Chapter 3

Constraint Based Analysis

In this chapter we present the technique of Constraint Based Analysis using a simple functional language, Fun. We begin by presenting an abstract specification of a Control Flow Analysis and then study its theoretical properties: it is correct with respect to a Structural Operational Semantics and it can be used to analyse all programs. This specification of the analysis does not immediately lend itself to an efficient algorithm for computing a solution so we proceed by developing first a syntax directed specification and then a constraint based formulation and finally we show how the constraints can be solved. We conclude by illustrating how the precision of the analysis can be improved by combining it with Data Flow Analysis and by incorporating context information thereby linking up with the development of the previous chapter.

3.1 Abstract 0-CFA Analysis

In Chapter 2 we saw how properties of data could be propagated through a program. In developing the specification we relied on the ability to identify for each program fragment all the possible successor (and predecessor) fragments via the operator flow (and flowR) and the interprocedural flow inter-flow*, (and inter-flow*R). The usefulness of the resulting specification was due to the number of successors and predecessors being small (usually just one or two except for procedure exits). This is a typical feature of imperative programs without procedures, but it usually fails for more general languages, whether imperative languages with procedures as parameters, functional languages, or object-oriented languages. In particular, the interprocedural techniques of Section 2.5 provide a solution for the simpler cases where the program text allows one to limit the number of successors, as is the case when a proce-
dure call is performed by explicitly mentioning the name of the procedure. However, these techniques are not powerful enough to handle the *dynamic dispatch problem* where variables can denote procedures. In Section 1.4 we illustrated this by the functional program

\[
\text{let } f = \text{fn } x \Rightarrow x + 1; \\
g = \text{fn } y \Rightarrow y + 2; \\
h = \text{fn } z \Rightarrow z + 3 \\
in (f \ g) + (f \ h)
\]

where the function application \( x + 1 \) in the body of \( f \) will transfer control to the body of the function \( x \), and here it is not so obvious what program fragment this actually is, since \( x \) is the formal parameter of \( f \). The Control Flow Analysis of the present chapter will provide a solution to the dynamic dispatch problem by determining for each subexpression a hopefully small number of functions that it may evaluate to; thereby it will determine where the flow of control may be transferred to in the case where the subexpression is the operator of a function application. In short, Control Flow Analysis will determine the *interprocedural flow* information (inter-flow, or IF) upon which the development of Section 2.5 is based.

**Syntax of the FUN language.** For the main part of this chapter we shall concentrate on a small functional language: the untyped lambda calculus extended with explicit operators for recursion, conditional and local definitions. The purpose of the Control Flow Analysis will be to compute for each subexpression the set of functions that it could evaluate to, and to express this it is important that we are able to label all program fragments. We shall be very explicit about this: a program fragment with a label is called an *expression* whereas a program fragment without a label is called a *term*. So we use the following syntactic categories:

\[
e \in \text{Exp} \quad \text{expressions (or labelled terms)} \\
t \in \text{Term} \quad \text{terms (or unlabelled expressions)}
\]

We assume that a countable set of variables is given and that constants (including the truth values), binary operators (including the usual arithmetic, boolean and relational operators) and labels are left unspecified:

\[
f, x \in \text{Var} \quad \text{variables} \\
c \in \text{Const} \quad \text{constants} \\
op \in \text{Op} \quad \text{binary operators} \\
l \in \text{Lab} \quad \text{labels}
\]

The abstract syntax of the language is now given by:

\[
e \ ::= \ t^l \\
t \ ::= \ c \ | \ x \ | \text{fn } x \Rightarrow e_0 \ | \text{fun } f \ x \Rightarrow e_0 \ | \ e_1 \ e_2 \\
| \ \text{if } e_0 \text{ then } e_1 \ \text{else } e_2 \ | \text{let } x = e_1 \ \text{in } e_2 \ | \ e_1 \ \text{op } e_2
\]