Chapter 7
Programming Languages

Alles ist eine Frage der Sprache. (Everything is a question of language.) — Ingeborg Bachmann (Alles)

In daily life, virtually all of human communication is expressed in one of the thousands of natural languages that are spoken world-wide; these languages are rich in their expressive capabilities, flexible in their applications, subtle in their nuances, and beautiful in their form. However, they are also full of gaps and ambiguities; while most of these can be usually overcome by intelligent beings that are able to deduce the intended interpretation from the context of the communication, they are from time to time are also the source of misunderstandings and disagreements, minor mishaps as well as major disasters. Thus, when communicating with ignorant partners such as computers, software developers use artificial languages that are designed in order to unambiguously express their intentions of how a computer program shall operate to solve a specific computational problem. However, even if millions of software developers use such programming languages every day, it is probably fair to say that only a minor fraction understands these languages in a sufficient depth to be able to answer subtle and critical questions about the behavior of the resulting programs. Ultimately, such an in-depth understanding requires a formal basis.

The goal of this chapter is to provide such a basis by showing how the semantics of programming languages can be precisely described in the language of logic, using the same kinds of techniques that have been introduced in the previous chapters for modeling "mathematical" languages. For this purpose, building upon the language of data types introduced in Chapter 6, Section 7.1 introduces an imperative programming language, i.e., a language whose core elements are commands that operate by reading from and writing to a common store. For this language we will give a formal type system; only well-typed programs will subsequently receive a semantics. Then Section 7.2 gives this language a "denotational" semantics that interprets commands as functions on stores; these functions are partial, i.e., may not return a result, which indicates that a program aborts or loops forever. Because partial functions are comparably inconvenient to deal with, we subsequently switch from a functional semantics to a relational one that allows arbitrarily many outcomes, which will also become useful in later chapters. Based on these results, we are able to prove the correctness of program transformations such as loop unrolling.

As an alternative framework, Section 7.3 introduces an "operational" semantics that models the programming language by a logical inference system whose judgements describe the transitions of a program from one state to another; here we differentiate between a "big-step" semantics that describes transitions from the initial to the final state and a "small-step" semantics that describes also the transitions between intermediate states. Each of these semantic forms (denotational semantics in functional and in relational flavor, operational semantics with big steps and with small steps) provides a different point of view on the same language; we formulate the precise relationships between the various forms and prove several equivalence results. As one application of semantics, Section 7.4 shows how to prove the correctness of a translation (compilation) of the high-level programming language considered so far to a formally modeled low-level machine language. Finally, Section 7.5 extends the programming language by procedures and discusses the formal modeling of core programming language concepts such as declaration scopes (static versus dynamic scoping), parameter passing mechanisms (value versus reference parameters), and recursive procedure definitions.

7.1 Programs and Commands

A Program Syntax

We start by defining the syntax of a simple imperative programming language. A program in this language operates by the execution of commands that read and write the values of variables. The program communicates with its environment by the values of certain given variables, the *parameters* of the program: the environment starts the program with some initial values of the parameters and, provided that the program terminates, observes their final values. For its computation, the program may also use some *local variables* which are, however, not seen by the environment.

Definition 7.1 (Program). A *program* is a phrase $P \in \mathit{Program}$ which is formed according to the following grammar:

$P \in \mathit{Program},\ X \in \mathit{Parameters},\ C \in \mathit{Command}$
$F \in \mathit{Formula},\ T \in \mathit{Term},\ V \in \mathit{Variable},\ S \in \mathit{Sort},\ I \in \mathit{Identifier}$

$$
\begin{array}{lcl}
P & ::= & \texttt{program}\ I(X)\ C \\
X & ::= & \sqcup \mid X, V\texttt{:}S \\
C & ::= & V\texttt{:=}T \mid \texttt{var}\ V\texttt{:}S\texttt{;}\ C \mid C_1\texttt{;}C_2 \mid \\
 & & \mid\ \texttt{if}\ F\ \texttt{then}\ C_1\ \texttt{else}\ C_2 \mid \texttt{if}\ F\ \texttt{then}\ C \mid \texttt{while}\ F\ \texttt{do}\ C
\end{array}
$$

The syntactic domains *Formula* of formulas, *Term* of terms, *Variable* of variables, *Sort* of sorts, and *Identifier* of identifiers, are formed as in Definition 6.1.

For readability, we use the syntax $\{C_1 ; C_2 ; \ldots ; C_n\}$ to denote the command $C_1 ; C_2'$, where C_2' denotes the command $C_2 ; C_3'$, where C_3' denotes the command $C_3 ; C_4'$, ..., and where finally C_{n-1}' denotes the command $C_{n-1} ; C_n$.

Example 7.1. As an example, take the following program *gcd* which computes the greatest common divisor g of two natural numbers a, b by the Euclidean algorithm:

```
program gcd(a:Nat,b:Nat,g:Nat,n:Nat) {
  var c:Nat; c := 0;
  while a > 0 ∧ b > 0 do {
    if a ≥ b
      then a := a-b
      else b := b-a
    c := c+1
  }
  if a > 0
    then g := a
    else g := b
  n := c
}
```

This program interacts with its environment by the parameters a, b, g, n. If a and b are initially not both zero, the program ultimately terminates with g set to the greatest common divisor of a and b; furthermore, it sets n to the number of loop iterations needed to compute that value. In the course of the computation, the program modifies both a and b such that one becomes zero and the other one the greatest common divisor; the program also uses a local variable c to keep track of the number of iterations. For instance, for the initial variable values $a = 12, b = 8$ (and g and n arbitrary), the program terminates with $a = 0$, $b = 4$, $g = 4$, and $n = 0$. For initial values $a = b = 0$, the program terminates with $a = b = g = n = 0$.

The program uses atomic formulas like $a > 0$ which denotes the application $gt(a, 0)$ of some predicate gt, terms like $a - b$ which denotes the application $minus(a, b)$ of some function *minus*, and terms like 0 which denotes the occurrence of some constant *zero*. □

Type Checking Programs

The well-formedness of a program is checked relative to a given signature Σ of sorts and operations that may be used in terms and formulas (see Section 6.2 for the essential concepts).

Definition 7.2 (Variable Typing). Let Σ be a signature. We define

$$VarTyping^{\Sigma} := Variable \to_{\bot} \Sigma.\mathrm{s}$$

as the set of all partial functions from variables to sorts in Σ; we call these functions *variable typings*.

Rules for $\Sigma \vdash P$: program:

$$\frac{\Sigma \vdash X : \text{parameters}(Vt, Vs) \quad \Sigma, Vt \vdash C : \text{command}}{\Sigma \vdash \texttt{program}\ I(X)\ C : \text{program}}$$

Rules for $\Sigma \vdash X$: parameters(Vt, Vs):

$$\Sigma \vdash \sqcup : \text{parameters}(\varnothing, [\,])$$

$$\frac{\Sigma \vdash X : \text{parameters}(Vt', Vs') \quad S \in \Sigma.\text{s} \quad \neg\exists S' \in Sort.\ \langle V, S' \rangle \in Vt' \quad Vt = Vt' \leftarrow \{\langle V, S \rangle\} \quad Vs = Vs' \circ [V]}{\Sigma \vdash X, V : S : \text{parameters}(Vt, Vs)}$$

Rules for $\Sigma, Vt \vdash C$: command:

$$\frac{\langle V, S \rangle \in Vt \quad \Sigma, Vt \vdash T : \text{term}(S)}{\Sigma, Vt \vdash V := T : \text{command}} \qquad \frac{S \in \Sigma.\text{s} \quad Vt_1 = Vt \leftarrow \{\langle V, S \rangle\} \quad \Sigma, Vt_1 \vdash C : \text{command}}{\Sigma, Vt \vdash \texttt{var}\ V : S\texttt{;}\ C : \text{command}}$$

$$\frac{\Sigma, Vt \vdash C_1 : \text{command} \quad \Sigma, Vt \vdash C_2 : \text{command}}{\Sigma, Vt \vdash C_1\,\texttt{;}\,C_2 : \text{command}}$$

$$\frac{\Sigma, Vt \vdash F : \text{formula} \quad \Sigma, Vt \vdash C_1 : \text{command} \quad \Sigma, Vt \vdash C_2 : \text{command}}{\Sigma, Vt \vdash \texttt{if}\ F\ \texttt{then}\ C_1\ \texttt{else}\ C_2 : \text{command}}$$

$$\frac{\Sigma, Vt \vdash F : \text{formula} \quad \Sigma, Vt \vdash C : \text{command}}{\Sigma, Vt \vdash \texttt{if}\ F\ \texttt{then}\ C : \text{command}} \qquad \frac{\Sigma, Vt \vdash F : \text{formula} \quad \Sigma, Vt \vdash C : \text{command}}{\Sigma, Vt \vdash \texttt{while}\ F\ \texttt{do}\ C : \text{command}}$$

Fig. 7.1 Type Checking Programs

Furthermore, given variable typings Vt_1 and Vt_2, we denote by $Vt_1 \leftarrow Vt_2$ the variable typing that arises from Vt_1 by adding all mappings from Vt_2 (overriding all conflicting mappings in Vt_1):

$$. \leftarrow . : \mathit{VarTyping} \to \mathit{VarTyping}$$
$$Vt_1 \leftarrow Vt_2 := (Vt_1 \setminus \{\langle V, S \rangle \mid \langle V, S \rangle \in Vt_1 \wedge \exists S' \in Sort.\ \langle V, S' \rangle \in Vt_2\}) \cup Vt_2$$

Now Figure 7.1 depicts a calculus for checking the type-correctness of a program. From this, the main judgement

$$\Sigma \vdash P : \text{program}$$

can be derived if program P is well-formed with respect to Σ. The auxiliary judgement

$$\Sigma \vdash X : \text{parameters}(Vt, Vs)$$

can be derived if the parameter list X gives rise to the variable typing $Vt \in \mathit{VarTyping}^{\Sigma}$ and to the sequence $Vs \in \mathit{Variable}^*$ of parameter names. The judgement

$$\Sigma, Vt \vdash C : \text{command}$$

can be derived if for the given variable typing Vt, command C is well-formed. Furthermore, the typing calculus refers to judgements for checking the well-formedness of formulas and terms; these have been introduced in Section 6.2.

The remainder of this chapter is dedicated to giving a meaning to programs that are well-formed according to this calculus.

7.2 A Denotational Semantics

Our goal is to give the programming language defined in the previous section a formal semantics (as we have done for various languages in the preceding chapters) by mapping every phrase of the program to a mathematical entity. This style of program semantics is also called *denotational semantics* (we will subsequently also deal with other kinds of program semantics).

Partial Functions

We will essentially define the meaning of a well-formed command, the core of a program, as a function from states to states, where a state is a function from program variables to values. However, since a command does not necessarily terminate, we will actually model states as *partial functions*, i.e., as binary relations that map an argument to at most one result (see also Section 5.7 where such functions were introduced). For instance, if a command C does not terminate on an initial state $s \in State$, the application of the partial function $[\![C]\!] : State^A \to_\perp State^A$ on s is not defined. Given a partial function $f : A \to_\perp B$, we therefore define

$$domain\, f \coloneqq \{x \in A \mid \exists y \in B.\ \langle x, y\rangle \in f\}$$

to denote the set of all arguments $x \in A$ for which the application of f yields a result $y \in B$.

However, since the domain $domain(f)$ of a partial function $f : A \to_\perp B$ is not necessarily identical to A, the unrestricted application of a partial function (which logically must denote a value) is problematic. We will therefore refrain from direct applications of f but restrict its use to formulas of the form

$$\mathsf{exists}\ y = f(x).\ F.$$

This notation was already introduced by Definition 5.13 as an abbreviation of

$$\exists y \in B.\ \langle x, y\rangle \in f \wedge F.$$

Here F is a subformula which may refer to the result $y \in B$ of the application of f to argument $x \in A$. The whole formula is "false" if there is no such value, i.e., f is not defined on x. Dually, we have defined the formula

$$\mathsf{forall}\ y = f(x).\ F.$$