to -- for backward analyses) the block labelled by the first component and so must be recomputed at the entry to (or exit from) the block labelled by the second component. As a final stage the algorithm presents the result ($\text{MFP}_0$, $\text{MFP}_*$) of the analysis in a form close to the formulation of the data flow equations.

Example 2.28 To illustrate how the algorithm works let us return to Example 2.26 where we consider the program

$$[x := a + b] ; [y := a + b] ; \text{while} \text{[y > a + b]} \text{do} ([a := a + 1] ; [x := a + b])$$

Writing $W$ for the list $((2, 3), (3, 4), (4, 5), (5, 3))$ and $U$ for the set $\{a + b, a + b + 1\}$, step 1 of the algorithm will initialise the data structures as in the first row in Table 2.9. Step 2 will inspect the first element of the worklist and rows 2-7 represent cases where there is a change in the array Analysis and hence a new pair is placed on top of the worklist; it is inspected in the next iteration. Rows 8-12 represent cases where no modification is made in the array and hence the worklist is getting smaller -- the elements of $W$ are merely inspected. Step 3 will then produce the solution we already saw in Example 2.5.

Properties of the algorithm. We shall first show that the algorithm computes the expected solution to the equation system.

Lemma 2.29 The worklist algorithm in Table 2.8 always terminates and it computes the least (or MFP) solution to the instance of the framework given as input.

Proof First we prove the termination result. Step 1 and 3 are bounded loops over finite sets and thus trivially terminate. Next consider step 2. Assume that there
Table 2.9: Iteration steps of the worklist algorithm.

<table>
<thead>
<tr>
<th>$W$</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$((1,2),W)$</td>
<td>$\emptyset$</td>
<td>$U$</td>
<td>$U$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((2,3),W)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>$U$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((3,4),W)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>${a+b,a+b}$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((4,5),W)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>${a+b,a+b}$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((5,3),W)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>${a+b,a+b}$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((3,4),W)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>${a+b,a+b}$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((4,5),W)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>${a+b,a+b}$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((3,4),\ldots)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>${a+b,a+b}$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((4,5),\ldots)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>${a+b,a+b}$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((5,3),\ldots)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>${a+b,a+b}$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
<tr>
<td>$((3,4),\ldots)$</td>
<td>$\emptyset$</td>
<td>${a+b}$</td>
<td>${a+b,a+b}$</td>
<td>$U$</td>
<td>$U$</td>
</tr>
</tbody>
</table>

The inequality follows since $f_e$ is monotone and the last equation follows from $(\text{Analysis}_e, \text{Analysis}_{e'})$ being a solution to the instance.

Part (ii). On termination of the loop, the worklist is empty. We show that

$$\forall \ell', \ell : (\ell, \ell') \in F \Rightarrow \text{Analysis}[\ell'] \supseteq f_e(\text{Analysis}[\ell])$$

by contradiction. So suppose that $\text{Analysis}[\ell'] \not\supseteq f_e(\text{Analysis}[\ell])$ for some $(\ell, \ell') \in F$ and let us obtain a contradiction. Consider the last time that $\text{Analysis}[\ell]$ was updated. If this was in step 1 we considered $(\ell, \ell')$ in step 2 and ensured that

$$\text{Analysis}[\ell'] \supseteq f_e(\text{Analysis}[\ell])$$

and this invariant has been maintained ever since; hence this case cannot apply. It follows that $\text{Analysis}[\ell]$ was last updated in step 2. But at that time $(\ell, \ell')$ was placed in the worklist once again. When considering $(\ell, \ell')$ in step 2 we then ensured that

$$\text{Analysis}[\ell'] \supseteq f_e(\text{Analysis}[\ell])$$

and this invariant has been maintained ever since; hence this case cannot apply either. This completes the proof by contradiction.

On termination of the loop we have:

$$\forall \ell : \ell \in E : \text{Analysis}[\ell] \supseteq \ell$$

This follows because it was established in step 1 and it is maintained ever since. Thus it follows that at termination of step 2:

$$\forall \ell : \text{Analysis}[\ell] \supseteq \bigcup \{f_e(\text{Analysis}[\ell']) | (\ell', \ell) \in F\} \cup \ell$$

Part (iii). By our assumptions and Proposition A.10 we have

$$\forall \ell : \text{MFP}_e(\ell) \supseteq \text{Analysis}_e(\ell)$$

since $\text{Analysis}_e(\ell)$ is the least solution to the above constraint system and $\text{MFP}_e$ equals the final value of $\text{Analysis}$. Together with part (i) this proves that

$$\forall \ell : \text{MFP}_e(\ell) = \text{Analysis}_e(\ell)$$

upon termination of step 2.

Based on the proof of termination in Lemma 2.29 we can determine an upper bound on the number of basic operations (for example an application of $f_e$, an application of $\cup$, or an update of Analysis) performed by the algorithm. For this we assume that the flow $F$ is represented in such a way (for example an array of lists) that all $(\ell', \ell'')$ emanating from $\ell'$ 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 $O(b)$ times, and each time it takes only a constant number of basic steps to
process it - not counting the time needed to add new pairs to \( W \); this yields at most \( O(\epsilon \cdot h) \) basic operations for step 2. Since \( h \geq 1 \) and \( \epsilon \geq \beta \) this gives at most \( O(\epsilon \cdot h) \) basic operations for the algorithm. (Since \( \epsilon \leq \beta^2 \) a potentially coarser bound is \( O(\beta^2 \cdot h) \).)

**Example 2.30** Consider the Reaching Definitions Analysis and suppose that there are at most \( v \geq 1 \) variables and \( b \geq 1 \) labels in the program, \( S_1 \), being analysed. Since \( L = \mathcal{P}(\text{Var} \times \text{Lab}) \), it follows that \( h \leq v \cdot b \) and thus we have an \( O(v \cdot b) \) upper bound on the number of basic operations.

Actually we can do better. If \( S_1 \) is label consistent then the variable of the pairs \( (z, f) \) of \( \mathcal{P}(\text{Var} \times \text{Lab}) \) will always be uniquely determined by the label \( f \) so we get an \( O(\beta^2) \) upper bound on the number of basic operations. Furthermore, \( F \) is flow consistent and inspection of the equations for flow consistent shows that for each label \( f \) we construct at most two pairs with \( f \) in the first component. This means that \( e \leq 2 \cdot \beta \) and we get an \( O(\beta^2) \) upper bound on the number of basic operations.

### 2.4.2 The MOP Solution

Let us now consider the other solution method for Monotone Frameworks where we more directly propagate analysis information along paths in the program. Historically, this is called the **MOP solution** (for Meet Over all Paths) although we do in fact take the join (or least upper bound) over all paths leading to an elementary block; once again the reason is that the classical literature tends to focus on analyses where \( U = I \).

**Paths.** For the moment, we adopt the informal notion of a path to the entry of a block as the list of blocks traversed from the start of the program up to that block (but not including it); analogously, we can define a path from an exit of the block. Data Flow Analyses determine properties of such paths. Forward analyses concern paths from the initial block to the entry of a block; backward analyses concern paths from the exit of a block to a final block. The effect of a path on the state can be computed by composing the transfer functions associated with the individual blocks in the path. In the forward case we collect information about the state of affairs before the block is executed and in the backward case we collect information about the state of affairs immediately after the block has been executed. This informal description contrasts with the approach taken in Section 2.1 and earlier in this section; there we presented equations which were defined in terms of the immediate precursors (successors) of a block (as defined by the flow and flow\(^R\) functions). We will see later that, for a large class of analyses, these two approaches coincide.

For the formal development let us consider an instance \( (L, F, E, i, f) \) of a Monotone Framework. We shall use the notation \( \mathcal{I} = [i_1, \ldots, i_n] \) for a sequence of \( n \geq 0 \) labels. We then define two sets of paths. The paths up to but not including \( \mathcal{I} \) are

\[
\text{path}_x(\mathcal{I}) = \{[i_1, \ldots, i_{n-1}] \mid n \geq 1 \land \forall i : (i, i_{n-1}) \in F \land i_n = \mathcal{I} \land i_1 \in E\}
\]

and the paths up to and including \( \mathcal{I} \) are:

\[
\text{path}_y(\mathcal{I}) = \{[i_1, \ldots, i_n] \mid n \geq 1 \land \forall i : (i, i_{n-1}) \in F \land i_n = \mathcal{I} \land i_1 \in E\}
\]

For a path \( \mathcal{I} = [i_1, \ldots, i_n] \) we define the transfer function

\[
f_{\mathcal{I}} = f_{i_n} \circ \cdots \circ f_{i_1} \circ id
\]

so that for the empty path we have \( f_{\emptyset} = id \) where \( id \) is the identity function.

By analogy with the definition of solutions to the equation system, in particular \( MFP_{x}(\mathcal{I}) \) and \( MFP_{y}(\mathcal{I}) \), we now define two components of the MOP solution. The solution up to but not including \( \mathcal{I} \) is

\[
MOP_{x}(\mathcal{I}) = \bigcup \{f_{\mathcal{I}}(x) \mid \mathcal{I} \in \text{path}^x(\mathcal{I})\}
\]

and the solution up to and including \( \mathcal{I} \) is:

\[
MOP_{y}(\mathcal{I}) = \bigcup \{f_{\mathcal{I}}(x) \mid \mathcal{I} \in \text{path}^y(\mathcal{I})\}
\]

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 2.31** The MOP solution for Constant Propagation is undecidable.

**Proof.** Let \( u_1, \ldots, u_m \) and \( v_1, \ldots, v_n \) be strings over the alphabet \( \{1, \ldots, 9\} \) (see Appendix C). The **Modified Post Correspondence Problem** is to determine whether or not there exists a sequence \( s_1, \ldots, s_m \) with \( s_i = 1 \) such that \( u_1 \cdots u_m = v_1 \cdots v_n \).

Let \( |u| \) denote the length of the string \( u \) and let \( [u] \) be its value interpreted as a natural number. Consider the program (omitting most labels)

\[
x ::= [u]; y ::= [v];
\]

while \( [\cdots] \) do

\[
\text{if } [\cdots] \text{ then } x := x \cdot 10^{\text{length}} + [u]; \quad y := y \cdot 10^{\text{length}} + [v] \text{ else skip}
\]

\[
x ::= \text{sign}(x-y) \cdot (x-y)
\]
where sign gives the sign (which is 1 for a positive argument and 0 or -1 otherwise) and where the details of \([ \cdot ]\) are of no concern to us.

Then \(MOP_n(t)\) will map a to 1 if and only if the Modified Post Correspondence Problem has no solution. Since the Modified Post Correspondence problem is undecidable \([76]\) so is the MOP solution for Constant Propagation (assuming that our selection of arithmetic operations does indeed allow those used to be defined).

**MOP versus MFP solutions.** We shall shortly prove that the MFP solution safely approximates the MOP solution (informally, \(MFP \supseteq MOP\)). In the case of a \((\infty, \rightarrow, \tau)\) or \((\infty, \leftrightarrow, \tau)\) analysis, the MFP solution is a subset of the MOP solution (\(\subseteq\) in \(\subseteq\)); in the case of a \((\infty, \rightarrow, \tau)\) or \((\infty, \leftrightarrow, \tau)\) analysis, the MFP solution is a superset of the MOP solution. We can also show that, in the case of Distributive Frameworks, the MOP and MFP solutions coincide.

**Lemma 2.32** Consider the MFP and MOP solutions to an instance \((L, F, F, B, f, f)\) of a Monotone Framework; then:

\[
MFP_n \supseteq MOP_n \quad \text{and} \quad MFP_n \supseteq MOP_n
\]

If the framework is distributive and if \(\text{path}_n(t) \neq \emptyset\) for all \(t \in E\) and \(F\) then:

\[
MFP_n = MOP_n \quad \text{and} \quad MFP_n = MOP_n
\]

**Proof** It is straightforward to show that:

\[
\forall t : MOP_n(t) \subseteq f_t(MOF_n(t))
\]

\[
\forall t : MFP_n(t) = f_t(MOF_n(t))
\]

For the first part of the lemma it therefore suffices to prove that:

\[
\forall t : MOP_n(t) \subseteq MFP_n(t)
\]

Note that \(MFP_n\) is the least fixed point of the functional \(F\) defined by:

\[
F(A) = \bigsqcup \{ f_t(A(t')) | (t', t) \in F \} \cup \emptyset
\]

Next let us restrict the length of the paths used to compute \(MOP_n\) for \(n \geq 0\) define:

\[
MOP_n(t) = \bigsqcup \{ f_t(t') | \text{path}_n(t') \}
\]

Clearly, \(MOP_n(t) = \bigsqcup MOP_n(t)\) and to prove \(MFP_n \supseteq MOP_n\) is therefore suffices to prove

\[
\forall n : MFP_n \supseteq MOP_n
\]

and we do so by numerical induction. The basis, \(MFP_n \supseteq MOP_0\), is trivial. The inductive step proceeds as follows:

\[
MFP_n = F(MFP_n)(t)
\]

2.4 Equation Solving

\[
\begin{align*}
= & \bigsqcup \{ f_t(MOF_n(t)) | (t', t) \in F \} \cup \emptyset \\
\supseteq & \bigsqcup \{ f_t(MOF_n(t)) | (t', t) \in F \} \cup \emptyset \\
\supseteq & \bigsqcup \{ f_t(f_t(t')) \} \cup \bigsqcup \{ f_t(f_t(t')) | \text{path}_n(t') \} | (t', t) \in F \} \cup \emptyset \\
\supseteq & \bigsqcup \{ f_t(f_t(t')) \} \cup \bigsqcup \{ f_t(f_t(t')) \} \cup \bigsqcup \{ f_t(f_t(t')) | \text{path}_n(t') \} | (t', t) \in F \} \cup \emptyset \\
= & \bigsqcup \{ f_t(t') \} \cup \bigsqcup \{ f_t(t') \} \cup \bigsqcup \{ f_t(f_t(t')) | \text{path}_n(t') \} | (t', t) \in F \} \cup \emptyset \\
= & \text{MOP}_n(t)
\end{align*}
\]

where we have used the induction hypothesis to get the first inequality. This completes the proof of \(MOP_n \supseteq MOP_n\) and \(MFP_n \supseteq MOP_n\).

To prove the second part of the lemma we now assume that the framework is distributive. Consider \(t \in E\) or \(m\). By assumption \(f_t\) is distributive, that is \(f_t(t_1 \cup t_2) = f_t(t_1) \cup f_t(t_2)\), and from Lemma A.9 of Appendix A it follows that

\[
f_t(Y) = \bigsqcup \{ f_t(t') | t' \in Y \}
\]

whenever \(t'\) is non-empty. By assumption we also have \(\text{path}_n(t) \neq \emptyset\) and it follows that

\[
\begin{align*}
f_t & \bigsqcup \{ f_t(t') | \text{path}_n(t) \} = \bigsqcup \{ f_t(t') | \text{path}_n(t) \} \\
& \bigsqcup \{ f_t(t') | \text{path}_n(t) \}
\end{align*}
\]

and this shows that:

\[
\forall t : f_t(MOP_n(t)) = MOP_n(t)
\]

Next we calculate:

\[
\begin{align*}
\text{MOP}_n(t) & = \bigsqcup \{ f_t(t') | \text{path}_n(t') \} \\
& \bigsqcup \{ f_t(t') \} \cup \bigsqcup \{ f_t(t') | \text{path}_n(t') \} \cup \{ (t', t) \in F \} \\
& \bigsqcup \{ f_t(f_t(t')) | \text{path}_n(t') \} \cup \bigsqcup \{ f_t(f_t(t')) \} \cup \bigsqcup \{ f_t(f_t(t')) | \text{path}_n(t') \} \cup \{ (t', t) \in F \} \\
& \bigsqcup \{ f_t(f_t(t')) | \text{path}_n(t') \} \cup \bigsqcup \{ f_t(f_t(t')) \} \cup \bigsqcup \{ f_t(f_t(t')) | \text{path}_n(t') \} \cup \{ (t', t) \in F \} \cup \emptyset
\end{align*}
\]

Together this shows that \((MOP_n, MOP_n)\) is a solution to the data flow equations. Using Proposition A.10 of Appendix A and the fact that \((MFP_n, MFP_n)\) is the least solution we get \(MOP_n \supseteq MFP_n\) and \(MOP_n \supseteq MFP_n\). Together with the results of the first part of the lemma we get \(MOP_n = MFP_n\) and \(MOP_n = MFP_n\).

We shall leave it to Exercise 2.13 to show that the condition that \(\text{path}_n(t) \neq \emptyset\) (for \(t \in E\) and \(F\)) does hold when the Monotone Framework is constructed from a program \(S_n\) in the manner of the earlier sections.

It is sometimes stated that the MOP solution is the desired solution and that one only uses the MFP solution because the MOP solution might not be
computable. In order to validate this belief we would need to prove that the 
MOP solution is semantically correct as was proved for the MFP solution in 
Section 2.2 in the case of Live Variables Analysis - in the case of Live Variables 
this is of course immediate since it is a Distributive Framework. We shall not 
do so because it is always possible to formulate the MOP solution as an MFP 
solution over a different property space (like \( P(L) \)) and therefore little is lost 
by focusing on the fixed point approach to Monotone Frameworks. (Also note that \( P(L) \) satisfies the Ascending Chain Condition when \( L \) is finite.)

### 2.5 Interprocedural Analysis

The Data Flow Analysis techniques that have been presented in the previous 
sections are called interprocedural analyses because they deal with simple 
languages without functions or procedures. It is somewhat more demanding 
to perform interprocedural analyses where functions and procedures are 
taken into account. Complications arise when ensuring that calls and returns 
match one another, when dealing with parameter mechanisms (and the aliasing 
that may result from call-by-reference) and when allowing procedures as 
parameters.

In this section we shall introduce some of the key techniques of interprocedural 
analysis. To keep things simple we just extend the WHILE language with top-level 
declarations of global mutually recursive procedures having a 
call-by-value parameter and a call-by-result parameter. The extension of the 
techniques to a language where procedures may have multiple call-by-value, 
call-by-result and call-by-value-result parameters is straightforward and so is 
the extension with local variable declarations (see Exercise 2.20); we shall 
freely use these extensions in examples.

**Syntax of the procedure language.** A program, \( P \), in the extended 
WHILE-language has the form

\[
\begin{align*}
\text{begin } D, S, \text{ end}
\end{align*}
\]

where \( D \) is a sequence of procedure declarations:

\[
D ::= \text{proc } p(\text{val } x, \text{res } y) \text{ is }^* S \text{ end }^* \mid D D
\]

Procedure names (denoted \( p \)) are syntactically distinct from variables (denoted \( x \) and \( y \)). The label \( \ell_c \) of \( S \) marks the entry to the procedure body and 
the label \( \ell_e \) of end marks the exit from the procedure body. The syntax 
of statements is extended with:

\[
S ::= \ldots \mid \text{call } p(\text{val } x, \text{res } y) \text{ is } \ell_c S \text{ end }^* \ell_e
\]

The call statement has two labels: \( \ell_c \) will be used for the call of the procedure 
and \( \ell_e \) will be used for the associated return; the actual parameters are \( x \) 
and \( y \).

The language is statically scoped, the parameter mechanism is call-by-value 
for the first parameter and call-by-result for the second parameter and the 
procedures may be mutually recursive. We shall assume throughout that the 
program is uniquely labelled (and hence label consistent); also we shall 
assume that only procedures that have been declared in \( D \) are ever called 
and that \( D \) does not contain two definitions of the same procedure name.

**Example 2.33** Consider the following program calculating the Fibonacci 
number of the positive integer stored in \( x \) and returning it in \( y \):

\[
\begin{align*}
\text{begin } \quad \text{proc } \text{fib}(\text{val } x, \text{res } y) \text{ is }^* \quad \\
\text{if } (x < 3) \text{ then } y := x + 1 \text{ end }^* \quad \\
\text{else } (\text{call } \text{fib}(x-1, y), \text{call } \text{fib}(x-2, y)) \text{ end }^* \quad \\
\text{call } \text{fib}(x, 0, y) \text{ end}
\end{align*}
\]

It uses the procedure \( \text{fib} \) that returns in \( y \) the Fibonacci number of \( x \) 
plus the value of \( u \). Both \( x \) and \( y \) are global variables whereas \( x, u \) 
and \( y \) are formal parameters and hence local variables.

**Flow graphs for statements.** The next step is to extend the definitions of 
the functions \( \text{init} \), \( \text{final} \), \( \text{blocks} \), \( \text{labels} \), and \( \text{flow} \) to specify the flow 
graphs also for the procedure language. For the new statement we take:

\[
\begin{align*}
\text{init}(\text{call } p(\text{val } x, \text{res } y)) & = \ell_c \\
\text{final}(\text{call } p(\text{val } x, \text{res } y)) & = (\ell_e) \\
\text{blocks}(\text{call } p(\text{val } x, \text{res } y)) & = \{\text{call } p(\text{val } x, \text{res } y)\} \\
\text{labels}(\text{call } p(\text{val } x, \text{res } y)) & = (\ell_c, \ell_e) \\
\text{flow}(\text{call } p(\text{val } x, \text{res } y)) & = ((\ell_c; \ell_a), (\ell_e; \ell_e)) \quad \text{if proc } \text{call } p(\text{val } x, \text{res } y) \text{ is }^* S \text{ end }^* \quad \\
& \quad \text{is in } D
\end{align*}
\]

Here \( (\ell_c; \ell_a) \) and \( (\ell_e; \ell_e) \) are new kinds of flows:

- \( (\ell_c; \ell_a) \) is the flow corresponding to calling a procedure at \( \ell_c \) and with 
  \( \ell_a \) being the entry point for the procedure body, and
- \( (\ell_e; \ell_e) \) is the flow corresponding to exiting a procedure body at \( \ell_e \) 
  and returning to the call at \( \ell_c \).

The definition of \( \text{flow}(\text{call } p(\text{val } x, \text{res } y)) \) exploits the fact that the syntax of 
procedure calls only allows us to use the (constant) name of a procedure
defined in the program; had we been allowed to use a variable that denotes a procedure (e.g., because it was a formal parameter to some procedure or because it was a variable being assigned some procedure) then it would be much harder to define \( \text{flow}([\text{call } p(a,z)]_{S}^{\ell}) \). This is often called the dynamic dispatch problem and we shall deal with it in Chapter 3.

**Flow graphs for programs.** Next consider the program \( P \), of the form begin \( D_{S} \), \( S \), end. For each procedure declaration \( \text{proc } p(\text{val } x, \text{res } y) \) \( S \) and \( \ell \) we set

\[
\begin{align*}
\text{init}(p) &= \{ \ell_{a} \} \\
\text{final}(p) &= \{ \ell_{s} \} \\
\text{blocks}(p) &= \{ \text{is }^{\ell_{a}}, \text{end }^{\ell_{a}} \} \cup \text{blocks}(S) \\
\text{labels}(p) &= \{ \ell_{a}, \ell_{s} \} \cup \text{labels}(S) \\
\text{flow}(p) &= \{ (\ell_{a}, \text{init}(S)) \} \cup \text{flow}(S) \cup \{ (\ell, \ell) \mid \ell \in \text{final}(S) \}
\end{align*}
\]

and for the entire program \( P \), we set

\[
\begin{align*}
\text{init}_{*} &= \text{init}(S) \\
\text{final}_{*} &= \text{final}(S) \\
\text{blocks}_{*} &= \bigcup \{ \text{blocks}(p) \mid \text{proc } p(\text{val } x, \text{res } y) \text{ is }^{\ell_{a}} \text{ end }^{\ell_{a}} \text{ is in } D_{S} \} \\
\text{labels}_{*} &= \bigcup \{ \text{labels}(p) \mid \text{proc } p(\text{val } x, \text{res } y) \text{ is }^{\ell_{a}} \text{ end }^{\ell_{a}} \text{ is in } D_{S} \} \\
\text{flow}_{*} &= \bigcup \{ \text{flow}(p) \mid \text{proc } p(\text{val } x, \text{res } y) \text{ is }^{\ell_{a}} \text{ end }^{\ell_{a}} \text{ is in } D_{S} \} \\
&\cup \text{flow}(S)
\end{align*}
\]

as well as \( \text{Lab}_{*} = \text{labels}_{*} \).

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

\[
\text{inter-flow}_{*} = \{ (\ell_{a}, \ell_{a}, \ell_{a}, \ell_{s}) \mid P \text{ contains } [\text{call } p(a,z)]^{\ell_{a}}_{S}, \text{ as well as } \text{proc } p(\text{val } x, \text{res } y) \text{ is }^{\ell_{a}} \text{ end }^{\ell_{a}} \}
\]

that clearly indicates the relationship between the labels of a procedure call and the corresponding procedure body. This information will be used later to analyse procedure calls and returns more precisely than is otherwise possible. Indeed, suppose that \( \text{inter-flow}_{*} \) contains \( (\ell_{a}, \ell_{a}, \ell_{a}, \ell_{s}) \) for \( i = 1, 2 \) in which case \( \text{flow}_{*} \) contains \( (\ell_{1}; \ell_{a}) \) and \( (\ell_{a}; \ell_{s}) \) for \( i = 1, 2 \). But this “gives rise to” the four tuples \( (\ell_{a}, \ell_{a}, \ell_{a}, \ell_{a}) \) for \( i = 1, 2 \) and \( j = 1, 2 \) and only the tuples with \( i = j \) match the return with the call; these tuples are exactly the ones in \( \text{inter-flow}_{*} \).

**Example 2.34** For the Fibonacci program considered in Example 2.33 we have

\[
\begin{align*}
\text{flow}_{*} &= \{ (1, 2), (2, 3), (3, 8), \\
&\quad (2, 4), (4, 1), (6, 5), (6, 6), (6, 7), (7, 8), \\
&\quad (9, 1), (9, 10) \}
\end{align*}
\]

\[
\text{inter-flow}_{*} = \{ (9, 1, 8, 10), (4, 1, 8, 5), (6, 1, 8, 7) \}
\]

and \( \text{init}_{*} = 9 \) and \( \text{final}_{*} = 10 \). The corresponding flow graph is illustrated in Figure 2.7.

For a forward analysis we use \( F = \text{flow}_{*} \) and \( E = \{ \text{init}_{*} \} \) much as before and we introduce a new “metavariable” \( \text{IF} = \text{inter-flow}_{*} \) for the interprocedural flow; for a backward analysis we use \( F = \text{flow}_{*}^{\Phi} \), \( E = \text{final}^{\Phi} \), and \( \text{IF} = \text{inter-flow}_{*}^{\Phi} \). Most of the explanations in the sequel will focus on forward analyses.

**2.5.1 Structural Operational Semantics**

We shall now show how the semantics of WHILE can be extended to cope with the new constructs. To ensure that the language allows **local data** in
procedures we shall need to distinguish between the values assigned to different incarnations of the same variable and for this we introduce an infinite set of locations (or addresses):

$$\xi \in \text{Loc}$$

locations

An environment, $\rho$, will map the variables in the current scope to their locations, and a store, $\varsigma$, will then specify the values of these locations:

$$\rho \in \text{Env} = \text{Var}_\omega \rightarrow \text{Loc}$$

environments

$$\varsigma \in \text{Store} = \text{Loc} \rightarrow \mathbb{Z}$$

stores

Here $\text{Var}_\omega$ is the (finite) set of variables occurring in the program and $\text{Loc} \rightarrow \mathbb{Z}$ denotes the set of partial functions from Loc to $\mathbb{Z}$ that have a finite domain. Thus the previously used states $\sigma \in \text{State} = \text{Var}_\omega \rightarrow \mathbb{Z}$ have been replaced by the two mappings $\rho$ and $\varsigma$ and can be reconstructed as $\sigma = \varsigma \circ \rho$: to determine the value of a variable $x$ we first determine its location $\xi = \rho(x)$ and next the value $\varsigma(\xi)$ stored in that location. For this to work it is essential that $\varsigma \circ \rho : \text{Var}_\omega \rightarrow \mathbb{Z}$ is a total function rather than a partial function; in other words, we demand that $\text{ran}(\rho) \subseteq \text{dom}(\varsigma)$ where $\text{ran}(\rho) = \{ \rho(x) \mid x \in \text{Var}_\omega \}$ and $\text{dom}(\varsigma) = \{ \xi \mid \varsigma(\xi) \text{ is defined} \}$.

The locations of the global variables of the program $P$, are given by a top-level environment denoted $\rho_\omega$; we shall assume that it maps all variables to unique locations. The semantics of statements is now given relative to modifications of this environment. The transitions have the general form

$$\rho \vdash_\omega (S, \varsigma) \rightarrow (S', \varsigma')$$

in case that the computation does not terminate in one step, and the form

$$\rho \vdash_\omega (S, \varsigma) \rightarrow \varsigma'$$

in case that it does terminate in one step. It is fairly straightforward to rewrite the semantics of WHILE given in Table 2.6 to have this form; as an example the clause [ass] for assignments becomes:

$$\rho \vdash_\omega (x := a, \varsigma) \rightarrow \varsigma' \rho(x) \rightarrow A[a] (\varsigma \circ \rho)$$

if $\varsigma \circ \rho$ is total

Note that there is no need to modify the semantics of arithmetic and boolean expressions.

For procedure calls we make use of the top-level environment, $\rho_\omega$, and we take:

$$\rho_\omega \vdash_\omega \{ \text{call } p(a, z) \}^2_{\xi_1, \xi_2} \rightarrow (\text{bind } \rho_\omega [x \mapsto \xi_1, y \mapsto \xi_2] \text{ in } S \text{ then } z := y_1, \varsigma[\xi_1 \mapsto A[a] (\varsigma \circ \rho), \xi_2 \mapsto v])$$

where $\xi_1, \xi_2 \notin \text{dom}(\rho), v \in \mathbb{Z}$

and $\text{proc } p \text{ (val } z, \text{ res } y) \text{ is } \varsigma' \text{ in } S$ and $\varsigma'$ is in $\mathcal{D}$.

The idea is that we allocate new locations $\xi_1$ and $\xi_2$ for the formal parameters $z$ and $y$, and we then make use of a bind-construct to combine the procedure body $S$ with the environment $\rho_\omega [x \mapsto \xi_1, y \mapsto \xi_2]$ in which it must be executed and we also record that the final value of $y$ must be returned in the actual parameter $z$. At the same time the store is updated such that the new location for $z$ is mapped to the value of the actual parameter $a$ whereas we do not control the initial value, $v$, of the new location for $y$. The bind-construct is only needed to ensure that we have static scope rules and its semantics is as follows:

$$\rho' \vdash_\omega (S, \varsigma) \rightarrow (S', \varsigma')$$

$$\rho' \vdash_\omega (\text{bind } \rho' \text{ in } S \text{ then } z := y, \varsigma' \rightarrow (\text{bind } \rho' \text{ in } S' \text{ then } z := y, \varsigma')$$

$$\rho' \vdash_\omega (S, \varsigma) \rightarrow \varsigma'$$

$$\rho' \vdash_\omega (\text{bind } \rho' \text{ in } S \text{ then } z := y, \varsigma') \rightarrow \varsigma'[\rho(z) \rightarrow \varsigma'(\rho'(y))]

The first rule expresses that executing one step of the body of the construct amounts to executing one step of the construct itself; note that we use the local environment when executing the body. The second rule expresses that when the execution of the body finishes then so does execution of the construct itself and we update the value of the global variable $z$ to be that of the local variable $y$; furthermore, there is no need for the local environment $\rho'$ to be retained as subsequent computations will use the previous environment $\rho$.

Remark. Although the semantics works with two mappings, an environment and a store, it is often the case that the analysis abstracts the state, i.e. the composition of the environment and the store. The correctness of the analysis will then have to relate the abstract state both to the environment and the store.

The correctness result will often be expressed in the style of Section 2.2: information obtained by analysing the original program will remain correct under execution of the program. The semantics presented above deviates from that of the WHILE-language in that it introduces the bind-construct which is only used in the intermediate configurations. So in order to prove the correctness result we will also need to specify how to analyse the bind-construct. We refer to Chapter 3 for an illustration of how to do this.
2.5.2 Intraprocedural versus Interprocedural Analysis

To appreciate why interprocedural analysis is harder than intraprocedural analysis we begin by just naively using the techniques from the previous sections. For this we suppose that:

- for each procedure call \(\text{call}(a, z)^s\), we have two transfer functions \(f_t\) and \(f_s\), corresponding to calling the procedure and returning from the call, and

- for each procedure definition \(\text{proc} p\{a(x,y)\}^s\), we have two transfer functions \(f_t\) and \(f_s\), corresponding to entering and exiting the procedure body.

A naive formulation. Given an instance \((I, \mathcal{F}, E, t, f)\) of a Monotone Framework we shall now treat the two kinds of flow \((f_t, f_s)\) (versus \((f_t, f_s)\) and \((f_t, f_s)\)) in the same way: we interpret the semi-colon as standing for a comma. While a Monotone Framework is allowed to interpret all transfer functions freely, we shall for now naively assume that the two transfer functions associated with procedure definitions are the identity functions, and that the two transfer functions associated with each procedure call are also the identity functions, thus effectively ignoring the parameter-passing.

We now obtain an equation system of the form considered in the previous sections:

\[
A^x(t) = f_t(A^x(t)) \\
A^x(t) = \bigcup \{A^x(t') \mid (t', t) \in \mathcal{F} \text{ or } (t', t) \in \mathcal{F} \} \cup f_t^x
\]

Here \(i^x\) is as in Section 2.3:

\[
i^x = \begin{cases} 
1 & \text{if } t \in E \\
0 & \text{if } t \notin E 
\end{cases}
\]

When inspecting this equation system it should be apparent that both procedure calls \((t_s, t_e)^s\) and procedure returns \((t_s, t_e)^s\) are treated like goto's: there is no mechanism for ensuring that information flowing along \((t_s, t_e)^s\) from a call to a procedure only flows back along \((t_s, t_e)^s\) from the procedure to the same call. (Indeed, nowhere does the formulation consult the interprocedural flow, \(\mathcal{F}\)). Expressed in terms of the flow graph in Figure 2.7, there is nothing preventing us from considering a path like \([0, 1, 2, 3, 8, 10]\) that does not correspond to a run of the program. Intuitively, the equation system considers a much too large set of “paths” through the program and hence will be grossly imprecise (although formally on the safe side).

Valid paths. A natural way to overcome this shortcoming is to somehow restrict the attention to paths that have the proper nesting of procedure calls and exits. We shall explore this idea in the context of redefining the MOP solution of Section 2.4 to only take the proper set of paths into account, thereby defining an MOP solution (or Meet over all Valid Paths).

So consider a program \(P_s\), of the form begin \(D_s, S, \ldots\). A path is said to be a complete path from \(t_1\) to \(t_2\) in \(P_s\) if it has proper nesting of procedure entries and exits and such that a procedure returns to the point where it was called. These paths are generated by the nonterminal \(CP_{t_1,t_2}\) according to the following productions:

\[
CP_{t_1,t_2} \rightarrow t_1 \quad \text{whenever } t_1 = t_2
\]

\[
CP_{t_1,t_2} \rightarrow t_1, CP_{t_2,t_3} \quad \text{whenever } (t_1, t_2) \in \mathcal{F};
\]

\[
\text{for a forward analysis this means that } (t_1, t_2) \in \mathcal{F}.
\]

\[
CP_{t_1,t_2} \rightarrow t_1, CP_{t_2,t_3}, CP_{t_3,t} \quad \text{whenever } (t_1, t_2, t_3, t_4) \in \mathcal{F};
\]

\[
\text{for a forward analysis this means that } P_s \text{ contains } \text{call}(p(a, z))^s\text{ and proc } p\{a(x,y)\}^s \text{ and } \text{proc } p\{a(x,y)\}^s \text{ and } \mathcal{F}.
\]

The matching of calls and returns is ensured by the last kind of productions: the flows \((t_s, t_e)^s\) and \((t_s, t_e)^s\) are forced to obey a parenthesis structure in that \(t_s, t_e\) will be in the generated path only if there is a matching occurrence of \(t_s, t_e\) and vice versa. Hence for a forward analysis, a terminating computation will give rise to a complete path from init to one of the labels of final. Note that the grammar constructed above will only have a finite set of nonterminals because there is only a finite set of labels in \(P_s\).

Example 2.35 For the Fibonacci program of Example 2.33 we obtain the following grammar (using forward flow and ignoring the parts not reachable from \(CP_{9,10}\)):

\[
CP_{9,10} \rightarrow 3, CP_{9,8}, CP_{10,10} \quad CP_{3,8} \rightarrow 3, CP_{3,8}, CP_{8,8}
\]

\[
CP_{10,10} \rightarrow 10 \quad CP_{8,8} \rightarrow 8
\]

\[
CP_{9,8} \rightarrow 9, CP_{9,8}, CP_{10,10} \quad CP_{4,8} \rightarrow 4, CP_{4,8}, CP_{5,8}
\]

\[
CP_{2,8} \rightarrow 2, CP_{2,8} \quad CP_{5,8} \rightarrow 5, CP_{5,8}
\]

\[
CP_{3,8} \rightarrow 2, CP_{3,8} \quad CP_{6,8} \rightarrow 6, CP_{6,8}, CP_{7,8}
\]

\[
CP_{8,8} \rightarrow 7, CP_{8,8}
\]

It is now easy to verify that the path \([9, 1, 2, 4, 1, 2, 3, 8, 5, 6, 1, 2, 3, 8, 7, 8, 10]\) is generated by \(CP_{9,10}\) whereas the path \([9, 1, 2, 4, 1, 2, 3, 8, 10]\) is not.

A path is said to be a valid path if it starts at an extremal node of \(P_s\) and if all the procedure exits match the procedure entries but it is possible that
and how to avoid taking too many invalid paths. An obvious approach is to encode information about the paths taken into the data flow properties themselves; to this end we introduce context information:

$$\delta \in \Delta$$

context information

The context may simply be an encoding of the path taken but we shall see in Subsection 2.5.5 that there are other possibilities. We shall now show how an instance of a Monotone Framework (as introduced in Section 2.3) can be extended to take context into account.

The intraprocedural fragment. Consider an instance \((L, \mathcal{F}, F, E, i, f)\) of a Monotone Framework. We shall now construct an instance

\[
(L, \mathcal{F}, F, E, \mathcal{C}, f, \bar{f})
\]

of an embellished Monotone Framework that takes context into account. We begin with the parts of its definition that are independent of the actual choice of \(\Delta\), i.e. the parts that correspond to the intraprocedural analysis:

- \(\mathcal{C}\) the transfer functions in \(\mathcal{F}\) are monotone; and
- each transfer function \(\bar{f}_c\) is given by \(\bar{f}_c(\cdot)(\delta) = f_c(\cdot)(\delta)\).

In other words, the new instance applies the transfer functions of the original instance in a pointwise fashion.

Ignoring procedures, the data flow equations will take the form displayed earlier:

\[
A_+(\ell) = \mathcal{F}_c(A_+(\ell))
\]

for all labels that do not label a procedure call (i.e. that do not occur as first or fourth components of a tuple in \(IF\))

\[
A_+(\ell) = \bigcup \{ A_+(\ell') | (\ell', \ell) \in F \text{ or } (\ell; \ell) \in F \} \cup \bar{f}_c
\]

for all labels (including those that label procedure calls)

Example 2.36 Let \((L_{\text{sign}}, \mathcal{F}_{\text{sign}}, F, E_{\text{sign}}, i_{\text{sign}}, f_{\text{sign}})\) be an instance of a Monotone Framework specifying a Detection of Signs Analysis (see Exercise 2.15) and assume that

\[
L_{\text{sign}} = \mathcal{P}(\text{Var} \rightarrow \text{Sign})
\]
where \( \text{Sign} = \{-, 0, +\} \). Thus \( L_{\text{sign}} \) describes sets of abstract states \( \sigma_{\text{sign}} \) mapping variables to their possible signs. The transfer function \( f_t^\sigma \) associated with the assignment \([x := a]^t\) will now be written as

\[
f_t^\sigma(Y) = \bigcup \{ \phi_t^\sigma(\sigma_{\text{sign}}) \mid \sigma_{\text{sign}} \in Y \}
\]

where \( Y \subseteq \text{Var} \to \text{Sign} \) and

\[
\phi_t^\sigma(\sigma_{\text{sign}}) = \{ \sigma_{\text{sign}}[x \mapsto a] \mid a \in A_{\text{sign}}[x][a](\sigma_{\text{sign}}) \}
\]

Here \( A_{\text{sign}}: \text{AExp} \to (\text{Var} \to \text{Sign}) \to \mathcal{P}(\text{Sign}) \) specifies the analysis of arithmetic expressions. The transfer functions for tests and \textit{skip}-statements are the identity functions.

Given a set \( \Delta \) of contexts, the embellished Monotone Framework will have

\[
\overline{L_{\text{sign}}} = \Delta \to L_{\text{sign}}
\]

but we shall prefer the following isomorphic definition

\[
\overline{L_{\text{sign}}} = \mathcal{P}(\Delta \times (\text{Var} \to \text{Sign}))
\]

Thus \( \overline{L_{\text{sign}}} \) describes sets of pairs of context and abstract states. The transfer function associated with the assignment \([x := a]^t\) will now be:

\[
f_t^\sigma(Z) = \bigcup \{ (\delta, x) \times \phi_t^\sigma(\sigma_{\text{sign}}) \mid (\delta, \sigma_{\text{sign}}) \in Z \}
\]

In subsequent examples we shall further develop this analysis.

The interprocedural fragment. It remains to formulate the data flow equations corresponding to procedures.

For a procedure definition \( \text{proc } p(\text{val } x, \text{res } y) \{ \text{val } x, \text{res } y \} \) \( S \) \( \text{end } ^t \), we have two transfer functions:

\[
\overline{f_t}(t), \overline{f_t}(t) : (\Delta \to L) \to (\Delta \to L)
\]

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

\[
\overline{f_t}(t) = t, \overline{f_t}(t) = \overline{f_t}(t)
\]

for all \( t \in \overline{L} \). Hence the effect of procedure entry is handled by the transfer function for procedure call (considered below) and similarly the effect of procedure exit is handled by the transfer function for procedure return (also considered below). For more advanced languages where many semantic actions take place at procedure entry or exit it may be preferable to reconsider this decision.

For a procedure call \( (t_1, t_2, t_3, t_4) \in \text{IF} \) we shall define two transfer functions. In our explanation we shall concentrate on the case of forward analyses where \( \overline{P}_k \) contains \( \{\text{call } p(a, x)\}^t_k \) as well as \( \text{proc } p(\text{val } x, \text{res } y) \) \( S \) \( \text{end } ^t \).

Corresponding to the actual call we have the transfer function

\[
\overline{f_t}(t) : (\Delta \to L) \to (\Delta \to L)
\]

and it is used in the equation:

\[
A_*(t) = \overline{f_t}(A_*(t)) \text{ for all } (t_1, t_2, t_3, t_4) \in \text{IF}
\]

In other words, the transfer function modifies the data flow properties (and the context) as required for passing to the procedure entry.

Corresponding to the return we have the transfer function

\[
\overline{f_t}(t) : (\Delta \to L) \times (\Delta \to L) \to (\Delta \to L)
\]

and it is used in the equation:

\[
A_*(t) = \overline{f_t}(t, A_*(t), A_*(t)) \text{ for all } (t_1, t_2, t_3, t_4) \in \text{IF}
\]

The first parameter of \( \overline{f_t}(t) \) describes the data flow properties at the call point for the procedure and the second parameter describes the properties at the exit from the procedure body. Ignoring the first parameter, the transfer function modifies the data flow properties (and the context) as required for passing back from the procedure exit. The purpose of the first parameter is
to recover some of the information (data flow properties as well as context information) that was available before the actual call; how this is done depends on the actual choice of the set, $\Delta$, of context information and we shall return to this shortly. Figure 2.8 illustrates the flow of data in the analysis of the procedure call.

Variations. The functionality and use of $f_{t,c}$ (as well as Figure 2.8) is sufficiently general that it allows us to deal with most of the scenarios found in the literature. A simple example being the possibility to define

$$ f_{t,c}^t (\bar{t}, \bar{v}) = \overline{f_{t,c} (\bar{v})} $$

thereby completely ignoring the information before the call; this is illustrated in Figure 2.9.

A somewhat more interesting example is the ability to define

$$ f_{t,c}^t (\bar{t}, \bar{v}) = f_{t,c}^t (\bar{t}) \cup f_{t,c}^\Delta (\bar{v}) $$

thereby allowing a simple combination of the information coming back from the call with the information pertaining before the call. This form is illustrated in Figure 2.10 and is often motivated on the grounds that $f_{t,c}^\Delta$ copies data that is local to the calling procedure whereas $f_{t,c}^\Delta$ copies information that is global. (It may be worth noticing that the function $f_{t,c}^\Delta$ is completely additive if and only if it can be written in this form with $f_{t,c}^t$ and $f_{t,c}^\Delta$ being completely additive.)

Context-sensitive versus context-insensitive. So far we have criticised the naive approach because it was unable to maintain the proper

relationship between procedure calls and procedure returns. A related criticism of the naive approach is that it cannot distinguish between the different calls of a procedure. The information about calling states is combined for all call sites, the procedure body is analysed only once using this combined information, and the resulting information about the set of return states is used at all return points. The phrase context-insensitive is often used to refer to this shortcoming.

The use of non-trivial context information not only helps to avoid the first criticism but also the second: if there are two different calls but they are reached with different contexts, $\delta_1$ and $\delta_2$, then all information obtained from the procedure will be clearly related to $\delta_1$ or $\delta_2$ and no undesired combination or "cross-over" will take place. The phrase context-sensitive is often used to refer to this ability.

Clearly a context-sensitive analysis is more precise than a context-insensitive analysis but at the same time it is also likely to be more costly. The choice between which technique to use amounts to a careful balance between precision and efficiency.

2.5.4 Call Strings as Context
To complete the design of the analysis of the program we must choose the set, $\Delta$, of context information and also specify the extremal value, $\bar{t}$, and define the two transfer functions associated with procedure calls. In this subsection we shall consider two approaches based on call strings and our explanation will be in terms of forward analyses.

Call strings of unbounded length. As the first possibility we simply encode the path taken; however, since our main interest is with pro-
procedure calls we shall only record flows of the form \((\ell_c; \ell_e)\) corresponding to a procedure call. Formally we take

\[\Delta = \text{Lab}^*\]

where the most recent label \(\ell_e\) of a procedure call is at the right end (just as was the case for valid paths and paths); elements of \(\Delta\) are called call strings. We then define the extremal value \(\bar{\ell}\) by the formula

\[\bar{\ell}(\delta) = \begin{cases} 
\ell & \text{if } \delta = \Lambda \\
\bot & \text{otherwise}
\end{cases}\]

where \(\Lambda\) is the empty sequence corresponding to the fact that there are no pending procedure calls when the program starts execution; \(\ell\) is the extremal value available from the underlying Monotone Framework.

**Example 2.37** For the Fibonacci program of Example 2.33 the following call strings will be of interest:

\[\Lambda, [9], [9, 4], [9, 4, 6], [9, 4, 6, 4], [9, 4, 6, 4, 6], \ldots\]

corresponding to the cases with 0, 1, 2, 3, \ldots pending procedure calls.

For a procedure call \((\ell_c, \ell_n, \ell_e, \ell_r) \in \text{IF}\), amounting to \([\text{call } p(a, x)](\delta)\) in the case of a forward analysis, we define the transfer function \(\bar{f}_{\ell_c}\) such that \(\bar{f}_{\ell_c}(\bar{\ell}(\delta)) = f_{\ell_c}(\bar{\ell}(\delta))\) where \([\delta, \ell_c]\) denotes the path obtained by appending \(\ell_c\) to \(\delta\) (so as to reflect that now we enter the body of the procedure) and the function \(f_{\ell_c} : \text{L} \rightarrow \text{L}\) describes how the property is modified. This is achieved by setting

\[\bar{f}_{\ell_c}(\bar{\ell}(\delta)) = \begin{cases} 
\bar{f}_{\ell_c}(\bar{\ell}(\delta)) & \text{when } \delta = [\delta, \ell_c] \\
\bot & \text{otherwise}
\end{cases}\]

which takes care of the special case of empty paths.

Next we define the transfer function \(\bar{f}_{\ell, \ell_e}\) corresponding to returning from the procedure call:

\[\bar{f}_{\ell, \ell_e}(\bar{\ell}, \bar{\ell'}) = \bar{f}_{\ell, \ell_e}(\bar{\ell}(\delta), \bar{\ell}(\delta'))\]

Here the information \(\bar{\ell}\) from the original call is combined with information \(\bar{\ell}'\) from the procedure exit using the function \(\bar{f}_{\ell, \ell_e} : \text{L} \times \text{L} \rightarrow \text{L}\). However, only information corresponding to the same contexts for call point \(\ell_c\) is combined: this is ensured by the two occurrences of \(\delta\) in the right hand side of the above formula.

**2.5 Interprocedural Analysis**

**Example 2.38** Let us return to the Detection of Signs Analysis of Example 2.36. For a procedure call \([\text{call } p(a, x)](\delta)\) where \(p\) is declared by

\[\text{proc } p\text{(val } x, \text{res } y)\text{ is}\ (\delta)\ S\ \text{end} \delta^e\]

we may have:

\[f_{\ell_c}^{\text{sig}}(\delta) = \{\delta' \times \phi_{\ell_c}^{\text{sig}}(\sigma_{\ell_c}^{\text{sig}}) \mid (\delta', \sigma_{\ell_c}^{\text{sig}}) \in Z \land \delta' = [\delta, \ell_c]\}
\]

\[\phi_{\ell_c}^{\text{sig}}(\sigma_{\ell_c}^{\text{sig}}) = \{\sigma'_{\ell_c} | x \mapsto s | y \mapsto s' \mid s \in A_{\text{expr}}(\sigma_{\ell_c}^{\text{sig}}) \land s' \in \{-, 0, +\}\}
\]

When returning from the procedure call we take:

\[f_{\ell, \ell_e}^{\text{sig}^2}(\delta, \delta') = \{\delta \times \phi_{\ell, \ell_e}^{\text{sig}^2}(\sigma_1^{\text{sig}}, \sigma_2^{\text{sig}}) \mid (\delta, \sigma_1^{\text{sig}}, \sigma_2^{\text{sig}}) \in Z' \land \delta' = [\delta, \ell_c]\}
\]

\[\phi_{\ell, \ell_e}^{\text{sig}^2}(\sigma_1^{\text{sig}}, \sigma_2^{\text{sig}}) = \{\sigma_1^{\text{sig}} | x \mapsto \sigma_1^{\text{sig}}(x) ; y \mapsto \sigma_2^{\text{sig}}(y) ; z \mapsto \sigma_2^{\text{sig}}(y)\}
\]

Thus we extract all the information from the procedure body except for the information about the formal parameters \(x\) and \(y\) and the actual parameter \(z\). For the formal parameters we rely on the information available before the current call which is still correct and for the actual parameter we perform the required update of the information. Note that to facilitate this definition it is crucial that the transfer function \(f_{\ell, \ell_e}\) takes two arguments: information from the call point as well as from the procedure exit.

**Call strings of bounded length.** Clearly the call strings can become arbitrarily long because the procedures may be recursive. It is therefore customary to restrict their length to be at most \(k\) for some number \(k \geq 0\); the idea being that only the last \(k\) calls are recorded. We write this as

\[\Delta = \text{Lab}^{<k}\]

and we still take the extremal value \(\bar{\ell}\) to be defined by the formula

\[\bar{\ell}(\delta) = \begin{cases} 
\ell & \text{if } \delta = \Lambda \\
\bot & \text{otherwise}
\end{cases}\]

Note that in the case \(k = 0\) we have \(\Delta = \{\Lambda\}\) which is equivalent to having no context information.

**Example 2.39** Consider the Fibonacci program of Example 2.33 and assume that we are only interested in recording the last call, i.e. \(k = 1\). Then the call strings of interest are:

\[\Lambda, [9], [4, 6]\]

Alternatively, we may choose to record the last two calls, i.e. \(k = 2\), in which case the following call strings are of interest:

\[\Lambda, [9], [9, 4], [9, 6], [4, 4], [4, 6], [6, 4], [6, 6]\]
In general, we would expect an analysis using these 8 contexts to be more precise than one using the 4 different contexts displayed above.

We shall now present the transfer functions for the general case where call strings have length at most \( k \). The transfer function \( \tilde{f}_{x} \) for procedure call is redefined by

\[
\tilde{f}_{x}(\delta) = \bigcup \{ f_{x}(\delta) \mid \delta' = [\delta, \epsilon]\} \]

where \([\delta, \epsilon]\) denotes the call string \([\delta, \epsilon]\) but possibly truncated (by omitting elements on the left) so as to have length at most \( k \). Since the function mapping \( \delta \) to \([\delta, \epsilon]\) is not injective (unlike the one mapping \( \delta \) to \([\epsilon, \delta]\)) we need to take the least upper bound over all \( \delta' \) that can be mapped to the relevant context \( \delta' \).

Similarly, the transfer function \( \tilde{f}_{x, y} \) for procedure return is redefined by

\[
\tilde{f}_{x, y}(\delta, \delta') = f_{x, y}(\delta, \delta') \cup \{[\delta, \epsilon] \mid \epsilon \in \delta'\}
\]

as should be expected.

**Example 2.40** Let us consider Detection of Sign Analysis in the special case where \( k = 0 \), i.e. where \( \Delta = \{A\} \) and hence \( \Delta \times (\text{Var}_{x} \rightarrow \text{Sign}) \) is isomorphic to \( \text{Var}_{x} \rightarrow \text{Sign} \). Using this isomorphism the formulae defining the transfer functions for procedure call can be simplified to

\[
\tilde{f}_{x}^{\text{sign}}(Y) = \bigcup \{ \phi_{x}^{\text{sign}}(s_{\text{sign}}) \mid s_{\text{sign}} \in Y \}
\]

\[
\tilde{f}_{x, y}^{\text{sign}}(Y, Y') = \bigcup \{ \phi_{x, y}^{\text{sign}}(s_{1}^{\text{sign}}, s_{2}^{\text{sign}}) \mid s_{1}^{\text{sign}} \in Y \land s_{2}^{\text{sign}} \in Y' \}
\]

where \( Y, Y' \subseteq \text{Var}_{x} \rightarrow \text{Sign} \). It is now easy to see that the analysis is context-insensitive: at procedure return it is not possible to distinguish between the different call points.

Let us next consider the case where \( k = 1 \). Here \( \Delta = \text{Lab} \cup \{A\} \) and the transfer functions for procedure call are:

\[
\tilde{f}_{x}^{\text{sign}}(Z) = \bigcup \{ \{\delta\} \times \phi_{x}^{\text{sign}}(s_{\text{sign}}) \mid (\delta, s_{\text{sign}}) \in Z \}
\]

\[
\tilde{f}_{x, y}^{\text{sign}}(Z, Z') = \bigcup \{ \{\delta\} \times \phi_{x, y}^{\text{sign}}(s_{1}^{\text{sign}}, s_{2}^{\text{sign}}) \mid (\delta, s_{1}^{\text{sign}}) \in Z \land (\delta, s_{2}^{\text{sign}}) \in Z' \}
\]

Now the transfer function \( \tilde{f}_{x}^{\text{sign}} \) will mark all data from the call point \( \epsilon \) with that label. Thus it does not harm that the information \( \tilde{f}_{x}^{\text{sign}}(Z) \) is merged with similar information \( \tilde{f}_{x}^{\text{sign}}(Z) \) from another procedure call. At the return from the call the transfer function \( \tilde{f}_{x, y}^{\text{sign}} \) selects those pairs \((\epsilon, s_{\text{sign}}) \in Z' \) that are relevant for the current call and combines them with those pairs \((\delta, s_{\text{sign}}) \in Z \) that describe the situation before the call; in particular, this allows us to reset the context to be that of the call point.

2.5.5 Assumption Sets as Context

An alternative to describing a path directly in terms of the calls being performed is to record information about the state in which the call was made; these methods can clearly be combined but in the interest of simplicity we shall abstain from doing so.

Large assumption sets. Throughout this subsection we shall make the simplifying assumption that

\[
L = \mathcal{P}(D)
\]

as is the case for the Detection of Signs Analysis. Much as in Examples 2.36 and 2.38 the property \( \mathcal{L} = \Delta \rightarrow L \) is then isomorphic to

\[
\tilde{\mathcal{L}} = \mathcal{P}(\Delta \times D)
\]

and we shall use this definition throughout this subsection. Restricting the attention to only recording information about the last call (corresponding to taking \( k = 1 \) above), one possibility is to take

\[
\Delta = \mathcal{P}(D)
\]

and we then take the extremal value to be

\[
\tilde{\epsilon} = \{(\{\epsilon\}, \epsilon)\}
\]

meaning that the initial context is described by the initial abstract state. This kind of context information is often called an assumption set and expresses a dependency on data (as opposed to a dependency on control as in the case of call strings).

**Example 2.41** Assume that we want to perform a Detection of Signs Analysis (Example 2.36) of the Fibonacci program of Example 2.33 and that the extremal value \( \epsilon_{\text{sign}} \) is the singleton \([x \mapsto +, y \mapsto -, z \mapsto -]\). Then the contexts of primary interest will be sets consisting of some of the following abstract states

\[
[x \mapsto +, y \mapsto 0, z \mapsto -], \quad [x \mapsto +, y \mapsto 0, z \mapsto 0], \quad [x \mapsto +, y \mapsto 0, z \mapsto +],
\]

\[
[x \mapsto +, y \mapsto +, z \mapsto -], \quad [x \mapsto +, y \mapsto +, z \mapsto 0], \quad [x \mapsto +, y \mapsto +, z \mapsto +]
\]

corresponding to the states in which the call-statements may be encountered.
For a procedure call \((l_1, e_1, e_2, e_3) \in I_1\), i.e. \([\text{call } p(a, x)]_{i_1}^{e_1}\) in the case of forward analysis, we define the transfer function \(f_{i_1}^{e_1}\) for procedure call by:

\[
 f_{i_1}^{e_1}(Z) = \bigcup \{ \delta' \times \phi_{i_1}^{e_1}(d) \mid (\delta, d) \in Z \land \delta' = (d' \mid (\delta, d') \in Z) \}
\]

where \(\phi_{i_1}^{e_1} : D \rightarrow \mathcal{P}(D)\). The idea is as follows: a pair \((\delta, d) \in Z\) describes a context and an abstract state for the current call. We now have to modify the context to take the call into account, i.e. we have to determine the set of possible abstract states in which the call could happen in the current context and this is \(\delta' = (d' \mid (\delta, d') \in Z)\). Given this context we proceed as in the call string formulations presented above and mark the data flow property with this context.

Next we shall consider the transfer function \(f_{i_1}^{e_1, e_2}\) for procedure return

\[
 f_{i_1}^{e_1, e_2}(Z, Z') = \bigcup \{ \delta' \times \phi_{i_1}^{e_2}(d, d') \mid (\delta, d) \in Z \land (\delta', d') \in Z' \land \delta' = (d' \mid (\delta, d') \in Z) \}
\]

where \(\phi_{i_1}^{e_2} : D \times D \rightarrow \mathcal{P}(D)\). Here \((\delta, d) \in Z\) describes the situation before the call and \((\delta', d') \in Z'\) describes the situation at the procedure exit. From the definition of \(f_{i_1}^{e_1}\) we know that the context matching \((\delta, d)\) will be \(\delta' = (d' \mid (\delta, d') \in Z)\) so we impose that condition. We can now combine information from before the call with that at the procedure exit much as in the call string approach; in particular, we can reset the context to be that of the call point.

There is one important snag with the definitions of the transfer functions \(f_{i_1}^{e_1}\) and \(f_{i_1}^{e_1, e_2}\): they are in general not monotone. One way to overcome this problem is to consider more general techniques for solving systems of equations where the transfer functions satisfy a weaker condition than monotonicity; we provide references to this approach in the Concluding Remarks.

Another way to overcome the problem is to use more approximate definitions that are indeed monotone: one possibility is to replace the condition \(\delta' = (d' \mid (\delta, d') \in Z)\) by \(\delta' \subseteq (d' \mid (\delta, d') \in Z)\). An even more approximate, but computationally more tractable, solution is to use smaller assumption sets as detailed below.

**Small assumption sets.** As a simpler version of using assumption sets one may take

\[ \Delta = D \]

and then use \(\mathcal{C} = \{(a, s)\}\) as the extremal value. So rather than basing the embellished Monotone Framework on \(\mathcal{P}(D) \times D\) as above we now base it on \(D \times D\). Of course, this is much less precise but, on the positive side, the size of the data flow properties has been reduced dramatically.

**2.5 Interprocedural Analysis**

For a procedure call \((l_1, e_1, e_2, e_3) \in I_1\), i.e. \([\text{call } p(a, x)]_{i_1}^{e_1}\) for forward analyses, the transfer function \(f_{i_1}^{e_1}\) is now defined by

\[
 f_{i_1}^{e_1}(Z) = \bigcup \{ (d) \times \phi_{i_1}^{e_1}(d) \mid (\delta, d) \in Z \}
\]

where, as before, \(\phi_{i_1}^{e_1} : D \rightarrow \mathcal{P}(D)\). Here the individual pieces of information concerning the abstract state of the call have their own local contexts; we have no way of grouping the abstract states corresponding to \(\delta\) as we did in the approach with large assumption sets.

The corresponding definition of the transfer function \(f_{i_1}^{e_1, e_2}\) for procedure return then is

\[
 f_{i_1}^{e_1, e_2}(Z, Z') = \bigcup \{ (\delta) \times \phi_{i_1}^{e_2}(d, d') \mid (\delta, d) \in Z \land (\delta', d') \in Z' \}
\]

where again \(\phi_{i_1}^{e_2} : D \times D \rightarrow \mathcal{P}(D)\). Examples of how to use assumption sets will be considered in the exercises.

**2.5.6 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. This may sound weird at first, but suppose that the analysis being performed is like the ones considered in Section 2.1 except that for simplicity all kill components are empty sets. Given these assumptions one might expect that the programs \(S_1; S_2\) and \(S_2; S_1\) give rise to the same analysis. 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.

**Sets of assigned variables.** We shall now present an example of a flow-insensitive analysis. Consider a program \(P\) of the form begin \(D, S\) end. For each procedure

\[
 \text{proc } p(\text{val } x, \text{res } y) \text{ is } \end{\}

in \(D\), the aim is to determine the set \(I_{AV}(p)\) of global variables that might be assigned directly or indirectly when \(p\) is called.

To compute these sets we need two auxiliary notions. The set \(AV(S)\) 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. It is defined inductively upon the structure of $S$:

\[
\begin{align*}
AV([\text{skip}]) &= \emptyset \\
AV([x := e]) &= \{x\} \\
AV(S_1 ; S_2) &= AV(S_1) \cup AV(S_2) \\
AV(\text{if } [b] \text{ then } S_1 \text{ else } S_2) &= AV(S_1) \cup AV(S_2) \\
AV(\text{while } [b] \text{ do } S) &= AV(S) \\
AV(\text{call } p(a,z)) &= \{z\}
\end{align*}
\]

Similarly we shall need the set $CP(S)$ 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. It is defined inductively upon the structure of $S$:

\[
\begin{align*}
CP([\text{skip}]) &= \emptyset \\
CP([x := e]) &= \emptyset \\
CP(S_1 ; S_2) &= CP(S_1) \cup CP(S_2) \\
CP(\text{if } [b] \text{ then } S_1 \text{ else } S_2) &= CP(S_1) \cup CP(S_2) \\
CP(\text{while } [b] \text{ do } S) &= CP(S) \\
CP(\text{call } p(a,z)) &= \{p\}
\end{align*}
\]

Both the sets $AV(S)$ and $CP(S)$ are well-defined by induction on the structure of $S$; also it should be clear that they are context-insensitive in the sense that any rearrangement of the statements inside $S$ would have given the same result. The information in $CP(\cdot)$ can be represented graphically: let the graph have a node for each procedure name as well as a node called main, for the program itself, and let the graph have an edge from $p$ (respectively $main$) to $p'$ whenever the procedure body $S$ of $p$ has $p' \in CP(S)$ (respectively $p' \in CP(S_a)$). This graph is usually called the procedure call graph.

We can now formulate a system of data flow equations that specifies how to obtain the desired sets $IAV(p)$:

\[
IAV(p) = (AV(S) \setminus \{z\}) \cup \bigcup \{IAV(p') \mid p' \in CP(S)\}
\]

where proc $p(\text{val } x, \text{res } y)$ is in $S$ and $x$ is in $D$.

By analogy with the considerations in Section 2.1 we want the least solution of this system of equations.

---

**Example 2.42** Let us now consider the following version of the Fibonacci program (omitting labels):

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

We then get the following equations:

\[
\begin{align*}
IAV(fib) &= (\emptyset \setminus \{z\}) \cup IAV(fib) \cup IAV(add) \\
IAV(add) &= \{y, u\} \setminus \{u\}
\end{align*}
\]

The associated procedure call graph is shown in Figure 2.11. The least solution to the equation system is

\[
IAV(fib) = IAV(add) = \{y\}
\]

showing that only the variable $y$ will be assigned by the procedure calls. (Had we instead taken the greatest solution to the equations we would have $IAV(fib) = IAV(add) = Var$, for any set $Var$, of variables that contains those used in the program and this would be completely unusable.

Note that the formulation of the example analysis did not associate information with entries and exits of blocks but rather with the blocks (or more generally the statements) themselves. This is a rather natural space saving approach for a context-insensitive analysis. It also relates to the discussion of Type and Effect Systems in Section 1.6: the “annotated base types” in Table 1.2 versus the “annotated type constructors” in Table 1.3.
2.6 Shape Analysis

We shall now study an extension of the *while*-language with heap allocated data structures and an intraprocedural Shape Analysis that gives a finite characterization of the shapes of these data structures. So while the aim of the previous sections has been to present the basic techniques of data flow analysis, the aim of this section is to show how the techniques can be used to specify a rather complex analysis.

Shape analysis information is not only useful for classical compiler optimizations but also for software development tools: the Shape Analysis will allow us to statically detect errors like dereferencing a nil-pointer — this is guaranteed to give rise to a dynamic error and a warning can be issued. Perhaps more surprisingly, the analysis allows us to validate certain properties of the shape of the data structures manipulated by the program; we can for example validate that a program for in-situ list reversal does indeed transform a non-cyclic list into a non-cyclic list.

**Syntax of the pointer language.** We shall study an extension of *while*-language that allows us to create cells in the heap; the cells are structured and may contain values as well as pointers to other cells. The data stored in a cell is accessed via selectors so we assume that a finite and non-empty set Sel of selector names are given:

\[ sel \in \text{Sel} \quad \text{selector names} \]

As an example Sel may include the Lisp-like selectors car and cdr for selecting the first and second components of pairs. The cells of the heap can be addressed by expressions like \( x.cdr \); this will first determine the cell pointed to by the variable \( x \) and then return the value of the cdr field. For the sake of simplicity we shall only allow one level of selectors although the development generalizes to several levels. Formally the pointer expressions

\[ p \in \text{PEExp} \]

are given by:

\[ p := x | x.\text{sel} \]

The syntax of the *while*-language is now extended to have:

\[ a := p \mid a_1 \mid a_2 \mid \text{nil} \]

\[ b := \text{true} \mid \text{false} \mid \text{not} \mid \text{and} \mid \text{or} \]

\[ S := [a] \mid [\text{skip}] \mid [S_1; S_2] \mid [\text{if} b \text{ then } S_1 \text{ else } S_2] \mid [\text{while } b \text{ do } S] \mid \text{malloc } p \]

Arithmetic expressions are extended to use pointer expressions rather than just variables, and an arithmetic expression can also be the constant nil.

2.6.1 Structural Operational Semantics

To model the scenario described above we shall introduce an infinite set Loc of locations (or addresses) for the heap cells:

\[ \xi \in \text{Loc} \quad \text{locations} \]

The value of a variable will now either be an integer (as before), a location (i.e. a pointer) or the special constant \( * \) reflecting that it is the nil value. Thus the states are given by

\[ s \in \text{State} = \text{Var.} \rightarrow (\mathbb{Z} + \text{Loc} + \{*\}) \]
where the use of partial functions with finite domain reflects that not all selector fields need to be defined; as we shall see later, a newly created cell with location $\xi$ will have all its fields to be undefined and hence the corresponding heap $\nu$ will have $\nu(\xi, sel)$ to be undefined for all $sel \in \text{Sel}$.

**Pointer expressions.** Given a state and a heap we need to determine the value of a pointer expression $p$ as an element of $\mathbb{Z} + \text{Loc} + \{\circ\}$. For this we introduce the function

$$p : \text{PExp}_p \rightarrow (\text{State} \times \text{Heap}) \rightarrow \mathbb{N} (\mathbb{Z} + \{\circ\} + \text{Loc})$$

where $\text{PExp}_p$ denotes pointer expressions with variables in $\text{Var}_p$. It is defined by:

$$\nu[x](\sigma, \nu) = \sigma(x)$$

$$p[x, sel](\sigma, \nu) = \begin{cases} \nu(\sigma(x), sel) & \text{if } \sigma(x) \in \text{Loc} \text{ and } \nu \text{ is defined on } (\sigma(x), sel) \\ \text{undef} & \text{if } \sigma(x) \notin \text{Loc} \text{ or } \nu \text{ is undefined on } (\sigma(x), sel) \end{cases}$$

The first clause takes care of the situation where $p$ is a simple variable and using the state we determine its value – note that this may be an integer, a location or the special null-value $\circ$. The second clause takes care of the case where the pointer expression has the form $x.sel$. Here we first have to determine the value of $x$; it only makes sense to inspect the $sel$-field in the case $x$ evaluates to a location that has a $sel$-field and hence the clause is split into two cases. In the case where $x$ evaluates to a location we simply inspect the heap $\nu$ to determine the value of the $sel$-field – again we may note that this can be an integer, a location or the special value $\circ$.

**Example 2.44** In Figure 2.12 the oval nodes model the cells of the heap $\nu$ and they are labelled with their location (or address). The unlabelled edges denote the state $\sigma$: an edge from a variable $x$ to some node labelled $\xi$ means that $\sigma(x) = \xi$; an edge from $x$ to the symbol $\circ$ means that $\sigma(x) = \circ$. The labelled edges model the heap $\nu$: an edge labelled $sel$ from a node labelled $\xi$ to a node labelled $\xi'$ means that there is a $sel$ pointer between the two cells, that is $\nu(\xi, sel) = \xi'$; an edge labelled $\circ$ from a node labelled $\xi$ to the symbol $\circ$ means that the pointer is a null-pointer, that is $\nu(\xi, sel) = \circ$.

Consider the pointer expression $x.cdr$ and assume that $\sigma$ and $\nu$ are as in row 0 of Figure 2.12, that is $\sigma(x) = \xi_1$ and $\nu(\xi_1, cdr) = \xi_2$. Then

$$p[x, cdr](\sigma, \nu) = \xi_2$$

**Arithmetic and boolean expressions.** It is now straightforward to extend the semantics of arithmetic and boolean expressions to handle pointer expressions and the null-constant. Obviously the functionality of the
Here the side condition ensures that the left hand side of the assignment does indeed evaluate to a location.

The construct malloc \( p \) is responsible for creating a new cell. We have two clauses depending on the form of \( p \):

\[
(malloc \ x, \sigma, \xi) \rightarrow (\sigma(x \mapsto \xi), \nu)
\]

where \( \xi \) does not occur in \( \sigma \) or \( \nu \)

\[
(malloc \ (x, sel) \sigma, \nu) \rightarrow (\sigma, \nu[(\sigma(x), sel) \mapsto \xi])
\]

where \( \xi \) does not occur in \( \sigma \) or \( \nu \) and \( \sigma(x) \in \text{Loc} \)

Note that in both cases we introduce a fresh location \( \xi \) but we do not specify any values for \( \nu(\xi, sel) \) - as discussed before we have settled for a semantics where the fields of \( \xi \) are undefined; obviously other choices are possible. Also note that in the last clause the side condition ensures that we already have a location corresponding to \( x \) and hence can create an edge to the new location.

Remark. The semantics only allows a limited reuse of garbage locations. For a statement like \([\text{malloc } x] \; ; \; [x:=\text{nil}] \; ; \; [\text{malloc } y] \; ; \; [x:=\text{nil}] \; ; \; [\text{malloc } y] \; ; \; [\text{malloc } x] \; ; \; [x:=\text{nil}] \; ; \; [\text{malloc } y] \; ; \; [x:=\text{nil}] \) we would not be able to reuse the location allocated at 1 although it will be unreachable (and hence garbage) after the statement labelled 3.

### 2.6.2 Shape Graphs

It should be evident that there are programs for which the heap can grow arbitrarily large. Therefore the aim of the analysis will be to come up with finite representations of it. To do so we shall introduce a method for combining the locations of the semantics into a finite number of abstract locations. We then introduce an abstract state \( S \) mapping variables to abstract locations (rather than locations) and an abstract heap \( H \) specifying the links between the abstract locations (rather than the locations). More precisely, the analysis will operate on shape graphs \((S, H, \approx)\) consisting of:

- an abstract state, \( S \),
- an abstract heap, \( H \), and
- sharing information, \( \approx \), for the abstract locations.

The last component allows us to recover some of the imprecision introduced by combining many locations into one abstract location. We shall now describe how a given state \( \sigma \) and heap \( \nu \) give rise to a shape graph \((S, H, \approx)\); in doing so we shall specify the functionality of \( S \), \( H \) and is in detail as well as formulate a total of five invariants.
Abstract locations. The abstract locations have the form \( n_X \) where \( X \) is a subset of the variables of \( \text{Var}_x \):

\[
\text{ALoc} = \{n_X \mid X \subseteq \text{Var}_x\}
\]

abstract locations

Since \( \text{Var}_x \) is finite it is clear that \( \text{ALoc} \) is finite and a given shape graph will contain a subset of the abstract locations of \( \text{ALoc} \).

The idea is that if \( x \in X \) then the abstract location \( n_X \) will represent the location \( \sigma(x) \). The abstract location \( n_Y \) is called the abstract summary location and will represent all the locations that cannot be reached directly from the state without consulting the heap. Clearly \( n_X \) and \( n_Y \) will represent disjoint sets of locations when \( X \neq \emptyset \).

In general, we shall enforce the invariant that two distinct abstract locations \( n_X \) and \( n_Y \) always represent disjoint sets of locations. As a consequence, for any two abstract locations \( n_X \) and \( n_Y \) it is either the case that \( X = Y \) or that \( X \cap Y = \emptyset \). To prove this assume by way of contradiction that \( X \neq Y \)

and that \( z \in X \cap Y \). From \( z \in X \) we get that \( \sigma(z) \) is represented by \( n_X \) and similarly \( z \in Y \) gives that \( \sigma(z) \) is represented by \( n_Y \). But then \( \sigma(z) \) must be distinct from \( \sigma(z) \) and we have the desired contradiction.

The invariant can be formulated as follows:

Invariant 1. If two abstract locations \( n_X \) and \( n_Y \) occur in the same shape graph then either \( X = Y \) or \( X \cap Y = \emptyset \).

Example 2.45 Consider the state and heap in row 2 of Figure 2.12. The variables \( x, y, z \) point to different locations \( \xi, \eta, \zeta \) respectively so in the shape graph they will be represented by different abstract locations named \( n_x, n_y, n_z \). The two locations \( \xi \) and \( \eta \) cannot be reached directly from the state so they will be represented by the abstract summary location \( n_S \).

Abstract states. One of the components of a shape graph is the abstract state, \( S \), that maps variables to abstract locations. To maintain the naming convention for abstract locations we shall ensure that:

Invariant 2. If \( x \) is mapped to \( n_X \) by the abstract state then \( x \in X \).

From Invariant 1 it follows that there will be at most one abstract location in the shape graph containing a given variable.

We shall only be interested in the shape of the heap so we shall not distinguish between integer values, nil-pointers and uninitialised fields; hence we can view the abstract state as an element of

\[
S \in \text{AState} = \mathcal{P}(\text{Var}_x \times \text{ALoc})
\]

where we have chosen to use powersets so as to simplify the notation in later parts of the development. We shall write \( \text{ALoc}(S) = \{n_X \mid \exists x : (x, n_X) \in S\} \) for the set of abstract locations occurring in \( S \). (Note that \( \text{AState} \) is too large in the sense that it contains elements that do not satisfy the invariants.)

Abstract heaps. Another component of the shape graph is the abstract heap, \( H \), that specifies the links between the abstract locations (just as the heap specifies the links between the locations in the semantics). The links will be specified by triples \( (n_Y, sel, n_W) \) and formally we take the abstract heap as an element of

\[
H \in \text{AHeap} = \mathcal{P}(\text{ALoc} \times \text{Sel} \times \text{ALoc})
\]

where we again do not distinguish between integers, nil-pointers and uninitialised fields. We shall write \( \text{ALoc}(H) = \{n_Y, n_W \mid \exists sel : (n_Y, sel, n_W) \in H\} \) for the set of abstract locations occurring in \( H \).

The intention is that if \( h((z_1, sel) = \xi_2 \) and \( \xi_1 \) and \( \xi_2 \) are represented by \( n_Y \) and \( n_W \) respectively, then \( (n_Y, sel, n_W) \in H \).

In the heap \( H \) there will be at most one location \( \xi_1 \) such that \( h((z_1, sel) = \xi_1 \). The abstract heap only partly shares this property because the abstract location \( n_S \) can represent several locations pointing to different locations.

However, the abstract heap must satisfy:

Invariant 3. Whenever \( (n_Y, sel, n_W) \) and \( (n_Y, sel, n_W) \) are in the abstract heap then either \( Y = \emptyset \) or \( W = \emptyset \).

Thus the target of a selector field will be uniquely determined by the source unless the source is the abstract summary location \( n_S \).

Example 2.46 Continuing Example 2.45 we can now see that the abstract state \( S_2 \) corresponding to the state of row 2 of Figure 2.12 will be

\[
S_2 = \{(x, n_x), (y, n_y), (z, n_z)\}
\]

The abstract heap \( H_2 \) corresponding to row 2 has

\[
H_2 = \{(n_x, cdr, n_x), (n_y, cdr, n_y), (n_z, cdr, n_z)\}
\]

The first triple reflects that the heap maps \( \xi_1 \) and \( \xi_1 \) to \( \zeta_1 \). \( \xi_1 \) is represented by \( n_x \) and \( \xi_1 \) is represented by \( n_y \). The second triple reflects that the heap maps \( \xi_z \) and \( \xi_z \) to \( \zeta_2 \). The final triple reflects that the heap maps \( \xi_2 \) and \( \xi_2 \) to \( \zeta_1 \). Note that there is no triple \( (n_z, cdr, n_y) \) because the heap maps \( \xi_1 \) and \( \xi_2 \) to \( \sigma \) rather than a location.

The resulting abstract state and abstract heap is illustrated in Figure 2.13 together with similar information for the other states and heaps of Figure 2.12.
2.12. The square nodes model abstract locations; the unlabelled edges from variables to square nodes model the abstract state and the labelled edges between square nodes model the abstract heap. If the abstract state does not associate an abstract location with some variable then that variable does not occur in the picture.

Note that even if the semantics use the same locations throughout the computation it need not be the case that the locations are associated with the same abstract locations at all points in the analysis. Consider Figures 2.12 and 2.13: the abstract location \( n_0 \) will in turn represent the locations \( \{x_1, x_2, x_3, x_4\}, \{x_1, x_2, x_3\}, \{x_3, x_4\}, \{x_1, x_4\}, \{x_1, x_3\}, \{x_1, x_4\}, \{x_1, x_3, x_4\} \). *Sharing information.* We are now ready to introduce the third and final component of the shape graphs. Consider the top row of Figure 2.14. The abstract state and abstract heap to the right represent the state and the heap shown in the second row. We shall now show how to distinguish between these two cases. The idea is to specify a subset, \( X \), of the abstract locations that represent locations that are shared due to pointers in the heap: an abstract location \( n_X \) will be included in \( X \) if it does represent a location that is the target of more than one pointer in the heap. In the top row of Figure 2.14, the abstract location \( n_{(x)} \) represents the location \( x_4 \) and it is not shared (by two or more heap pointers) so \( n_{(x)} \notin X \); the fat box indicates that the abstract location is unshared. On the other hand, in the second row \( x_4 \) is shared (both \( x_2 \) and \( x_4 \) point to it) so \( n_{(x_4)} \in X \); the double box indicates that the abstract location might be shared.

Obviously, the abstract heaps themselves also contain some implicit sharing information: this is illustrated in the bottom row of Figure 2.14 where there are two distinct edges with target \( n_{(x)} \). We shall ensure that this implicit sharing information is consistent with the explicit sharing information (as given by is) by imposing two invariants. The first ensures that information in the sharing component is also reflected in the abstract heap:

**Invariant 4.** If \( n_X \in X \) is then either

- (a) \( (n_0, sel, n_X) \) is in the abstract heap for some \( sel \), or
- (b) there exists two distinct triples \( (n_0, sel_1, n_X) \) and \( (n_0, sel_2, n_X) \) in the abstract heap (that is either \( sel_1 \neq sel_2 \) or \( V \neq W \)).

Case 4(a) takes care of the situation where there might be several locations represented by \( n_0 \) that point to \( n_X \) (as in the second and third rows of Figure 2.14). Case 4(b) takes care of the case where two distinct pointers (with different source or different selectors) point to \( n_X \) (as in the bottom row of Figure 2.14).

The second invariant ensures that sharing information present in the abstract heap is also reflected in the sharing component:

**Invariant 5.** Whenever there are two distinct triples \( (n_0, sel_1, n_X) \) and \( (n_0, sel_2, n_X) \) in the abstract heap and \( n_X \neq n_0 \) then \( n_X \in X \).
This takes care of the case where \( n_X \) represents a single location being the target of two or more heap pointers (as in the bottom row of Figure 2.14). Note that invariant 5 is the “inverse” of invariant 4(b) and that we have no “inverse” of invariant 4(a) - the presence of a pointer from \( n_S \) to \( n_X \) gives no information about sharing properties of \( n_X \).

In the case of the abstract summary location the explicit sharing information clearly gives extra information: if \( n_S \) is then there might be a location represented by \( n_S \) that is the target of two or more heap pointers, whereas if \( n_S \) is then all the locations represented by \( n_S \) will be the target of at most one heap pointer. The explicit sharing information may also give extra information for abstract locations \( n_X \) where \( X \neq \emptyset \): from 4(a) alone we cannot deduce that \( n_X \) is shared - this is clearly illustrated for the node \( n_F \) by the top two rows of Figure 2.14.

The complete lattice of shape graphs. To summarise, a shape graph is a triple consisting of an abstract state \( S \), an abstract heap \( H \), and a set of abstract locations that are shared:

\[
\begin{align*}
S & \in \text{AState} = \mathcal{P}({\text{Var}} \times \text{ALoc}) \\
H & \in \text{AHeap} = \mathcal{P}(\text{ALoc} \times \text{Sel} \times \text{ALoc}) \\
is & \in \text{IsShared} = \mathcal{P}(\text{ALoc})
\end{align*}
\]

where \( \text{ALoc} = \{ n_Z | Z \subseteq \text{Var}_S \} \). A shape graph \((S, H, is)\) is a compatible shape graph if it fulfills the five invariants presented above:

1. \( \forall n_V, n_W \in \text{ALoc}(S) \cup \text{ALoc}(H) \cup \text{Is} : (V = W) \lor (V \cap W = \emptyset) \)
2. \( \forall (x, n_X) \in S : x \in X \)
3. \( \forall (n_V, \text{sel}, n_W) \in H : (V = \emptyset) \lor (W = W') \)
4. \( \forall n_X \in \text{Is} : (3\text{sel} : (n_Q, n_S, n_X) \in H) \lor ((3\text{sel} \neq n_S, n_X) \in H : \text{sel}_1 \neq \text{sel}_2 \lor V \neq W) \)
5. \( \forall (n_V, \text{sel}_1, n_X) \in H : (\text{sel}_1 \neq \text{sel}_2 \lor V \neq W) \wedge X \neq \emptyset) \Rightarrow n_X \in \text{Is} \)

The set of compatible shape graphs is denoted

\[ \text{SG} = \{ (S, H, is) | (S, H, is) \text{ is compatible} \} \]

and the analysis, to be called \( \text{Shape} \), will operate over sets of compatible shape graphs, i.e. elements of \( \mathcal{P}(\text{SG}) \). Since \( \mathcal{P}(\text{SG}) \) is a powerset, it is trivially a complete lattice with \( \cup \) being \( \cup \) and \( \subseteq \) being \( \subseteq \). Furthermore, \( \mathcal{P}(\text{SG}) \) is finite because \( \text{SG} \subseteq \text{AState} \times \text{AHeap} \times \text{IsShared} \) and all of \( \text{AState} \), \( \text{AHeap} \) and \( \text{IsShared} \) are finite.

![Figure 2.15: The single shape graph in the extremal value \( \ell \) for the list reversal program.](image)

2.6.3 The Analysis

The analysis will be specified as an instance of a Monotone Framework with the complete lattice of properties being \( \mathcal{P}(\text{SG}) \). For each label consistent program \( S \), with isolated entries we obtain a set of equations of the form

\[
\text{Shape}_e(\ell) = \begin{cases} 
\ell & \text{if } \ell = \text{init}(S_e) \\
\bigcup \{ \text{Shape}_e(\ell') | (\ell', \ell) \in \text{flow}(S_e) \} & \text{otherwise}
\end{cases}
\]

\[
\text{Shape}_e(\ell) = f^A_\ell(\text{Shape}_B(\ell))
\]

where \( \ell \in \mathcal{P}(\text{SG}) \) is the extremal value holding at entry to \( S \), and \( f^A_\ell \) are the transfer functions to be developed below.

The analysis is a forward analysis since it is defined in terms of the set \( \text{flow}(S_e) \), and it is a map analysis since we are using \( \cup \) as the combination operation. However, there are also aspects of a must analysis because each individual shape graph must contain any superfloors information.

This will be useful for achieving strong update and strong nullification; here “strong” means that an update or nullification of a pointer expression allows one to remove the existing binding before adding a new one. This in turn leads to a very powerful analysis.

Example 2.47 Consider again the list reversal program of Example 2.43 and assume that \( x \) initially points to an unshared list with at least two elements and that \( y \) and \( z \) are initially undefined; the singleton shape graph corresponding to this state and heap is illustrated in Figure 2.15 and will be the extremal value \( \ell \) used throughout this development.

The Shape Analysis computes the sets \( \text{Shape}_e(\ell) \) and \( \text{Shape}_e(\ell) \) of shape graphs describing the state and heap before and after executing the elementary block labelled \( \ell \). The equations for \( \text{Shape}_e(\ell) \) are

\[
\begin{align*}
\text{Shape}_e(1) &= f^A_1(\text{Shape}_e(1)) = f^A_1(1) \\
\text{Shape}_e(2) &= f^A_2(\text{Shape}_e(2)) = f^A_2(\text{Shape}_e(1) \cup \text{Shape}_e(6)) \\
\text{Shape}_e(3) &= f^A_3(\text{Shape}_e(3)) = f^A_3(\text{Shape}_e(2)) \\
\text{Shape}_e(4) &= f^A_4(\text{Shape}_e(4)) = f^A_4(\text{Shape}_e(3)) \\
\text{Shape}_e(5) &= f^A_5(\text{Shape}_e(5)) = f^A_5(\text{Shape}_e(4))
\end{align*}
\]
Transfer function for \([x:=n1]^t\) where \(a\) is of the form \(n, a_1, p_a, a_2\) or \(nil\). The effect of this assignment will be to remove the binding to \(x\), and to rename all abstract locations so that they do not include \(x\) in their name. The renaming of abstract locations is specified by the function
\[
k_x(x) = n_{\text{is}}(x)
\]
and we then take
\[
\phi^A_x((S, H, is)) = \{ \text{kill}_x((S, H, is)) \}
\]
where \(\text{kill}_x((S, H, is)) = (S', H', is')\) is given by
\[
S' = \{ (x, k_x(n_x)) \mid (x, n_x) \in S \land x \neq x \}
\]
\[
H' = \{ (k_x(n_x), sel, k_x(n_w)) \mid (n_x, sel, n_w) \in H \}
\]
\[
is' = \{ k_x(n_x) \mid n_x \in is \}
\]
so that we obtain strong nullification. It is easy to check that if \((S, H, is)\) is compatible then so is \((S', H', is')\).

Example 2.49 The statement \([y:=n1]^t\) of the list reversal program of Example 2.43 is of the form considered here. Since there is no occurrence of \(y\) in the single shape graph in \(z\) of Figure 2.15, the shape graph \(\text{Shape}_x(1)\) in Example 2.47 is equal to \(x\).

An interesting case is when \((x, n_{\text{is}}) \in S\) since this will cause the two abstract locations \(n_{\text{is}}\) and \(n_4\) to be merged. The sharing information is then updated to capture that we can only be sure that \(n_4\) is unshared in the updated shape graph if both \(n_4\) and \(n_{\text{is}}\) were unshared in the original shape graph. This is illustrated in Figure 2.16: the left hand picture shows the interesting parts of the shape graph \((S, H, is)\) and the right hand picture shows the corresponding parts of \((S', H', is')\). We shall assume that the square boxes represent distinct abstract locations so in particular \(V, [x]\), \(W\) and \(U\) are all distinct sets. The fat boxes represent unshared abstract locations as before, the thin boxes represent abstract locations whose sharing information is not affected by the transfer function, and unlabelled edges between abstract locations represent pointers that are unaffected by the transfer function.

Example 2.50 The statement \([x:=n1]^t\) of the list reversal program of Example 2.43 illustrates this case: for each of the shape graphs of \(\text{Shape}_x(2)\) the abstract location \(n_{\text{is}}\) is merged with \(n_4\) to produce one of the shape graphs of \(\text{Shape}_x(7)\).

Remark. The analysis does not perform garbage collection: it might be the case that there are no heap pointers to \(n_{\text{is}}\) and then the corresponding location in the heap will be unreachable after the assignment. Nonetheless the analysis will merge the two abstract locations \(n_{\text{is}}\) and \(n_4\) and insist on a pointer from \(n_4\) to any abstract location that \(n_{\text{is}}\) might point to.
Transfer function for $[x := y]^t$. If $x = y$ then the transfer function $f_t^{SA}$ is just the identity.

Next suppose that $x \neq y$. The first effect of the assignment is to remove the old bindings to $x$; for this we use the kill$_{x}$ operation introduced above. Then the new binding to $x$ is recorded; this includes renaming the abstract location that includes $y$ in its variable set to also include $x$. The renaming of the abstract locations is specified by the function:

$$g_t^{SA}(n_g) = \begin{cases} n_{yU(a)} & \text{if } y \in Z \\ n_g & \text{otherwise} \end{cases}$$

We shall then take

$$f_t^{SA}((S, H, is)) = ((S'', H'', is''))$$

where $(S', H', is') = kill_x((S, H, is))$ and

$$S'' = \{(x, g_t^{SA}(n_x)) \mid (x, n_x) \in S'\}$$

$$H'' = \{(y, n_y) \mid (y, n_y) \in H' \land y' = y\}$$

$$is'' = \{n_{yU(a)} \mid n_y \in is'\}$$

so that we obtain strong update. Here the second clause in the formula for $S''$ adds the new binding to $x$. Again we note that if $(S, H, is)$ is compatible then so is $(S'', H'', is'')$.

The clause is illustrated in Figure 2.17 where we assume that nodes represent distinct abstract locations; it follows from the invariants that $y \in Y$ but $y \notin V$ and $y \notin W$. Note that $n_{yU(a)}$ inherits the sharing properties of $n_Y$ although

Both $x$ and $y$ will point to the same cell; the reason is that the sharing information only records sharing in the heap - not sharing via the state.

**Example 2.51** The statement $[y := x]^t$ of the list reversal program of Example 2.43 is of the form considered here: each of the shape graphs of Shape$_r(3)$ in Example 2.47 is transformed into one of the shape graphs of Shape$_r(4)$.

Also the statement $[z := y]^t$ is of the form considered here: each of the shape graphs of Shape$_r(2)$ is transformed into one of those of Shape$_r(3)$.

**Transfer function for $[x := y, sel]^t$.** First assume that $x = y$; then the assignment is semantically equivalent to the following sequence of assignments

$$[t := y, sel]^t; [x := t]^t; [t := n1]^t$$

where $t$ is a fresh variable and $\ell_1$, $\ell_2$ and $\ell_3$ are fresh labels. The transfer function $f_t^{SA}$ can therefore be obtained as

$$f_t^{SA} = f_{\ell_3} \circ f_{t} \circ f_{\ell_1}$$

where the transfer functions $f_{\ell_3}$ and $f_{t}$ follow the pattern described above.

We shall therefore concentrate on the transfer function $f_{\ell_3}$, or equivalently, $f_{\ell_1}$ in the case where $x \neq y$.

**Example 2.52** The statement $[x := x, cdr]^t$ of the list reversal program of Example 2.43 is transformed into $[t := x, cdr]^t; [x := t]^t; [t := n1]^t$. We shall return to the analysis of $[t := x, cdr]^t$ later.

So assume that $x \neq y$ and let $(S, H, is)$ be a compatible shape graph before the analysis of the statement. As in the previous case, the first step will be to remove the old binding for $x$ and again we use the auxiliary function kill$_{x}$:

$$(S', H', is') = kill_x((S, H, is))$$

The next step will be to rename the abstract location corresponding to $y, sel$ to include $x$ in its name and to establish the binding of $x$ to that abstract location. We can now identify three possibilities:

1. There is no abstract location $n_Y$ such that $(y, n_Y) \in S'$ or there is an abstract location $n_Y$ such that $(y, n_Y) \in S'$ but no $n_z$ such that $(n_Y, sel, n_z) \in H'$; in this case the shape graph will represent a state and a heap where $y$ or $y, sel$ is an integer, nil or undefined.

2. There is an abstract location $n_Y$ such that $(y, n_Y) \in S'$ and there is an abstract location $n_Y \neq n_z$ such that $(n_Y, sel, n_z) \in H'$; in this case the shape graph will represent a state and a heap where the location pointed to by $y, sel$ will also be pointed to by some other variable (in $U$).
2.6 Shape Analysis

We shall then take
\[
\phi^S((S, H, is)) = ((S'', H'', is''))
\]
where \((S', H', is') = kill_e((S, H, is))\) and
\[
S'' = \{(x, h''_x(n_x)) \mid (x, n_x) \in S'\} \cup \{(x, h''_x(n_x))\}
\]
\[
H'' = \{h''_v(n_v), sel', h''_w(n_w) \mid (n_v, sel', n_w) \in H'\}
\]
\[
is'' = \{h''_x(n_x) \mid n_x \in is'\}
\]
The inclusion of \((x, h''_x(n_x))\) in \(S''\) reflects the assignment. The definition of \(is''\) ensures that sharing is preserved by the operation; in particular, \(nu_{U(is')}\) is shared in \(H''\) if and only if \(nu_{is}\) is shared in \(H'\).

The effect of the assignment is illustrated in Figure 2.18 in the case where \(n_x \in is\). As before we assume that the abstract locations shown on the figure are distinct in particular \(Y\) and \(V\) are all different from \(U\).

Case 3. We now consider the statement \([x := y, sel']\) (where \(x \neq y\)) in the case where there is an abstract location \(n_x\) such that \((y, n_x) \in S'\) and furthermore \((n_y, sel', n_x) \in H'\). As before the invariants ensure that \(n_x\) is uniquely determined. The location \(n_x\) describes the location for \(y, sel\) as well as a (possibly empty) set of other locations. We now have to materialize a new abstract location \(n_x\) from \(n_x\); then \(n_x\) will describe the location for \(y, sel\) and \(n_x\) will continue to represent the remaining locations. Having introduced a new abstract location we will have to modify the abstract heap accordingly.

This is a potentially difficult operation, so let us consider the following sequence of assignments:

\[
[x:= n_1]; [x := y, sel'] ; [x := n_2]
\]

Clearly \([x := n_1]; [x := y, sel']\) is equivalent to \([x := y, sel']\) both in terms of the analysis and the semantics. Indeed, \((S'', H'', is'') = kill_e((S, H, is))\) represents the effect of removing the binding to \(x\). We are trying to determine candidate shape graphs \((S'', H'', is'')\) holding after the assignment \([x := y, sel']\) (where \(x \neq y\)) but let us first study our expectations of \((S'', H'', is'')\) It is immediate that \((S'', H'', is'') = kill_e((S', H', is'))\). Furthermore, the stack and heaps possible at the point described by \((S', H', is')\) should be the same as those possible at the point described by \((S'', H'', is'')\). This suggests demanding that
\[
(S'', H'', is'') = (S', H', is')
\]
which means that \( \text{kill}_a((S'', H'', \text{is}'')) = (S', H', \text{is}') \). It is also immediate that \((x, n_{(x)}) \in S''\) and that \((n_{y}, \text{sel}, n_{(x)}) \in H''\).

We shall then take

\[
\phi_2^A((S, H, \text{is})) = \{(S'', H'', \text{is}'') \mid \text{is}\}
\]

\[
\text{kill}_a((S'', H'', \text{is}'')) = (S', H', \text{is}') \land
\]

\[
(x, n_{(x)}) \in S'' \land (n_{y}, \text{sel}, n_{(x)}) \in H''
\]

where \((S', H', \text{is}') = \text{kill}_a((S, H, \text{is}))\).

It is hopefully clear that we have not missed any shape graphs \((S'', H'', \text{is}'')\) that might be the result of the assignment. What might be a worry is that we have included an excessive amount of irrelevant shape graphs. (Indeed producing all compatible shape graphs would be trivially sound but also utterly useless.) Although it is possible to do slightly better (see Exercise 2.23) we shall now argue that amount of imprecision in the above definition is not excessive.

We first establish that

\[
S'' = S' \cup \{(x, n_{(x)})\}
\]

showing that the abstract state is fully determined. Consider \((x, n_{(x)}) \in S''\). If \(x = x\) it follows from the compatibility of \((S'', H'', \text{is}'')\) that \(n_{(x)} = n_{(x)}\). If \(x \neq x\) it follows from \((x, n_{(x)}) \in S''\) and the compatibility of \((S'', H'', \text{is}'')\) that \(x \not\in Z\) and hence \((x, n_{(x)}) = (x, k_a(n_{(x)})\). This establishes that \(S'' \subseteq S' \cup \{(x, n_{(x)})\}\). Next consider \((u, n_{(u)}) \in S'\). We know that \(u \neq x\) and \(x \not\in U\) from the definition of \(S'\) and from compatibility of \((S', H', \text{is}')\). There must exist \((u, n_{(u)}) \in S''\) such that \(k_a(n_{(u)}) = n_{(u)}\) but since \(x \neq u\) this gives \(n_{(u)} = n_{(u)}\). It follows that \(S'' \supseteq S' \cup \{(x, n_{(x)})\}\) and we have proved the required equality.

We next establish that

\[
is' \setminus \{n_{y}\} = \text{is}'' \setminus \{n_{y}, n_{(x)}\}
\]

\[
n_{y} \in \text{is}' \iff n_{y} \in \text{is}'' \lor n_{(x)} \in \text{is}''
\]

showing that

- abstract locations apart from \(n_{y}\) retain their sharing information,
- if \(n_{y}\) is shared then that sharing cannot go away but must give rise to sharing of at least one of \(n_{y}\) or \(n_{(x)}\), and
- if \(n_{y}\) is not shared then no sharing can be introduced for \(n_{y}\) or \(n_{(x)}\).

Since both \((S', H', \text{is}')\) and \((S'', H'', \text{is}'')\) are compatible shape graphs it follows that if \(n_{y} \in \text{is}'\) then \(x \not\in U\) and if \(n_{y} \in \text{is}''\) then \(x \not\in U\). Hence \(\text{is}' = \{k_a(n_{y}) \mid n_{y} \in \text{is}''\}\) establishes \(\text{is}' \setminus \{n_{y}\} = \text{is}'' \setminus \{n_{y}, n_{(x)}\}\).

Figure 2.19: The effect of \([x := y, \text{sel}]^*\) in a special case (part 1).

because \(k_a(n_{y}) = n_{y} \neq n_{y}\) for all \(n_{y} \in \text{is}'' \setminus \{n_{y}, n_{(x)}\}\). Furthermore, \(n_{y} \in \text{is}'' \lor n_{(x)} \in \text{is}''\) gives \(n_{y} \in \text{is}',\) and \(n_{y} \notin \text{is}'' \land n_{(x)} \notin \text{is}''\) gives \(n_{y} \notin \text{is}'\).

Thus we have established the required relationship.

We now turn to the abstract heap. We shall classify the labelled edges \((n_{v}, \text{sel}', n_{w})\) into four groups depending on whether or not the source or target may be one of the nodes \(n_{y}\) or \(n_{(x)}\): 

\[
(n_{v}, \text{sel}', n_{w}) \text{ is external} \iff \{n_{v}, n_{w}\} \cap \{n_{y}, n_{(x)}\} = \emptyset
\]

\[
(n_{v}, \text{sel}', n_{w}) \text{ is internal} \iff \{n_{v}, n_{w}\} \subseteq \{n_{y}, n_{(x)}\}
\]

\[
(n_{v}, \text{sel}', n_{w}) \text{ is going-out} \iff n_{v} \in \{n_{y}, n_{(x)}\} \land n_{w} \notin \{n_{y}, n_{(x)}\}
\]

\[
(n_{v}, \text{sel}', n_{w}) \text{ is going-in} \iff n_{v} \notin \{n_{y}, n_{(x)}\} \land n_{w} \in \{n_{y}, n_{(x)}\}
\]

We shall also say that two edges \((n_{v}, \text{sel}', n_{w})\) and \((n_{v}', \text{sel}', n_{w}')\) are related if and only if \(k_a(n_{v}) = k_a(n_{v}')\), \(\text{sel}' = \text{sel}''\) and \(k_a(n_{w}) = k_a(n_{w}')\). Clearly an external edge is related only to itself.

Reasoning as above one can show that:

- \(H'\) and \(H''\) have the same external edges,
- each internal edge in \(H'\) is related to an internal edge in \(H''\) and vice versa,
- each edge going-out in \(H'\) is related to an edge going-out in \(H''\) and vice versa, and
- each edge going-in in \(H'\) is related to an edge going-in in \(H''\) and vice versa.
transfer function is shown in Figure 2.20. First note that the going-in edge \((n_V, sel, n_0) \in H\) is changed to \((n_V, sel, n_{(a)}) \in H'\) in all shape graphs. Next note that the going-in edge labelled sel will only point to \(n_0\) because \(n_{(a)}\) is not shared (as \(n_0\) is not) and \(n_V\) points to \(n_{(a)}\). The going-out edge labelled sel can start at both \(n_0\) and \(n_{(a)}\) but it cannot do so simultaneously because \(n_{(a)}\) is not shared. The internal edge labelled sel can only point to \(n_0\) because \(n_{(a)}\) is not shared and \(n_V\) points to \(n_{(a)}\); it can start at both \(n_0\) and \(n_{(a)}\) and can even do so simultaneously. This explains why there are six shape graphs in \(\phi_S^A((S, H, Is))\), all of which are clearly needed.

**Example 2.53** The statement \([x := n_a\ cdr]^{(a)}\) introduced in Example 2.52 is of the form considered here: the transfer function will transform each of the shape graphs of Shape\(_2\,(4)\) and subsequent transformations will produce Shape\(_2\,(5)\).

**Transfer function for** \([x, sel := n_{(a)}]^{(a)}\) where \(a\) is of the form \(n, a_1, a_2, o_a\) or \(o\). Again we consider a compatible shape graph \((S, H, Is)\). First assume that there is no \(n_X\) such that \((x, n_X) \in S\); then \(x\) will not point to a cell in the heap and the statement will have no effect on the shape of the heap so the transfer function \(f^{(a)}_{x, sel}\) is just the identity. Next assume that there is a (necessarily unique) \(n_X\) such that \((x, n_X) \in S\) but that there is no \(n_U\) such that \((n_X, sel, n_U) \in H\); then the cell pointed to by \(sel\) does not point to another cell so the statement will not change the shape of the heap and also in this case the transfer function \(f^{(a)}_{x, sel}\) will be the identity.

The interesting case is when there are abstract locations \(n_X\) and \(n_U\) such that \((x, n_X) \in S\) and \((n_X, sel, n_U) \in H\); these abstract locations will be unique because of the invariants. The effect of the assignment will be to remove the triple \((n_X, sel, n_U)\) from \(H\):

\[
\phi_S^A((S, H, Is)) = \{kill_{x, sel}((S, H, Is))\}
\]

where \(kill_{x, sel}((S, H, Is)) = (S', H', Is')\) is given by:

\[
S' = S
\]

\[
H' = \{ (n_V, sel', n_W) \mid (n_V, sel, n_W) \in H \land (X = V \land sel = sel') \}
\]

\[
is' = \begin{cases} is \setminus \{n_U\} & \text{if } n_U \in Is \land \#into(n_U, H') \leq 1 \land \\
\delta sel : (n_0, sel', n_U) \in H' & \text{otherwise}
\end{cases}
\]

The sharing information is as before except that we may be able to do better for the node \(n_U\) - we have removed one of the pointers to it and in the case where there is at most one pointer left and it does not have source \(n_0\) the corresponding location will be unshared. This is yet another aspect of strong update. Here we write \#into\((n_U, H')\) for the number of pointers to \(n_U\) in \(H'\).

This clause is illustrated in Figure 2.21.
Transfer function for \([z \text{ sel} := y]^t\). First assume that \(z = y\). The statement is then semantically equivalent to
\[
[t := y]^t; [z \text{ sel} := t]^t; [t := nil]^t
\]
where \(t\) is a fresh variable and \(\ell_1, \ell_2, \ell_3\) are fresh labels. The transfer function \(f_t^{SA}\) is then given by
\[
f_t^{SA} = f_{\ell_1}^{SA} \circ f_{\ell_2}^{SA} \circ f_{\ell_3}^{SA}
\]
The transfer functions \(f_{\ell_1}^{SA}\) and \(f_{\ell_2}^{SA}\) follow the pattern we have seen before so we shall concentrate on the clause for \(f_{\ell_2}^{SA}\), or equivalently, \(f_t^{SA}\) in the case where \(z \neq y\).

So assume that \(z \neq y\) and that \((S, H, iS)\) is a compatible shape graph. It may be the case that there is no \(n_x\) such that \((x, n_x) \in S\) and in that case the transfer function will be the identity since the statement cannot affect the shape of the heap.

So assume that \(n_x\) satisfies \((x, n_x) \in S\). The case where there is no \(n_y\) such that \((y, n_y) \in S\) corresponds to a situation where the value of \(y\) is an integer, the nil-value or undefined and is therefore similar to the case \([z \text{ sel} := \text{ nil}]^t\):
\[
\phi_t^{SA}(S, H, iS) = \{\text{kill}_{z \text{ sel}}(S, H, iS)\}
\]
The interesting case is when \(x \neq y\), \((x, n_x) \in S\) and \((y, n_y) \in S\). The first step will be to remove the binding for \(x\) \text{ sel} and for this we can use the \(\text{kill}_{x \text{ sel}}\) function. The second step will be to establish the new binding. So we take
\[
\phi_t^{SA}(S, H, iS) = \{(S'', H'', iS'')\}
\]
The statement \( \text{malloc}(x, \text{set}) \) is equivalent to the sequence
\[
\text{malloc}(t1^1; [x, \text{set} := t1^0]; t := t1) \]
where \( t \) is a fresh variable and \( t1, t2 \) and \( t3 \) are fresh labels. The transfer function \( f^{t_3}_{t_1} \) is then
\[
f^{t_3}_{t_1} = f^{t_2}_{t_1} \circ f^{t_3}_{t_2} \circ f^{t_2}_{t_1}
\]
The transfer functions \( f^{t_3}_{t_1}, f^{t_2}_{t_1} \) and \( f^{t_3}_{t_2} \) all follow the patterns we have seen before so this completes the specification of the transfer function.

Concluding Remarks

Data Flow Analysis for imperative languages. As mentioned in the beginning of this chapter, Data Flow Analysis has a long tradition. Most compiler textbooks contain sections on optimisation which mainly discuss Data Flow Analyses and their implementation [5, 55, 181]. The emphasis in these books is often on practical implementations of data flow analyses. A classic textbook which provides a more theoretical treatment of the subject is by Hecht [99]; the book contains a detailed discussion of the examples Data Flow Analyses in Section 2.1, and also presents a more traditional treatment of Monotone Frameworks based on the use of semi-lattices as well as a number of algorithms (see Chapter 6 for a more thorough treatment of algorithms). Marlowe and Hyder [103] provide a survey of data flow frameworks. Steffen [154] and Schmidt [151] express data flow analyses using modal logic (rather than equations) thereby opening up the possibility of using model checking techniques for program analysis.

The examples presented in Section 2.1 are fairly standard. Alternative treatments of this material can be found in any of the books already cited. The examples may all be represented as Bit Vector Frameworks (see Exercise 2.9): the lattice elements may be represented by a vector of bits and the lattice operations efficiently implemented as boolean operations. The method used in Section 2.2 to prove the correctness of the Live Variables Analysis is adapted from [112] and is expressed by means of an operational semantics [140, 130]. The notion of faint variables, introduced in Exercise 2.4, was first introduced by Giegerich, Mötschke and Wilhelm [56].

The use of semi-lattices in Data Flow Analysis was first proposed in [96]. The notion of Monotone Frameworks is due to Kam and Ullman [93]. These early papers, and much of the later literature, use the dual notions (meets and maximal fixed points) to our presentation. Kam and Ullman [93] prove that the existence of a general algorithm to compute MOP solutions would imply the decidability of the Modified Post Correspondence Problem [76]. Cousot and Cousot [37] model abstract program properties by complete semi-lattices in their paper on Abstract Interpretation (see Chapter 4).

We have associated transfer functions with elementary blocks. It would be possible to associate transfer functions with flows instead as e.g. in [147]. These two approaches have equal power: to go from the first to the second, the transfer functions may be moved from the blocks to their outgoing flows; to go from the second to the first, we can introduce artificial blocks. In fact artificial blocks can be avoided as shown in Exercise 2.11.

Most of the papers that we have cited so far concentrate on intraprocedural analysis. An early, influential, paper on interprocedural analysis is [155] that studies two approaches to establishing context. One is based on call strings and expresses aspects of the dynamic calling context; our presentation is inspired by [178]. The other is the “functional approach” that is based on data and that shares some of the aims of abstraction sets [90, 138, 146]; the technical formulation is different because [155] obtains the effect by calculating the transfer functions for the call statement. Most of the subsequent papers in the literature can be seen as variations and combinations over this theme; a substantial effort in this direction may be found in [44]. As mentioned in Section 2.5.5, the use of large assumption sets may lead to equation systems where the transfer functions are not monotone; we refer to [51, 52] for a modern presentation of techniques that allow the solution of so-called weakly monotone systems of equations.

Point analysis. There is an extensive literature on the analysis of alias problems for languages with pointers. Following [62] we can distinguish between analyses of pointers to (1) statically allocated data (typically on the stack) and (2) dynamically allocated data (typically in the heap). The analysis of pointers to statically allocated data is the simplest; typically the data will have compile-time names and the analysis result can be presented as a set of points-to pairs of the form \((p, x)\) meaning that the pointer \(p\) points to the data \(x\) or as alias pairs of the form \((a, p, x)\) meaning that \(a\) and \(x\) are aliased. Analyses in this category include [47, 100, 148, 182, 162, 158].

The analysis of dynamically allocated data is more complicated since the objects of interest are inherently anonymous. The simplest analyses [38, 61] study the connectivity of the heap: they attempt to split the heap into disjoint parts and do not keep any information about the internal structure of the individual parts. These analyses have been found quite useful for many applications.

The more complex analyses of dynamically allocated data give more precise information about the shape of the heap. A number of approaches use graphs to represent the heap. A main distinction between these approaches is how they map a heap of potentially unbounded size to a graph of bounded size: some bound the length of paths in the heap [84, 155], others merge heap cells created at the same program point [85, 30], and yet others merge heap cells that cannot be kept apart by the set of pointer variables pointing to them [148, 149]. Another group of analyses obtain information about the shape.
of the heap by more directly approximating the access paths. Here a main
distinction is the kind of properties of the access paths that are recorded:
some focus on simple connectivity properties [62], others use some limited
form of regular expressions [101], and yet others use monomial relations [45].
The analysis presented in Section 2.6 is based on the work of Sagiv, Reps
and Wilhelm [148, 149]. In contrast to [148, 149] it uses sets of composable
shape graphs; [148, 149] merge sets of compatible shape graphs into a single
summary shape graph and then use various mechanisms for extracting parts
of the individual compatible shape graphs and in this way an exponential
factor in the cost of the analysis can be avoided. The sharing component
of the shape graphs is designed to detect list-like properties; it can be replaced
by other components detecting other shape properties [150].

Static Single Assignments. Some program analyses can be performed
more efficiently or more accurately when the program is transformed
into an intermediate form called static single assignment (SSA) form [42].
The benefit of SSA form is that the definition-use chains of the program are
explicit in the representation: each variable has at most one definition (mean-
ing that it is assigned at most once). As a consequence some optimisations
can be performed more efficiently using this representation [12, 150, 109, 110].
The transformation to SSA form amounts to renaming the variables and
introducing special assignments at the points where flow of control might
join. The assignments use so-called ϕ-functions; each argument position of
the ϕ-function identifies one of the program points where flow of control might
come from. The special statements have the form \( x := \phi(x_1, \ldots, x_n) \) and the
idea is that the value of \( x \) will equal the value of \( x_i \) whenever control comes
from the \( i \)-th predecessor. The algorithms for transforming to SSA form
often proceed in two stages: the first stage identifies the points where flow
of control might join and where the special assignments are to be inserted,
and the second stage renames the variables to ensure that each of them is
assigned at most once. To obtain a compact representation of the program
one wants to minimise the number of extra statements (and variables) and
there are techniques based on additional data flow information for achieving
this.

Data Flow Analysis for other language paradigms. The
analysis techniques that we have studied assume the existence of some
representation of the flow of control in the program. For the class of imperative
languages that we have studied, it is relatively easy to determine this control
flow information. For many languages, for example functional programming
languages, this is not the case. The next chapter presents techniques for
determining control flow information for such languages and shows how Data
Flow Analysis can be integrated with control flow analysis.

Concluding Remarks

The techniques we have presented can be applied directly to other language
paradigms. Two examples are in object-oriented programming and a commu-
nicating processes language. Vitek, Horspool and Uhl [178] present an analy-
sis for object-oriented languages which determines classes of objects and their
lifetimes. Their analysis is an interprocedural analysis that uses a graph-
based representation of the memory as data flow values. Reif and Smolka
[142] apply Data Flow Analysis techniques to distributed communicating
processes to detect unreachable code and to determine the values of program
expressions. They apply their analysis to a language with asynchronous com-
munication. Their reachability analysis is based on an algorithm that builds
a spanning tree for each process flow graph and links matching transmit
and receive between processes. They construct a Monotone Framework for
determining value sets.

Intraprocedural control flow analysis. Many compilers trans-
form the source program into an intermediate form consisting of sequences
of fairly low-level instructions (like three address code) and then perform
the optimisations based on this representation. For this to work more informa-
tion is needed about the flow of control within the individual program
segments; in our terminology we need the flow (or flow%) relation in order
to apply the data flow analysis techniques. It is the task of intraprocedural
control flow analysis to provide this information.

More refined forms of intraprocedural control flow analysis include struc-
tural analysis [154, 110] that aims at discovering a wider variety of control
structures in the code - control structures like conditionals, while-loops and
repeat-loops that resemble those of the source language. The starting point
for structural analysis is the flow graph; it is examined to identify instances of
the various control structures, the instances are replaced by abstract nodes
and the connecting edges are collapsed; this process is repeated until the
flow graph has been collapsed into a single abstract node. The approach
described above is a refinement of classical techniques based on identifying
natural loops (or interval analysis [69, 5, 110]) intended to provide more
meaningful program structure.

We refer to the Concluding Remarks of Chapter 6 for a discussion of systems
implementing data flow analysers.
Mini Projects

Mini Project 2.1 ud- and du-chains

The aim of this mini project is to develop a more thorough understanding of the concepts of ud- and du-chains introduced in Subsection 2.1.5.

1. The function ud is specified in terms of definition clear paths, whilst UD re-uses functions introduced for the Reaching Definitions and Live Variables Analyses. Prove that the two functions compute the same information.

2. DU can be defined by analogy with UD. Starting from the definition of du, develop an equational definition of DU and verify its correctness.

3. A Constant Propagation Analysis is presented in Subsection 2.3.3; an alternative approach would be to use du- and ud-chains. Suppose there is a block \( x := n \) that assigns a constant \( n \) to a variable \( x \). Following the du-chain it is possible to find all blocks using the variable \( x \). It is only safe to replace a use of \( x \) by the constant \( n \) in a block \( t' \) if all other definitions that reach \( t' \) also assign the same constant \( n \) to \( x \). This can be determined by using the ud-chain. This is illustrated in Figure 2.23. Considering the program of Example 2.12, Constant Folding (followed by Dead Code Elimination) can be used to produce the following program:

\[
\text{if} [z=3] \text{ then } [z=0] \text{ else } [z=3] ; [y=3] ; [x=3+2] \]

Develop a formal description of this analysis.

Table 2.10: The instrumented semantics of While.

Mini Project 2.2 Correctness of Reaching Definitions

The aim of this mini project is to prove the correctness of Reaching Definitions with respect to the notion of semantic reaching definitions introduced in Section 1.5. To get a precise definition of the set of traces of interest we shall begin by introducing a so-called instrumented semantics: an extension of a more traditional semantics that keeps track of additional information that is mainly of interest for the program analysis.

The instrumented semantics has transitions of the forms:

\[
(S, \sigma, tr) \rightarrow (S', \sigma', tr') \quad \text{and} \quad (S, \sigma, tr) \rightarrow (S', \sigma', tr')
\]

All configurations include a trace \( tr \in \text{Trace} = (\text{Var} \times \text{Var})^* \) that records the elementary block in which a variable is being assigned. The detailed definition of the instrumented semantics is given in Table 2.10.

Given a program \( S_0 \) and an initial state \( \sigma_0 \in \text{State} \) it is natural to construct the trace

\[
tr_0 = ((x_1, ?), \ldots, (x_n, ?))
\]

where \( x_1, \ldots, x_n \) are the variables in \( S_0 \) and to consider the finite derivation
sequence:
\[(s_0, s_t, tr_x) \rightarrow^* (s_t', tr')\]

Intuitively, there should be a similar derivation sequence \((s_0, s_t, tr_x) \rightarrow^* (s_t', tr')\) in the Structural Operational Semantics. Similar remarks apply to infinite derivation sequences.

As in Section 2.2 we shall study the constraint system \(RD^c(S_i)\) corresponding to the equation system \(RD^e(S_i)\). Let \(reach\) be a collection of functions:

\[
reach_{entry}, reach_{exit} : Lab \rightarrow P(Var \times Lab)
\]

We say that \(reach\) solves \(RD^c(S_i)\), and write

\[reach \models RD^c(S_i)\]

if the functions satisfy the constraints; similarly for \(reach \models RD^e(S_i)\).

1. Formulate and prove results corresponding to Lemmas 2.15, 2.16 and 2.18.

The correctness relation \(\sim\) will relate traces \(tr \in \text{Trace}\) to the information obtained by the analysis. Let \(Y \subseteq P(Var \times Lab)\) and define

\[tr \sim Y \iff \forall x \in Var, (x, \text{SRD}(tr)(x)) \in Y\]

meaning that \(Y\) contains at least the semantically reaching definitions obtained from the trace \(tr\) by the function \(\text{SRD}\) introduced in Section 1.5.

2. Formulate and prove results corresponding to Lemma 2.20, Theorem 2.21 and Corollary 2.22.

**Mini Project 2.3 A Prototype Implementation**

In this mini project we shall implement one of the program analyses considered in Section 2.1. As an implementation language we shall choose a functional language such as Standard ML or Haskell. We can then define a suitable data type for WHILE programs as follows:

```
type var = string

type label = int

datatype aexp = String of var | Const of int |
| Op of string * aexp * aexp

and bexp = True | False |
| Not of bexp | Boolean of string * bexp * bexp |
| Relop of string * aexp * aexp

datatype stat = Assign of var * aexp * label | Skip of label |
| Seq of stat * stat |
| If of bexp * label * stat * stat |
| While of bexp * label * stat |
```

**Exercises**

**Exercise 2.1** Formulate data flow equations for the Reaching Definitions Analysis of the program studied in Example 1.1 of Chapter 1 and in particular define the appropriate \(\text{gen}\) and \(\text{kill}\) functions.

**Exercise 2.2** Consider the following program:

\[x := 1; (\text{while } y > 0) \{ x := x - 1 \}; x := 2\]

Perform a Live Variables Analysis for this program using the equations of Section 2.1.4.

**Exercise 2.3** A modification of the Available Expressions Analysis detects when an expression is available in a particular variable; a non-trivial expression \(e\) is available in \(x\) at a label \(l\) if it has been evaluated and assigned to \(x\) on all paths leading to \(l\) and if the values of \(x\) and the variables in the expression have not changed since then. Write down the data flow equations and any auxiliary functions for this analysis.

**Exercise 2.4** Consider the following program:

\[x := 1; x := x - 1; x := 2\]

Clearly \(x\) is dead at the exits from 2 and 3. But \(x\) is live at the exit of 1 even though its only use is to calculate a new value for a variable that turns out to be dead. We shall say that a variable is a faint variable if it is dead or if it is only used to calculate new values for faint variables; otherwise it is strongly live. In the example \(x\) is faint at the exits from 1, 2 and 3. Define a Data Flow Analysis that detects strongly live variables. (Hint: For an assignment \(x := a\) the definition of \(f_c(l)\) should be by cases on whether \(x\) is in \(l\) or not.)
Exercise 2.5 A basic block is often taken to be a maximal group of statements such that all transfers to the block are to the first statement in the group and, once the block has been entered, all statements in the group are executed sequentially. In this exercise we shall consider basic blocks of the form
\[ [x_1 := a_1; \ldots; x_n := a_n; b]^{f} \]
where \( n \geq 0 \) and \( b \) is \( x := a, \text{skip} \) or \( b \). Reformulate the analyses of Section 2.1 for this more general notion of basic block.

Exercise 2.6 Consider the analyses Available Expressions and Reaching Definitions. Which of the equations make sense for programs that do not have isolated entries (and how can this be improved)? Similarly, which of the equations for Very Busy Expressions and Live Variables make sense for programs that do not have isolated exits (and how can this be improved)? (Hint: See the beginning of Section 2.3.)

Exercise 2.7 Consider the correctness proof for the Live Variables Analysis in Section 2.2. Give a compositional definition of \( LV^{f}(\cdot) \) for a label consistent statement using
\[ LV^{f}(\text{skip}) = (LV_{\text{exit}}(f) = LV_{\text{entry}}(f)) \]
as one of the clauses and observe that a similar development is possible for \( LV^{f}(\cdot) \). Give a formal definition of \( \text{live} \models C \) where \( C \) is a set of equalities or inclusions as might have been produced by \( LV^{f}(S) \) or \( LV^{f}(S) \).

Prove that \( \{\text{live} \models LV^{f}(S)\} \) is a Moore family in the sense of Appendix A (with \( \uplus \) being \( \uplus \)) and determine whether or not a similar result holds for \( \{\text{live} \models LV^{f}(S)\} \).

Exercise 2.8 Show that Constant Propagation is a Monotone Framework with the set \( \mathcal{F}_{CP} \) as defined in Section 2.3.3.

Exercise 2.9 A Bit Vector Framework is a special instance of a Monotone Framework where
- \( L = (P(D), \subseteq) \) for some finite set \( D \) and where \( \subseteq \) is either \( \subseteq \) or \( \supseteq \), and
- \( \mathcal{F} = \{f : P(D) \rightarrow P(D) \mid \forall Y, Y' \subseteq D : Y' \subseteq D : f(Y) = (Y \cap Y') \cup Y'\} \)

Show that the four classical analyses of Section 2.1 are Bit Vector Frameworks. Show that all Bit Vector Frameworks are indeed Distributive Frameworks. Devise a Distributive Framework that is not also a Bit Vector Framework.

Exercise 2.10 Consider the Constant Propagation Analysis of Section 2.3.3 and the program
\[ (\text{if } \cdot \cdot \cdot \text{ then } [x := i; y := 1]; \text{else } [x := i; y := 1]; [x := x + y])^{f} \]
Show that \( \mathcal{MFP}_{c}(6) \) differs from \( \mathcal{MOP}_{c}(6) \).

Exercise 2.11 In our formulation of Monotone Frameworks we associate transfer functions with basic blocks. In a statement of the form
\[ \text{if } b \text{ then } S_1 \text{ else } S_2 \]
this prevents us from using the result of the test to pass different information to \( S_1 \) and \( S_2 \); as an example suppose that \( x \) is known to be positive or negative and that \( b \) is the test \( x > 0 \), then \( x \) is always positive at the entry to \( S_1 \) and always negative at the entry to \( S_2 \). To remedy this deficiency consider writing
\[ b^{f} \]
where \( b^{f} \) corresponds to \( b \) evaluating to true and \( b^{f} \) corresponds to \( b \) evaluating to false. Make the necessary changes to the development in Sections 2.1 and 2.3. (Begin by considering forward analyses.)

Exercise 2.12 Consider one of the analyses Available Expressions, Very Busy Expressions and Live Variables Analysis and perform a complexity analysis in the manner of Example 2.30.

Exercise 2.13 Let \( F \) be \( \text{flow}(S_{r}) \) and \( E \) be \( \{\text{init}(S_{r})\} \) for a label consistent program \( S_{r} \). Show that
\[ \forall f \in \text{Lab}_{\text{L}} : \text{path}_{f}(f) \neq \emptyset \]
Prove a similar result when \( F \) is \( \text{flow}^{0}(S_{r}) \) and \( E \) is \( \text{final}(S_{r}) \).

Exercise 2.14 In a Detection of Signs Analysis one models all negative numbers by the symbol \(-\), zero by the symbol \( 0 \), and all positive numbers by the symbol \(+\). As an example, the set \( \{-2, -1, 1\} \) is modelled by the set \( \{\ldots, -1, 0, +\} \), that is an element of the powerset \( P(\{-, 0, +\}) \).

Let \( S_{r} \) be a program and \( \text{Var}_{r} \) be the finite set of variables in \( S_{r} \). Take \( L_{t} \) to be \( \text{Var}_{r} \times \{-0, +\} \) and define an instance \( (L, F, F', E_{t}, f, f') \) of a Monotone Framework for performing Detection of Signs Analysis.

Similarly, take \( L_{t} \) to be \( \text{Var}_{r} \times \{-0, +\} \) and define an instance \( (L', F', E_{t}', f, f') \) of a Monotone Framework for Detection of Signs Analysis. Is there any difference in the precision obtained by the two approaches?
Exercise 2.15 In the previous exercise we defined a Detection of Signs Analysis that could not record the interdependencies between signs of variables (e.g. that two variables $x$ and $y$ always will have the same sign); this is sometimes called an independent attribute analysis. In this exercise we shall consider a variant of the analysis that is able to record the interdependencies between signs of variables; this is sometimes called a relational analysis. To do so take $L$ to be $P(\{x \in \text{Var} : \neg \{0, +\}\})$ and define an instance $(L, F, F_0, F_1, f)$ of a Monotone Framework for performing Detection of Signs Analysis. Construct an example showing that the result of this relational analysis may be more informative than that of the independent attribute analysis. The distinction between independent attribute methods and relational analysis is further discussed in Chapter 4.

Exercise 2.16 The interprocedural analysis using bounded call strings uses contexts to record the last $k$ call sites. Reformulate the analysis for a notion of context that records the last $k$ distinct call sites. Discuss whether or not this analysis is useful for distinguishing between the call of a procedure and subsequent recursive calls.

Exercise 2.17 Consider the Fibonacci program of Example 2.33 and the Detection of Signs Analysis of Exercise 2.15 and Example 2.36. Construct the data flow equations corresponding to using large and small assumption sets, respectively.

Exercise 2.18 Choose one of the four classical analyses from Section 2.1 and formulate it as an interprocedural analysis based on call strings. (Hint: Some may be easier than others.)

Exercise 2.19 Extend the syntax of programs to have the form

```
begin D; input x: S; output y end
```

so that it maps integers to integers rather than states to states. Consider the Detection of Signs Analysis and define the transfer functions for the input and output statements.

Exercise 2.20 Consider extending the procedure language such that procedures can have multiple call-by-value, call-by-result and call-by-value-result parameters as well as local variables and reconsider the Detection of Signs Analysis. How should one define the transfer functions associated with procedure call, procedure entry, procedure exit, and procedure return?

Exercise* 2.21 Compute the shape graphs $\text{Shape}_n(1), \ldots, \text{Shape}_n(7)$ of Example 2.47 using the information supplied in Examples 2.48, ..., 2.53. (Warning: there will be more than 50 shape graphs.)

Exercise 2.22 In the Shape Analysis of Section 2.6 work out direct definitions of the transfer functions for elementary statements of the forms

\[ \text{[z:=x,sel]}^f, [\text{z.sel}:=z]^f, [\text{z.sel}:=\text{z.sel}^f] \text{ and } [\text{malloc}(\text{z.sel})]^f \]
Exercise 1.6 Consider the Chaotic Iteration algorithm of Section 1.7 and suppose that
\[ \emptyset \subseteq R \subseteq F(R) \subseteq F^n(R) = F^{n+1}(\emptyset) \]
holds immediately before the assignment to RD; show that this also holds afterwards. (Hint: Write RD for \((R_D, R_D, \ldots, R_D, \ldots, R_D)\) and use the monotonicity of \(F\) and \(R \subseteq F(R)\) to establish that \(R \subseteq R_D \subseteq F(R_D) \subseteq F^n(R)\).)

Exercise 1.7 Use the Chaotic Iteration scheme of Section 1.7 to show that the information displayed in Table 1.1 is in fact the least fixed point of the function \(F\) defined in Section 1.3.

Exercise 1.8 Consider the following program
\[
\begin{align*}
[z := 1^2]; & \text{while } [x > 0^2] \text{ do } ([z := 2y^2]; [z := x - 1^4])
\end{align*}
\]
computing the \(x\)-th power of the number stored in \(y\). Formulate a system of data flow equations in the manner of Section 1.3. Next use the Chaotic Iteration strategy of Section 1.7 to compute the least solution and present it in a table (like Table 1.1).

Exercise 1.9 Perform Constant Folding upon the program
\[
\begin{align*}
[x := 10^2]; & \text{while } [x > 0^2] \text{ do } ([x := x + 1^2]; [z := x + z^2])
\end{align*}
\]
so as to obtain
\[
\begin{align*}
[x := 10^2]; & \text{while } [x > 0^2] \text{ do } ([y := 20^2]; [z := 30^2])
\end{align*}
\]
How many ways of obtaining the result are there?

Exercise 1.10 The specification of Constant Folding in Section 1.8 only considers arithmetic expressions. Extend it to deal also with boolean expressions. Consider adding axioms like
\[ RD \vdash [(x := a^e); S] \Rightarrow S \]
\[ RD \vdash [(\text{if } \text{true} \text{ then } S_1 \text{ else } S_2)] \Rightarrow S_1 \]
and discuss what complications arise.

Exercise 1.11 Consider adding the axiom
\[ RD \vdash [z := a^e] \Rightarrow [x := a[y \mapsto a]^e] \]
if
\[
\begin{cases}
  y \in \text{FV}(a) \land (y, a) \notin RD_{\text{entry}}(f) \\
  \forall (z, c') \in RD_{\text{entry}}(f) : (y = z \Rightarrow \vdash_1 \cdots \vdash_t) = [y := a]^c
\end{cases}
\]
to the specification of Constant Folding given in Section 1.8 and discuss whether or not this is a good idea.

---

Chapter 2

Data Flow Analysis

In this chapter we introduce techniques for Data Flow Analysis. Data Flow Analysis is the traditional form of program analysis which is described in many textbooks on compiler writing. We will present analyses for the simple imperative language WHILE that was introduced in Chapter 1. This includes a number of classical Data Flow Analyses: Available Expressions, Reaching Definitions, Very Busy Expressions and Live Variables. We introduce an operational semantics for WHILE and demonstrate the correctness of the Live Variables Analysis. We then present the notion of Monotone Frameworks and show how the examples may be recast as such frameworks. We continue by presenting a worklist algorithm for solving flow equations and we study its termination and correctness properties. The chapter concludes with a presentation of some advanced topics, including Interprocedural Data Flow Analysis and Shape Analysis.

Throughout the chapter we will clarify the distinctions between intra- and interprocedural analyses, between forward and backward analyses, between may and must analyses (or union and intersection analyses), between flow sensitive and flow insensitive analyses, and between context sensitive and context insensitive analyses.

2.1 Intra- and Interprocedural Analysis

In this section we present a number of example Data Flow Analyses for the WHILE language. The analyses are each defined by pairs of functions that map labels to the appropriate sets; one function in each pair specifies information that is true on entry to the block, the second specifies information that is true at the exit.