Chapter 1

Introduction

In this book we shall introduce four of the main approaches to program analysis: Data Flow Analysis, Constraint Based Analysis, Abstract Interpretation, and Type and Effect Systems. Each of Chapters 2 to 5 deals with one of these approaches at some length and generally treats the more advanced material in later sections. Throughout the book we aim at stressing the many similarities between what may at a first glance appear to be very unrelated approaches. To help to get this idea across, and to serve as a gentle introduction, this chapter treats all of the approaches at the level of examples. The technical details are worked out but it may be difficult to apply the techniques to related examples until some of the material of later chapters has been studied.

1.1 The Nature of Program Analysis

Program analysis offers static compile-time techniques for predicting safe and computable approximations to the set of values or behaviours arising dynamically at run-time when executing a program on a computer. A main application is to allow compilers to generate code avoiding ***redundant*** computations, e.g. by reusing available results or by moving loop invariant computations out of loops, or avoiding ***superfluous*** computations, e.g. of results known to be not needed or of results known already at compile-time. Among the more recent applications is the validation of software (possibly purchased from sub-contractors) to reduce the likelihood of malicious or unintended behaviour. Common for these applications is the need to combine information from different parts of the program.

A main aim of this book is to give an overview of a number of approaches to program analysis, all of which have a quite extensive literature, and to show

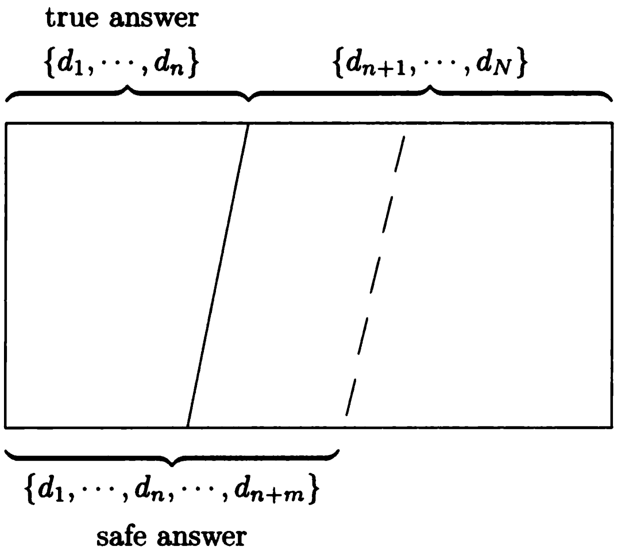
true answer
$\{d_1, \cdots, d_n\}$
$\{d_{n+1}, \cdots, d_N\}$
$\{d_1, \cdots, d_n, \cdots, d_{n+m}\}$
safe answer

Figure 1.1: The nature of approximation: erring on the safe side.

that there is a large amount of *commonality* among the approaches. This should help in cultivating the ability to choose the right approach for the right task and in exploiting insights developed in one approach to enhance the power of other approaches.

One common theme behind all approaches to program analysis is that in order to remain computable one can only provide *approximate answers.* As an example consider a simple language of statements and the program

```
read(x); (if x>0 then y:=1 else (y:=2;S)); z:=y
```

where S is some statement that does not contain an assignment to y. Intuitively, the values of y that can reach z:=y will be 1 or 2.

Now suppose an analysis claims that the only value for y that can reach z:=y is in fact 1. While this seems intuitively wrong, it is in fact correct in the case where S is known never to terminate for $x \leq 0$ and $y = 2$. But since it is *undecidable* whether or not S terminates, we normally do not expect our analysis to attempt to detect this situation. So in general, we expect the program analysis to produce a possibly *larger set* of possibilities than what will ever happen during execution of the program. This means that we shall also accept a program analysis claiming that the values of y reaching z:=y are among 1, 2 or 27, although we will clearly prefer the analysis that gives the more precise answer that the values are among 1 or 2. This notion of safe approximation is illustrated in Figure 1.1. Clearly the challenge is not to produce the safe "$\{d_1, \cdots, d_N\}$" too often as the analysis will then be utterly useless. Note, that although the analysis does not give precise information it may still give useful information: knowing that the value of y is one of 1, 2 and 27 just before the assignment `z:=y` still tells us that z will be positive, and that z will fit within 1 byte of storage etc. To avoid confusion it may help to be precise in the use of terminology: it is better to say "the values of y possible at `z:=y` are among 1 and 2" than the slightly shorter and more frequently used "the values of y possible at `z:=y` are 1 and 2".

Another common theme, to be stressed throughout this book, is that all program analyses should be *semantics based*: this means that the information obtained from the analysis can be proved to be safe (or correct) with respect to a semantics of the programming language. It is a sad fact that new program analyses often contain subtle bugs, and a formal justification of the program analysis will help finding these bugs sooner rather than later. However, we should stress that we do *not* suggest that program analyses be *semantics directed*: this would mean that the structure of the program analysis should reflect the structure of the semantics and this will be the case only for a few approaches which are not covered in this book.

1.2 Setting the Scene

Syntax of the WHILE language. We shall consider a simple imperative language called WHILE. A program in WHILE is just a statement which may be, and normally will be, a sequence of statements. In the interest of simplicity, we will associate data flow information with single assignment statements, the tests that appear in conditionals and loops, and **skip** statements. We will require a method to identify these. The most convenient way of doing this is to work with a labelled program – as indicated in the syntax below. We will often refer to the labelled items (assignments, tests and **skip** statements) as *elementary blocks*. In this chapter we will assume that distinct elementary blocks are initially assigned distinct labels; we could drop this requirement, in which case some of the examples would need to be slightly reformulated and the resultant analyses would be less accurate.

We use the following syntactic categories:

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

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

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

ℓ	$\mathsf{RD}_{entry}(\ell)$	$\mathsf{RD}_{exit}(\ell)$
1	(x,?),(y,?),(z,?)	(x,?),(y,1),(z,?)
2	(x,?),(y,1),(z,?)	(x,?),(y,1),(z,2)
3	(x,?),(y,1),(y,5),(z,2),(z,4)	(x,?),(y,1),(y,5),(z,2),(z,4)
4	(x,?),(y,1),(y,5),(z,2),(z,4)	(x,?),(y,1),(y,5),(z,4)
5	(x,?),(y,1),(y,5),(z,4)	(x,?),(y,5),(z,4)
6	(x,?),(y,1),(y,5),(z,2),(z,4)	(x,?),(y,6),(z,2),(z,4)

Table 1.1: Reaching Definitions information for the factorial program.

$$\begin{array}{lcll} op_a & \in & \mathbf{Op}_a & \text{arithmetic operators} \\ op_b & \in & \mathbf{Op}_b & \text{boolean operators} \\ op_r & \in & \mathbf{Op}_r & \text{relational operators} \end{array}$$

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

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

One way to think of the abstract syntax is as specifying the parse trees of the language; it will then be the purpose of the *concrete syntax* to provide sufficient information to enable unique parse trees to be constructed. In this book we shall *not* be concerned with concrete syntax: whenever we talk about some syntactic entity we will always be talking about the abstract syntax so there will be no ambiguity with respect to the form of the entity. We shall use a textual representation of the abstract syntax and to disambiguate it we shall use parentheses. For statements one often writes **begin** $\cdots$ **end** or $\{\cdots\}$ for this but we shall feel free to use $(\cdots)$. Similarly, we use brackets $(\cdots)$ to resolve ambiguities in other syntactic categories. To cut down on the number of brackets needed we shall use the familiar relative precedences of arithmetic, boolean and relational operators.

Example 1.1 An example of a program written in this language is the following which computes the factorial of the number stored in x and leaves the result in z:

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

Reaching Definitions Analysis. The use of distinct labels allows us to identify the primitive constructs of a program without explicitly constructing a flow graph (or flow chart). It also allows us to introduce a program analysis to be used throughout the chapter: *Reaching Definitions Analysis*, or as it should be called more properly, reaching assignments analysis:

> An assignment (called a definition in the classical literature) of the form $[x := a]^\ell$ *may reach* a certain program point (typically the entry or exit of an elementary block) if there is an execution of the program where x was last assigned a value at ℓ when the program point is reached.

Consider the factorial program of Example 1.1. Here $[\mathtt{y:=x}]^1$ reaches the entry to $[\mathtt{z:=1}]^2$; to allow a more succinct presentation we shall say that (y,1) reaches the entry to 2. Also we shall say that (x,?) reaches the entry to 2; here "?" is a special label not appearing in the program and it is used to record the possibility of an uninitialised variable reaching a certain program point.

Full information about reaching definitions for the factorial program is then given by the pair $\mathsf{RD} = (\mathsf{RD}_{entry}, \mathsf{RD}_{exit})$ of functions in Table 1.1. Careful inspection of this table reveals that the entry and exit information agree for elementary blocks of the form $[b]^\ell$ whereas for elementary blocks of the form $[x := a]^\ell$ they may differ on pairs (x, ℓ'). We shall come back to this when formulating the analysis in subsequent sections.

Returning to the discussion of safe approximation note that if we modify Table 1.1 to include the pair (z,2) in $\mathsf{RD}_{entry}(5)$ and $\mathsf{RD}_{exit}(5)$ we still have safe information about reaching definitions but the information is more approximate. However, if we remove (z,2) from $\mathsf{RD}_{entry}(6)$ and $\mathsf{RD}_{exit}(6)$ then the information will no longer be safe – there exists a run of the factorial program where the set {(x,?),(y,6),(z,4)} does not correctly describe the reaching definitions at the exit of label 6.

1.3 Data Flow Analysis

In *Data Flow Analysis* it is customary to think of a program as a graph: the nodes are the elementary blocks and the edges describe how control might pass from one elementary block to another. Figure 1.2 shows the flow graph for the factorial program of Example 1.1. We shall first illustrate the more common *equational approach* to Data Flow Analysis and then a *constraint based approach* that will serve as a stepping stone to Section 1.4.

1.3.1 The Equational Approach

The equation system. An analysis like Reaching Definitions can be specified by extracting a number of equations from a program. There are two classes of equations. One class of equations relate exit information of a node to entry information for the same node. For the factorial program

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

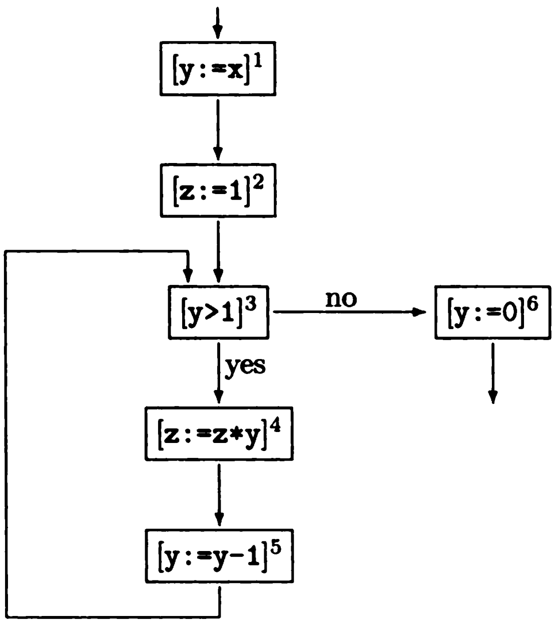

Figure 1.2: Flow graph for the factorial program.

we obtain the following six equations:

$$\begin{aligned}
\mathsf{RD}_{exit}(1) &= (\mathsf{RD}_{entry}(1)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathtt{y},1)\}\\
\mathsf{RD}_{exit}(2) &= (\mathsf{RD}_{entry}(2)\backslash\{(\mathtt{z},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathtt{z},2)\}\\
\mathsf{RD}_{exit}(3) &= \mathsf{RD}_{entry}(3)\\
\mathsf{RD}_{exit}(4) &= (\mathsf{RD}_{entry}(4)\backslash\{(\mathtt{z},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathtt{z},4)\}\\
\mathsf{RD}_{exit}(5) &= (\mathsf{RD}_{entry}(5)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathtt{y},5)\}\\
\mathsf{RD}_{exit}(6) &= (\mathsf{RD}_{entry}(6)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathtt{y},6)\}
\end{aligned}$$

These are instances of the following schema: for an assignment $[x := a]^{\ell'}$ we exclude all pairs (x,ℓ) from $\mathsf{RD}_{entry}(\ell')$ and add (x,ℓ') in order to obtain $\mathsf{RD}_{exit}(\ell')$ – this reflects that x is redefined at ℓ. For all other elementary blocks $[\cdots]^{\ell'}$ we let $\mathsf{RD}_{exit}(\ell')$ equal $\mathsf{RD}_{entry}(\ell')$ – reflecting that no variables are changed.

The other class of equations relate entry information of a node to exit information of nodes from which there is an edge to the node of interest; that is, entry information is obtained from all the exit information where control could have come from. For the example program we obtain the following equations:

$$\mathsf{RD}_{entry}(2) = \mathsf{RD}_{exit}(1)$$

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

In general, we write $\mathsf{RD}_{entry}(\ell) = \mathsf{RD}_{exit}(\ell_1) \cup \cdots \cup \mathsf{RD}_{exit}(\ell_n)$ if $\ell_1, \cdots, \ell_n$ are all the labels from which control might pass to ℓ. We shall consider more precise ways of explaining this in Chapter 2. Finally, let us consider the equation

$$\mathsf{RD}_{entry}(1) = \{(x, ?) \mid x \text{ is a variable in the program}\}$$

that makes it clear that the label "?" is to be used for uninitialised variables; so in our case

$$\mathsf{RD}_{entry}(1) = \{(\mathbf{x}, ?), (\mathbf{y}, ?), (\mathbf{z}, ?)\}$$

The least solution. The above system of equations defines the twelve sets

$$\mathsf{RD}_{entry}(1), \cdots, \mathsf{RD}_{exit}(6)$$

in terms of each other. Writing $\vec{\mathsf{RD}}$ for this twelve-tuple of sets we can regard the equation system as defining a function F and demanding that:

$$\vec{\mathsf{RD}} = F(\vec{\mathsf{RD}})$$

To be more specific we can write

$$F(\vec{\mathsf{RD}}) = (F_{entry}(1)(\vec{\mathsf{RD}}), F_{exit}(1)(\vec{\mathsf{RD}}), \cdots, F_{entry}(6)(\vec{\mathsf{RD}}), F_{exit}(6)(\vec{\mathsf{RD}}))$$

where e.g.:

$$F_{entry}(3)(\cdots, \mathsf{RD}_{exit}(2), \cdots, \mathsf{RD}_{exit}(5), \cdots) = \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(5)$$

It should be clear that F operates over twelve-tuples of sets of pairs of variables and labels; this can be written as

$$F : (\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12} \to (\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$$

where it might be natural to take $\mathbf{Var}_\star = \mathbf{Var}$ and $\mathbf{Lab}_\star = \mathbf{Lab}$. However, it will simplify the presentation in this chapter to let $\mathbf{Var}_\star$ be a *finite* subset of $\mathbf{Var}$ that contains the variables occurring in the program $S_\star$ of interest and similarly for $\mathbf{Lab}_\star$. So for the example program we might have $\mathbf{Var}_\star = \{\mathbf{x}, \mathbf{y}, \mathbf{z}\}$ and $\mathbf{Lab}_\star = \{1, \cdots, 6, ?\}$.

It is immediate that $(\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$ can be partially ordered by setting

$$\vec{\mathsf{RD}} \sqsubseteq \vec{\mathsf{RD}}' \quad \text{iff} \quad \forall i : \mathsf{RD}_i \subseteq \mathsf{RD}'_i$$

where $\vec{\mathsf{RD}} = (\mathsf{RD}_1, \cdots, \mathsf{RD}_{12})$ and similarly $\vec{\mathsf{RD}}' = (\mathsf{RD}'_1, \cdots, \mathsf{RD}'_{12})$. This turns $(\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$ into a complete lattice (see Appendix A) with least element

$$\vec{\emptyset} = (\emptyset, \cdots, \emptyset)$$

and binary least upper bounds given by:

$$\vec{\mathsf{RD}} \sqcup \vec{\mathsf{RD}}' = (\mathsf{RD}_1 \cup \mathsf{RD}'_1, \cdots, \mathsf{RD}_{12} \cup \mathsf{RD}'_{12})$$

It is easy to show that F is in fact a monotone function (see Appendix A) meaning that:

$$\vec{\mathsf{RD}} \sqsubseteq \vec{\mathsf{RD}}' \text{ implies } F(\vec{\mathsf{RD}}) \sqsubseteq F(\vec{\mathsf{RD}}')$$

This involves calculations like

$$\begin{array}{ll} & \mathsf{RD}_{exit}(2) \subseteq \mathsf{RD}'_{exit}(2) \text{ and } \mathsf{RD}_{exit}(5) \subseteq \mathsf{RD}'_{exit}(5) \\ \text{imply} & \\ & \mathsf{RD}_{exit}(2) \cup \mathsf{RD}_{exit}(5) \subseteq \mathsf{RD}'_{exit}(2) \cup \mathsf{RD}'_{exit}(5) \end{array}$$

and the details are left to the reader.

Consider the sequence $(F^n(\vec{\emptyset}))_n$ and note that $\vec{\emptyset} \sqsubseteq F(\vec{\emptyset})$. Since F is monotone, a straightforward mathematical induction (see Appendix B) gives that $F^n(\vec{\emptyset}) \sqsubseteq F^{n+1}(\vec{\emptyset})$ for all n. All the elements of the sequence will be in $(\mathcal{P}(\mathbf{Var}_\star \times \mathbf{Lab}_\star))^{12}$ and since this is a finite set it cannot be the case that all elements of the sequence are distinct so there must be some n such that:

$$F^{n+1}(\vec{\emptyset}) = F^n(\vec{\emptyset})$$

But since $F^{n+1}(\vec{\emptyset}) = F(F^n(\vec{\emptyset}))$ this just says that $F^n(\vec{\emptyset})$ is a *fixed point* of F and hence that $F^n(\vec{\emptyset})$ is a solution to the above equation system.

In fact we have obtained the *least solution* to the equation system. To see this suppose that $\vec{\mathsf{RD}}$ is some other solution, i.e. $\vec{\mathsf{RD}} = F(\vec{\mathsf{RD}})$. Then a straightforward mathematical induction shows that $F^n(\vec{\emptyset}) \sqsubseteq \vec{\mathsf{RD}}$. Hence the solution $F^n(\vec{\emptyset})$ contains the fewest pairs of reaching definitions that is consistent with the program, and intuitively, this is also the solution we want: while we can add additional pairs of reaching definitions without making the analysis semantically unsound, this will make the analysis less usable as discussed in Section 1.1. In Exercise 1.7 we shall see that the least solution is in fact the one displayed in Table 1.1.

1.3.2 The Constraint Based Approach

The constraint system. An alternative to the equational approach above is to use a *constraint based approach.* The idea is here to extract a number of inclusions (or inequations or constraints) out of a program. We shall present the constraint system for Reaching Definitions in such a way that the relationship to the equational approach becomes apparent; however, it is not a general phenomenon that the constraints are naturally divided into two classes as was the case for the equations.

For the factorial program

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

we obtain the following constraints for expressing the effect of elementary blocks:

$$\begin{array}{rcl}
\mathsf{RD}_{exit}(1) & \supseteq & \mathsf{RD}_{entry}(1)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(1) & \supseteq & \{(\mathtt{y},1)\} \\
\mathsf{RD}_{exit}(2) & \supseteq & \mathsf{RD}_{entry}(2)\backslash\{(\mathtt{z},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(2) & \supseteq & \{(\mathtt{z},2)\} \\
\mathsf{RD}_{exit}(3) & \supseteq & \mathsf{RD}_{entry}(3) \\
\mathsf{RD}_{exit}(4) & \supseteq & \mathsf{RD}_{entry}(4)\backslash\{(\mathtt{z},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(4) & \supseteq & \{(\mathtt{z},4)\} \\
\mathsf{RD}_{exit}(5) & \supseteq & \mathsf{RD}_{entry}(5)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(5) & \supseteq & \{(\mathtt{y},5)\} \\
\mathsf{RD}_{exit}(6) & \supseteq & \mathsf{RD}_{entry}(6)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\} \\
\mathsf{RD}_{exit}(6) & \supseteq & \{(\mathtt{y},6)\}
\end{array}$$

By considering this system a certain methodology emerges: for an assignment $[x := a]^{\ell'}$ we have one constraint that excludes all pairs (x, ℓ) from $\mathsf{RD}_{entry}(\ell')$ in reaching $\mathsf{RD}_{exit}(\ell')$ and we have one constraint for incorporating (x, ℓ'); for all other elementary blocks $[\cdots]^{\ell'}$ we just have one constraint that allows everything in $\mathsf{RD}_{entry}(\ell')$ to reach $\mathsf{RD}_{exit}(\ell')$.

Next consider the constraints for more directly expressing how control may flow through the program. For the example program we obtain the constraints:

$$\begin{array}{rcl}
\mathsf{RD}_{entry}(2) & \supseteq & \mathsf{RD}_{exit}(1) \\
\mathsf{RD}_{entry}(3) & \supseteq & \mathsf{RD}_{exit}(2) \\
\mathsf{RD}_{entry}(3) & \supseteq & \mathsf{RD}_{exit}(5) \\
\mathsf{RD}_{entry}(5) & \supseteq & \mathsf{RD}_{exit}(4) \\
\mathsf{RD}_{entry}(6) & \supseteq & \mathsf{RD}_{exit}(3)
\end{array}$$

In general, we have a constraint $\mathsf{RD}_{entry}(\ell) \supseteq \mathsf{RD}_{exit}(\ell')$ if it is possible for control to pass from ℓ' to ℓ. Finally, the constraint

$$\mathsf{RD}_{entry}(1) \supseteq \{(\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?)\}$$

records that we cannot be sure about the definition point of uninitialised variables.

The least solution revisited. It is not hard to see that a solution to the equation system presented previously will also be a solution to the above constraint system. To make this connection more transparent we can rearrange the constraints by *collecting* all constraints with the same left hand side. This means that for example

$$\begin{array}{rcl} \mathsf{RD}_{exit}(1) & \supseteq & \mathsf{RD}_{entry}(1)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\} \\ \mathsf{RD}_{exit}(1) & \supseteq & \{(\mathtt{y},1)\} \end{array}$$

will be replaced by

$$\mathsf{RD}_{exit}(1) \supseteq (\mathsf{RD}_{entry}(1)\backslash\{(\mathtt{y},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathtt{y},1)\}$$

and clearly this has no consequence for whether or not $\vec{\mathsf{RD}}$ is a solution. In other words we obtain a version of the previous equation system except that all equalities have been replaced by inclusions. Formally, whereas the equational approach demands that $\vec{\mathsf{RD}} = F(\vec{\mathsf{RD}})$, the constraint based approach demands that $\vec{\mathsf{RD}} \sqsupseteq F(\vec{\mathsf{RD}})$ for the *same* function F. It is therefore immediate that a solution to the equation system is also a solution to the constraint system whereas the converse is not necessarily the case.

Luckily we can show that both the equation system and the constraint system have the same *least solution.* Recall that the least solution to $\vec{\mathsf{RD}} = F(\vec{\mathsf{RD}})$ is constructed as $F^n(\vec{\emptyset})$ for a value of n such that $F^n(\vec{\emptyset}) = F^{n+1}(\vec{\emptyset})$. If $\vec{\mathsf{RD}}$ is a solution to the constraint system, that is $\vec{\mathsf{RD}} \sqsupseteq F(\vec{\mathsf{RD}})$, then $\vec{\emptyset} \sqsubseteq \vec{\mathsf{RD}}$ is immediate and the monotonicity of F and mathematical induction then gives $F^n(\vec{\emptyset}) \sqsubseteq \vec{\mathsf{RD}}$. Since $F^n(\vec{\emptyset})$ is a solution to the constraint system this shows that it is also the least solution to the constraint system.

In summary, we have thus seen a very strong connection between the equational approach and the constraint based approach. This connection is not always as apparent as it is here: one of the characteristics of the constraint based approach is that often constraints with the same left hand side are generated at many different places in the program and therefore it may require serious work to collect them.

1.4 Constraint Based Analysis

The purpose of *Control Flow Analysis* is to determine information about what "elementary blocks" may lead to what other "elementary blocks". This information is immediately available for the WHILE language unlike what is the case for more advanced imperative, functional and object-oriented languages. Often Control Flow Analysis is expressed as a Constraint Based Analysis as will be illustrated in this section.

Consider the following functional program:

```
let  f = fn x => x 1;
     g = fn y => y+2;
     h = fn z => z+3
in   (f g) + (f h)
```

It defines a higher-order function `f` with formal parameter `x` and body `x 1`; then it defines two functions `g` and `h` that are given as actual parameters to `f` in the body of the `let`-construct. Semantically, `x` will be bound to each of these two functions in turn so both `g` and `h` will be applied to `1` and the result of the computation will be the value 7.

An application of `f` will transfer control to the body of `f`, i.e. to `x 1`, and this application of `x` will transfer control to the body of `x`. The problem is that we cannot immediately point to the body of `x`: we need to know what parameters `f` will be called with. This is exactly the information that the Control Flow Analysis gives us:

> For each function application, which functions may be applied.

As is typical of functional languages, the labelling scheme used would seem to have a very different character than the one employed for imperative languages because the "elementary blocks" may be nested. We shall therefore label *all* subexpressions as in the following simple program that will be used to illustrate the analysis.

Example 1.2 Consider the program:

$$[[\texttt{fn x =>}\ [\mathrm{x}]^1]^2\ [\texttt{fn y =>}\ [\mathrm{y}]^3]^4]^5$$

It calls the identity function `fn x => x` on the argument `fn y => y` and clearly evaluates to `fn y => y` itself (omitting all $[\cdots]^\ell$). ■

We shall now be interested in associating information with the labels themselves, rather than with the entries and exits of the labels – thereby we exploit the fact that there are no side-effects in our simple functional language. The Control Flow Analysis will be specified by a pair $(\widehat{\mathsf{C}}, \widehat{\rho})$ of functions where $\widehat{\mathsf{C}}(\ell)$ is supposed to contain the values that the subexpression (or "elementary block") labelled ℓ may evaluate to and $\widehat{\rho}(x)$ contain the values that the variable x can be bound to.

The constraint system. One way to specify the Control Flow Analysis then is by means of a collection of constraints and we shall illustrate this for the program of Example 1.2. There are three classes of constraints. One class of constraints relate the values of function abstractions to their labels:

$$\{\texttt{fn x =>}\ [\mathrm{x}]^1\} \subseteq \widehat{\mathsf{C}}(2)$$
$$\{\texttt{fn y =>}\ [\mathrm{y}]^3\} \subseteq \widehat{\mathsf{C}}(4)$$

These constraints state that a function abstraction evaluates to a closure containing the abstraction itself. So the general pattern is that an occurrence of $[\texttt{fn}\ x \texttt{ => } e]^\ell$ in the program gives rise to a constraint $\{\texttt{fn}\ x \texttt{ => } e\} \subseteq \widehat{\mathsf{C}}(\ell)$.

The second class of constraints relate the values of variables to their labels:

$$\widehat{\rho}(\mathsf{x}) \subseteq \widehat{\mathsf{C}}(1)$$
$$\widehat{\rho}(\mathsf{y}) \subseteq \widehat{\mathsf{C}}(3)$$

The constraints state that a variable always evaluates to its value. So for each occurrence of $[x]^\ell$ in the program we will have a constraint $\widehat{\rho}(x) \subseteq \widehat{\mathsf{C}}(\ell)$.

The third class of constraints concerns function application: for each application point $[e_1\ e_2]^\ell$, and for each possible function $[\texttt{fn}\ x \texttt{ => } e]^{\ell'}$ that could be called at this point, we will have: (i) a constraint expressing that the formal parameter of the function is bound to the actual parameter at the application point, and (ii) a constraint expressing that the result obtained by evaluating the body of the function is a possible result of the application.

Our example program has just one application $[[\cdots]^2\ [\cdots]^4]^5$, but there are two candidates for the function, i.e. $\widehat{\mathsf{C}}(2)$ is a subset of the set $\{\texttt{fn x => } [\mathsf{x}]^1, \texttt{fn y => } [\mathsf{y}]^3\}$. If the function $\texttt{fn x => } [\mathsf{x}]^1$ is applied then the two constraints are $\widehat{\mathsf{C}}(4) \subseteq \widehat{\rho}(\mathsf{x})$ and $\widehat{\mathsf{C}}(1) \subseteq \widehat{\mathsf{C}}(5)$. We express this as *conditional constraints*:

$$\{\texttt{fn x => } [\mathsf{x}]^1\} \subseteq \widehat{\mathsf{C}}(2) \Rightarrow \widehat{\mathsf{C}}(4) \subseteq \widehat{\rho}(\mathsf{x})$$
$$\{\texttt{fn x => } [\mathsf{x}]^1\} \subseteq \widehat{\mathsf{C}}(2) \Rightarrow \widehat{\mathsf{C}}(1) \subseteq \widehat{\mathsf{C}}(5)$$

Alternatively, the function being applied could be $\texttt{fn y => } [\mathsf{y}]^3$ and the corresponding conditional constraints are:

$$\{\texttt{fn y => } [\mathsf{y}]^3\} \subseteq \widehat{\mathsf{C}}(2) \Rightarrow \widehat{\mathsf{C}}(4) \subseteq \widehat{\rho}(\mathsf{y})$$
$$\{\texttt{fn y => } [\mathsf{y}]^3\} \subseteq \widehat{\mathsf{C}}(2) \Rightarrow \widehat{\mathsf{C}}(3) \subseteq \widehat{\mathsf{C}}(5)$$

The least solution. As in Section 1.3 we shall be interested in the least solution to this set of constraints: the smaller the sets of values given by $\widehat{\mathsf{C}}$ and $\widehat{\rho}$, the more precise the analysis is in predicting which functions are applied. In Exercise 1.2 we show that the following choice of $\widehat{\mathsf{C}}$ and $\widehat{\rho}$ gives a solution to the above constraints:

$$\begin{array}{rcl}
\widehat{\mathsf{C}}(1) & = & \{\texttt{fn y => } [\mathsf{y}]^3\} \\
\widehat{\mathsf{C}}(2) & = & \{\texttt{fn x => } [\mathsf{x}]^1\} \\
\widehat{\mathsf{C}}(3) & = & \emptyset \\
\widehat{\mathsf{C}}(4) & = & \{\texttt{fn y => } [\mathsf{y}]^3\} \\
\widehat{\mathsf{C}}(5) & = & \{\texttt{fn y => } [\mathsf{y}]^3\} \\
\widehat{\rho}(\mathsf{x}) & = & \{\texttt{fn y => } [\mathsf{y}]^3\} \\
\widehat{\rho}(\mathsf{y}) & = & \emptyset
\end{array}$$

Among other things this tells us that the function abstraction `fn y => y` is never applied (since $\widehat{\rho}(\mathtt{y}) = \emptyset$) and that the program may only evaluate to the function abstraction `fn y => y` (since $\widehat{\mathsf{C}}(5) = \{\mathtt{fn\ y\ =>\ [y]^3}\}$).

Note the similarities between the constraint based approaches to Data Flow Analysis and Constraint Based Analysis: in both cases the syntactic structure of the program gives rise to a set of constraints whose least solution is desired. The main difference is that the constraints for the Constraint Based Analysis have a more complex structure than those for the Data Flow Analysis.

1.5 Abstract Interpretation

The theory of *Abstract Interpretation* is a general methodology for calculating analyses rather than just specifying them and then relying on a posteriori validation. To some extent the application of Abstract Interpretation is independent of the specification style used for presenting the program analysis and so applies not only to the Data Flow Analysis formulation to be used here.

Collecting semantics. As a preliminary step we shall formulate a so-called *collecting semantics* that records the set of *traces tr* that can reach a given program point:

$$tr \in \mathbf{Trace} = (\mathbf{Var} \times \mathbf{Lab})^*$$

Intuitively, a trace will record where the variables have obtained their values in the course of the computation. So for the factorial program

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

we will for example have the trace

$$((\mathtt{x},?),(\mathtt{y},?),(\mathtt{z},?),(\mathtt{y},1),(\mathtt{z},2),(\mathtt{z},4),(\mathtt{y},5),(\mathtt{z},4),(\mathtt{y},5),(\mathtt{y},6))$$

corresponding to a run of the program where the body of the **while**-loop is executed twice.

The traces contain sufficient information that we can extract a set of *semantically reaching definitions*:

$$\mathsf{SRD}(tr)(x) = \ell \quad \text{iff} \quad \text{the rightmost pair } (x,\ell') \text{ in } tr \text{ has } \ell = \ell'$$

We shall write $\mathsf{DOM}(tr)$ for the set of variables for which $\mathsf{SRD}(tr)$ is defined, i.e. $x \in \mathsf{DOM}(tr)$ iff some pair (x,ℓ) occurs in tr.

In order for the Reaching Definitions Analysis to be correct (or safe) we shall require that it captures the semantic reaching definitions, that is, if tr is a possible trace just before entering the elementary block labelled ℓ then we shall demand that

$$\forall x \in \mathsf{DOM}(tr) : (x, \mathsf{SRD}(tr)(x)) \in \mathsf{RD}_{entry}(\ell)$$

in order to trust the information in $\mathsf{RD}_{entry}(\ell)$ about the set of definitions that may reach the entry to ℓ. In later chapters, we will conduct proofs of results like this.

The collecting semantics will specify a *superset* of the possible traces at the various program points. We shall specify the collecting semantics CS in the style of the Reaching Definitions Analysis in Section 1.3; more precisely, we shall specify a twelve-tuple of elements from $(\mathcal{P}(\textbf{Trace}))^{12}$ by means of a set of equations. First we have

$$\begin{array}{rcl}
\mathsf{CS}_{exit}(1) & = & \{tr : (\mathtt{y}, 1) \mid tr \in \mathsf{CS}_{entry}(1)\} \\
\mathsf{CS}_{exit}(2) & = & \{tr : (\mathtt{z}, 2) \mid tr \in \mathsf{CS}_{entry}(2)\} \\
\mathsf{CS}_{exit}(3) & = & \mathsf{CS}_{entry}(3) \\
\mathsf{CS}_{exit}(4) & = & \{tr : (\mathtt{z}, 4) \mid tr \in \mathsf{CS}_{entry}(4)\} \\
\mathsf{CS}_{exit}(5) & = & \{tr : (\mathtt{y}, 5) \mid tr \in \mathsf{CS}_{entry}(5)\} \\
\mathsf{CS}_{exit}(6) & = & \{tr : (\mathtt{y}, 6) \mid tr \in \mathsf{CS}_{entry}(6)\}
\end{array}$$

showing how the assignment statements give rise to extensions of the traces. Here we write $tr : (x, \ell)$ for appending an element (x, ℓ) to a list tr, that is $((x_1, \ell_1), \cdots, (x_n, \ell_n)) : (x, \ell)$ equals $((x_1, \ell_1), \cdots, (x_n, \ell_n), (x, \ell))$. Furthermore, we have

$$\begin{array}{rcl}
\mathsf{CS}_{entry}(2) & = & \mathsf{CS}_{exit}(1) \\
\mathsf{CS}_{entry}(3) & = & \mathsf{CS}_{exit}(2) \cup \mathsf{CS}_{exit}(5) \\
\mathsf{CS}_{entry}(4) & = & \mathsf{CS}_{exit}(3) \\
\mathsf{CS}_{entry}(5) & = & \mathsf{CS}_{exit}(4) \\
\mathsf{CS}_{entry}(6) & = & \mathsf{CS}_{exit}(3)
\end{array}$$

corresponding to the flow of control in the program; more detailed information about the values of the variables would allow us to define the sets $\mathsf{CS}_{entry}(4)$ and $\mathsf{CS}_{entry}(6)$ more precisely but the above definitions are sufficient for illustrating the approach. Finally, we take

$$\mathsf{CS}_{entry}(1) \quad = \quad \{((\mathtt{x}, ?), (\mathtt{y}, ?), (\mathtt{z}, ?))\}$$

corresponding to the fact that all variables are uninitialised in the beginning.

In the manner of the previous sections we can rewrite the above system of equations in the form

$$\vec{\mathsf{CS}} \quad = \quad G(\vec{\mathsf{CS}})$$

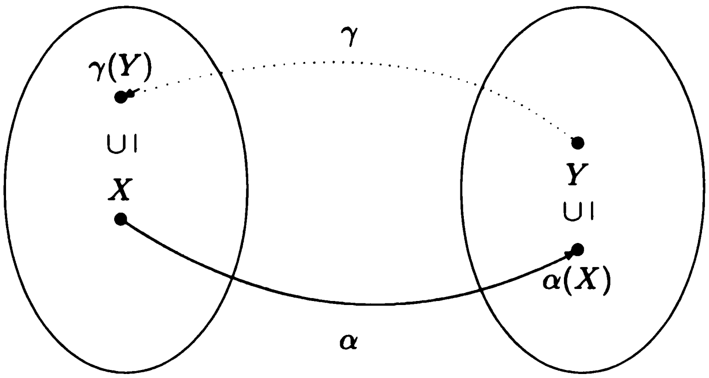

Figure 1.3: The adjunction (α, γ).

where $\overrightarrow{\mathsf{CS}}$ is a twelve-tuple of elements from $(\mathcal{P}(\textbf{Trace}))^{12}$ and where G is a monotone function of functionality:

$$G : (\mathcal{P}(\textbf{Trace}))^{12} \to (\mathcal{P}(\textbf{Trace}))^{12}$$

As is explained in Appendix A there is a body of general theory that ensures that the equation system in fact has a least solution; we shall write it as $\mathit{lfp}(G)$. However, since $(\mathcal{P}(\textbf{Trace}))^{12}$ is not finite we cannot simply use the methods of the previous sections in order to construct $\mathit{lfp}(G)$.

Galois connections. As we have seen the collecting semantics operates on sets of traces whereas the Reaching Definitions Analysis operates on sets of pairs of variables and labels. To relate these "worlds" we define an abstraction function α and a concretisation function γ as illustrated in:

$$\begin{tikzcd}
\mathcal{P}(\textbf{Trace}) \arrow[r, shift right, "\alpha"'] & \mathcal{P}(\textbf{Var} \times \textbf{Lab}) \arrow[l, shift right, "\gamma"']
\end{tikzcd}$$

The idea is that the *abstraction function* α extracts the reachability information present in a set of traces; it is natural to define

$$\alpha(X) = \{(x, \mathsf{SRD}(\mathit{tr})(x)) \mid x \in \mathsf{DOM}(\mathit{tr}) \ \wedge \mathit{tr} \ \in X\}$$

where we exploit the notion of semantically reaching definitions.

The *concretisation function* γ then produces all traces tr that are consistent with the given reachability information:

$$\gamma(Y) = \{\mathit{tr} \mid \forall x \in \mathsf{DOM}(\mathit{tr}) : (x, \mathsf{SRD}(\mathit{tr})(x)) \in Y\}$$

Often it is demanded that α and γ satisfy the condition

$$\alpha(X) \subseteq Y \Leftrightarrow X \subseteq \gamma(Y)$$

and we shall say that (α, γ) is an *adjunction*, or a *Galois connection*, whenever this condition is satisfied; this is illustrated in Figure 1.3. We shall leave it to the reader to verify that (α, γ) as defined above does in fact fulfil this condition.

Induced analysis. We shall now show how the collecting semantics can be used to *calculate* (as opposed to "guess") an analysis like the one in Section 1.3; we shall say that the analysis is an *induced analysis*. For this we define

$$\begin{array}{rcl} \vec{\alpha}(X_1, \cdots, X_{12}) & = & (\alpha(X_1), \cdots, \alpha(X_{12})) \\ \vec{\gamma}(Y_1, \cdots, Y_{12}) & = & (\gamma(Y_1), \cdots, \gamma(Y_{12})) \end{array}$$

where α and γ are as above and we consider the function $\vec{\alpha} \circ G \circ \vec{\gamma}$ of functionality:

$$(\vec{\alpha} \circ G \circ \vec{\gamma}) : (\mathcal{P}(\mathbf{Var} \times \mathbf{Lab}))^{12} \to (\mathcal{P}(\mathbf{Var} \times \mathbf{Lab}))^{12}$$

This function defines a Reaching Definitions analysis in an indirect way. Since G is specified by a set of equations (over $\mathcal{P}(\mathbf{Trace})$) we can use $\vec{\alpha} \circ G \circ \vec{\gamma}$ to calculate a new set of equations (over $\mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$). We shall illustrate this for one of the equations:

$$\mathsf{CS}_{exit}(4) = \{tr : (\mathbf{z}, 4) \mid tr \in \mathsf{CS}_{entry}(4)\}$$

The corresponding clause in the definition of G is:

$$G_{exit}(4)(\cdots, \mathsf{CS}_{entry}(4), \cdots) = \{tr : (\mathbf{z}, 4) \mid tr \in \mathsf{CS}_{entry}(4)\}$$

We can now calculate the corresponding clause in the definition of $\vec{\alpha} \circ G \circ \vec{\gamma}$:

$$\begin{array}{l}
\alpha(G_{exit}(4)(\vec{\gamma}(\cdots, \mathsf{RD}_{entry}(4), \cdots))) \\
\quad = \alpha(\{tr : (\mathbf{z}, 4) \mid tr \in \gamma(\mathsf{RD}_{entry}(4))\}) \\
\quad = \{(x, \mathsf{SRD}(tr : (\mathbf{z}, 4))(x)) \\
\qquad \mid x \in \mathsf{DOM}(tr : (\mathbf{z}, 4)), \\
\qquad\quad \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\
\quad = \{(x, \mathsf{SRD}(tr : (\mathbf{z}, 4))(x)) \\
\qquad \mid x \neq \mathbf{z},\ x \in \mathsf{DOM}(tr : (\mathbf{z}, 4)), \\
\qquad\quad \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\
\qquad \cup \{(x, \mathsf{SRD}(tr : (\mathbf{z}, 4))(x)) \\
\qquad \mid x = \mathbf{z},\ x \in \mathsf{DOM}(tr : (\mathbf{z}, 4)), \\
\qquad\quad \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\
\quad = \{(x, \mathsf{SRD}(tr)(x)) \\
\qquad \mid x \neq \mathbf{z},\ x \in \mathsf{DOM}(tr), \\
\qquad\quad \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\
\qquad \cup \{(\mathbf{z}, 4) \\
\qquad \mid \forall y \in \mathsf{DOM}(tr) : (y, \mathsf{SRD}(tr)(y)) \in \mathsf{RD}_{entry}(4)\} \\
\quad = (\mathsf{RD}_{entry}(4) \setminus \{(\mathbf{z}, \ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathbf{z}, 4)\}
\end{array}$$

The resulting equation

$$\mathsf{RD}_{exit}(4) = (\mathsf{RD}_{entry}(4) \setminus \{(\mathbf{z},\ell) \mid \ell \in \mathbf{Lab}\}) \cup \{(\mathbf{z},4)\}$$

is as in Section 1.3. Similar calculations can be performed for the other equations.

The least solution. As explained in Appendix A the equation system

$$\vec{\mathsf{RD}} \;=\; (\vec{\alpha} \circ G \circ \vec{\gamma})(\vec{\mathsf{RD}})$$

has a least solution; we shall write it as $\mathit{lfp}(\vec{\alpha} \circ G \circ \vec{\gamma})$. It is interesting to note that if one replaces the infinite sets **Var** and **Lab** with finite sets $\mathbf{Var}_\star$ and $\mathbf{Lab}_\star$ as before, then the least fixed point of $\vec{\alpha} \circ G \circ \vec{\gamma}$ can be obtained as $(\vec{\alpha} \circ G \circ \vec{\gamma})^n(\vec{\emptyset})$ just as was the case for F previously.

In Exercise 1.4 we shall show that $\vec{\alpha} \circ G \circ \vec{\gamma} \sqsubseteq F$ and that $\vec{\alpha}(G^n(\vec{\emptyset})) \sqsubseteq (\vec{\alpha} \circ G \circ \vec{\gamma})^n(\vec{\emptyset}) \sqsubseteq F^n(\vec{\emptyset})$ holds for all n. In fact it will be the case that

$$\vec{\alpha}(\mathit{lfp}(G)) \sqsubseteq \mathit{lfp}(\vec{\alpha} \circ G \circ \vec{\gamma}) \sqsubseteq \mathit{lfp}(F)$$

and this just says that the least solution to the equation system defined by $\vec{\alpha} \circ G \circ \vec{\gamma}$ is correct with respect to the collecting semantics, and similarly that the least solution to the equation system of Section 1.3 is also correct with respect to the collecting semantics. Thus it follows that we will only need to show that the collecting semantics is correct – the correctness of the induced analysis will follow for free.

For some analyses one is able to prove the stronger result $\vec{\alpha} \circ G \circ \vec{\gamma} = F$. Then the analysis is *optimal* (given the choice of approximate properties it operates on) and clearly $\mathit{lfp}(\vec{\alpha} \circ G \circ \vec{\gamma}) = \mathit{lfp}(F)$. In Exercise 1.4 we shall study whether or not this is the case here.

1.6 Type and Effect Systems

A simple type system. The ideal setting for explaining *Type and Effect Systems* is to consider a typed functional or imperative language. However, even our simple toy language can be considered to be typed: a statement S maps a state to a state (in case it terminates) and may therefore be considered to have type $\Sigma \to \Sigma$ where Σ denotes the type of states; we write this as the judgement:

$$S : \Sigma \to \Sigma$$

One way to formalise this is by the following utterly trivial system of axioms and inference rules: