So far our techniques have applied equally well to typed and untyped programming languages. This flexibility does not apply to the development to be performed in this chapter: here we demand that our programming language is typed because we will use the syntax for types in order to express the program analysis properties of interest (as was already illustrated in Section 1.6).

We shall first present an Annotated Type System for Control Flow Analysis in Section 5.1, demonstrate its semantic soundness and other theoretical properties in Section 5.2, and then in Section 5.3 show how to obtain an algorithm for computing the annotated types (and prove that it is sound and complete). In Sections 5.4 and 5.5 we give examples of other analyses specified by Type and Effect Systems. In Section 5.4 we study Type and Effect Systems with rules for subtyping, polymorphism and polymorphic recursion and illustrate their use in an analysis for tracking Side Effects, an Exception Analysis and an analysis for Region Inference. Finally, in Section 5.5 we show that the annotations can be given more structure and we illustrate this for a Communication Analysis.

5.1 Control Flow Analysis

Syntax of the FUN language. To illustrate the approach we shall make use of the functional language FUN also considered in Chapter 3; that the approach also applies to the imperative language of Chapter 2 was briefly sketched in Section 1.6. However, in this chapter we shall use a slightly different labelling scheme from the one in Chapter 3; the syntactic category of interest is

\[ e \in \text{Exp} \] expressions
and it is defined by:

$$e ::= c | x | \text{fun}_\pi x \Rightarrow e_0 | \text{fun}_\pi f x \Rightarrow e_0 | e_1 e_2$$

if $e_0$ then $e_1$ else $e_2$ | let $x = e_1$ in $e_2$ | $e_1$ op $e_2$

The program points, $\pi \in \text{Pnt}$, are used to name the function abstractions in the program; this could also be done using the notion of labelled terms from Chapter 3 but for our present purposes we do not need the full generality of this machinery – the reason is that now we will use the types to record information that was previously associated with labels. Hence our syntax just makes use of expressions and dispenses with terms.

As in the previous chapters we shall assume that a countable set of variables is given and that constants (including the truth values), binary operators (including the natural arithmetic, boolean and relational operators) and program points are left unspecified:

- $c \in \text{Const}$ constants
- $op \in \text{Op}$ binary operators
- $f, x \in \text{Var}$ variables
- $\pi \in \text{Pnt}$ program points

**Example 5.1** The functional program $(\text{fn } x \Rightarrow x) (\text{fn } y \Rightarrow y)$ considered in Chapters 1 and 3 is now written as

$$(\text{fn}_x x \Rightarrow x) (\text{fn}_y y \Rightarrow y)$$

just as we did in Example 1.5. ■

**Example 5.2** The expression loop of Example 3.2 is now written:

$$\begin{align*}
\text{let } g &= (\text{fun}_f f x \Rightarrow f (\text{fn}_y y \Rightarrow y)) \\
\text{in } g (\text{fn}_z z \Rightarrow z)
\end{align*}$$

Recall that this is a looping program: $g$ is first applied to the identity function $\text{fn}_z z \Rightarrow z$ but it ignores its argument and calls itself recursively with the function $\text{fn}_y y \Rightarrow y$. ■

### 5.1.1 The Underlying Type System

The analyses will be specified as extensions of the ordinary type system in order to record the program properties of interest. For this reason the ordinary type system is sometimes called the *underlying type system* and we shall start by specifying it.