15

The Simple Type System

Rudimentary type systems appeared in even the earliest higher-level programming languages, such as Fortran, Cobol, and Algol 60. Soon it was realized that such systems permitted a wide variety of programming mistakes to be detected during compilation, and provided information that could be used effectively to improve data representations. The consequence, beginning in the late 1960's, was a sequence of languages, such as PL/I, Algol 68, Pascal, and Ada, with extensive and complicated type systems.

Unfortunately, the type systems of these languages had serious deficiencies that limited their acceptance: Many of the languages contained "type leaks" that prevented the compile-time detection of certain type errors, many required so much type information that programs became unworkably verbose, and all imposed constraints that made certain useful programming techniques difficult or impossible.

Meanwhile, however, more theoretically-minded researchers began a systematic study of type systems, particularly for functional languages. Over the last two decades, they have shown that the application of sophisticated mathematical and logical techniques could lead to substantial improvements. Polymorphic systems have been devised that permit a procedure to be applied to a variety of types of parameters without sacrificing compile-time type checking. Type-inference algorithms have been developed that reduce or eliminate the need for explicit type information. (These two developments were central to the design of ML in the late 1970's.)

Other researchers, while exploring the connections between types and constructive logic, discovered that (at least in the setting of functional programming) the distinction between the type of a program and a full-blown logical specification of the program's behavior is one of degree rather than kind. This line of research led to the development of such systems as NuPrl and Coq that automatically extract a program from a mechanically verified proof (in intuitionistic logic) that its specification can be met. It also led to the Logical Framework, a metalogic for logical systems, including purely mathematical logics, logics for proving program

correctness, and type systems. Most important, perhaps, it led to the realization that specifications of interfaces between program modules are also instances of type systems; this realization led to the module systems of languages such as Standard ML.

At present there is a rich variety of type systems that are the subject of active research. In particular, many questions of how different systems are related, or how they might be unified, are still open research topics. Nevertheless, these systems all share a common starting point that is often called the *simple* type system. (It is essentially the simply typed lambda calculus, with appropriate extensions to deal with the additional features of the functional languages described in previous chapters.) We will begin our treatment of types by describing the simple system in this chapter, and then consider various extensions (largely in isolation from one another) in Section 15.7 and in later chapters.

Our exposition will be limited to purely functional languages. The discussion of syntactic issues (i.e. what are types and when do expressions satisfy typing judgements?) will be largely independent of evaluation order. However, the presentation of semantics (i.e. what do types mean?) will be limited to the direct denotational semantics of normal-order evaluation.

15.1 Types, Contexts, and Judgements

The types of the simple type system themselves constitute a small language, whose abstract syntax is

$$\begin{aligned}\langle \text{type} \rangle ::= \ & \mathbf{int} \mid \mathbf{bool} \mid \langle \text{type} \rangle \to \langle \text{type} \rangle \\ & \mid \mathbf{prod}(\langle \text{type} \rangle, \ldots, \langle \text{type} \rangle) \mid \mathbf{sum}(\langle \text{type} \rangle, \ldots, \langle \text{type} \rangle) \\ & \mid \mathbf{list}\ \langle \text{type} \rangle \mid \mathbf{cont}\ \langle \text{type} \rangle\end{aligned}$$

We will also use the following abbreviations:

$$\theta_0 \times \theta_1 \stackrel{\text{def}}{=} \mathbf{prod}(\theta_0, \theta_1)$$

$$\mathbf{unit} \stackrel{\text{def}}{=} \mathbf{prod}()$$

$$\theta_0 + \theta_1 \stackrel{\text{def}}{=} \mathbf{sum}(\theta_0, \theta_1).$$

In writing types, we will give operators the following precedences (in decreasing order):

$$(\mathbf{list}, \mathbf{cont}), \times, +, \to,$$

where $\times$ and $+$ are left-associative but $\to$ is right-associative. For example, $\mathbf{int} \to \mathbf{list\ int} \times \mathbf{bool} \to \mathbf{int}$ stands for $\mathbf{int} \to (((\mathbf{list\ int}) \times \mathbf{bool}) \to \mathbf{int})$.

We will give precise meanings to types in Sections 15.4 to 15.6, but in the meantime it is useful to give an informal reading:

- The *primitive type* **int** denotes the set of integers.
- The *primitive type* **bool** denotes the set of boolean values.
- The *function type* $\theta_0 \to \theta_1$ denotes the set of functions that return a value of type θ_1 when applied to a value of type θ_0.
- The *product* $\mathbf{prod}(\theta_0, \ldots, \theta_{n-1})$ denotes the set of tuples with n fields of types $\theta_0, \ldots, \theta_{n-1}$.
- The *sum* or *disjoint union* $\mathbf{sum}(\theta_0, \ldots, \theta_{n-1})$ denotes the set of alternative values with tags less than n, where a value tagged with k has type θ_k.
- **list** θ denotes the set of lists whose elements have type θ.
- **cont** θ denotes the set of continuations that accept values of type θ.

The next step is to introduce *contexts*, which are lists that assign types to patterns

$$\langle \text{context} \rangle ::= \ | \ \langle \text{context} \rangle, \langle \text{pat} \rangle : \langle \text{type} \rangle$$

where in $p_0 : \theta_0, \ldots, p_{n-1} : \theta_{n-1}$ the patterns $p_0, \ldots, p_{n-1}$ have no variables in common. (In simple cases, the patterns are merely variables.) Then, if π is a context, e is an expression, and θ is a type,

$$\pi \vdash e : \theta$$

is called a *typing judgement* and is read "e has type θ under π" or "π entails that e has type θ".

(The present use of the word *context* is unrelated to our earlier use, as in Section 2.8, to denote an expression with a hole in it where one can place various subexpressions. Some authors call π a *type assignment* or a *signature*, but these terms can also be confusing: A *type assignment* has nothing to do with the imperative operation of assignment, and the term *signature* usually denotes a specification of the types of constants or primitive operators rather than of variables or patterns.)

The following are examples of valid typing judgements (for the moment, the reader should ignore the underlining):

$$\begin{aligned}
\mathsf{m}:\underline{\mathbf{int}}, \mathsf{n}:\underline{\mathbf{int}} &\vdash \mathsf{m} + \mathsf{n} : \underline{\mathbf{int}} \\
\mathsf{f}:\mathbf{int} \to \mathbf{int}, \mathsf{x}:\mathbf{int} &\vdash \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int} \\
\mathsf{f}:\mathbf{int} \to \mathbf{int} &\vdash \lambda \mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int} \to \mathbf{int} \\
&\vdash \lambda \mathsf{f}.\ \lambda \mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int} \\
\mathsf{f}:\mathbf{bool} \to \mathbf{bool}, \mathsf{x}:\mathbf{bool} &\vdash \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{bool} \\
&\vdash \lambda \mathsf{f}.\ \lambda \mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : (\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool} \to \mathbf{bool} \\
\langle \mathsf{f}, \mathsf{x} \rangle : (\mathbf{int} \to \mathbf{int}) \times \mathbf{int} &\vdash \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int} \\
&\vdash \lambda \langle \mathsf{f}, \mathsf{x} \rangle.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : (\mathbf{int} \to \mathbf{int}) \times \mathbf{int} \to \mathbf{int}.
\end{aligned}$$

The fourth, sixth, and eighth lines displayed above are typing judgements with empty contexts, which make sense for closed expressions. Such judgements can also be given for the functions defined in Section 11.5:

$$\vdash \mathit{FACT} : \underline{\textbf{int}} \to \underline{\textbf{int}}$$

$$\vdash \mathit{ITERATE} : \underline{\textbf{int}} \to (\textbf{int} \to \textbf{int}) \to \textbf{int} \to \textbf{int}$$

$$\vdash \mathit{ITERATE} : \underline{\textbf{int}} \to (\textbf{bool} \to \textbf{bool}) \to \textbf{bool} \to \textbf{bool}$$

$$\vdash \mathit{APPEND} : \textbf{list int} \to \textbf{list int} \to \textbf{list int}$$

$$\vdash \mathit{MAPCAR} : (\textbf{int} \to \textbf{bool}) \to \textbf{list int} \to \textbf{list bool}$$

$$\vdash \mathit{FILTER} : \textbf{list int} \to (\textbf{int} \to \underline{\textbf{bool}}) \to \textbf{list int}$$

$$\vdash \mathit{EQLIST} : \textbf{list } \underline{\textbf{int}} \to \textbf{list } \underline{\textbf{int}} \to \underline{\textbf{bool}}$$

$$\vdash \mathit{MERGE} : \textbf{list } \underline{\textbf{int}} \to \textbf{list } \underline{\textbf{int}} \to \textbf{list } \underline{\textbf{int}}$$

$$\vdash \mathit{REDUCE} : \textbf{list int} \to (\textbf{int} \to \textbf{bool} \to \textbf{bool}) \to \textbf{bool} \to \textbf{bool}$$

$$\vdash \mathit{SEARCH} : \textbf{list int} \to (\textbf{int} \to \underline{\textbf{bool}}) \to (\textbf{int} \to \textbf{bool}) \to (\textbf{unit} \to \textbf{bool}) \to \textbf{bool}.$$

(Here each of the uppercase names abbreviates a closed expression for the appropriate function; for example, *FACT* abbreviates **letrec** fact $\equiv \lambda$n. **if** n $=$ 0 **then** 1 **else** n $\times$ fact(n $-$ 1) **in** fact.)

Notice that two judgements can give different types to the same expression, even under the same context. In fact, each of the above judgements would remain valid if one replaced all of the nonunderlined occurrences of **int** by some arbitrary type and all of the nonunderlined occurrences of **bool** by some (possibly different) arbitrary type. At the other extreme, there are *ill-typed* expressions, such as **true** $+$ x or x x, that do not satisfy any typing judgement.

15.2 Inference Rules

We use inference rules to specify which typing judgements are valid. Within these *type inference rules* (abbreviated TY RULE), we will use the following metavariables, sometimes with primes or subscripts:

π	⟨context⟩	v	⟨var⟩
θ	⟨type⟩	p	⟨pat⟩
e	⟨exp⟩	k, n	**N**.

The valid typing judgements of the simple type system are those that can be proven using the following rules. (We give rules for the constructs of both the eager-evaluation language of Chapter 11 and the normal-order language of Chapter 14, including constructs defined as syntactic sugar.)

TY RULE: Constants and Primitive Operations

$$\frac{}{\pi \vdash \mathbf{true} : \mathbf{bool}} \qquad \frac{}{\pi \vdash 0 : \mathbf{int}} \quad \text{etc.}$$

$$\frac{\pi \vdash e_0 : \mathbf{int} \qquad \pi \vdash e_1 : \mathbf{int}}{\pi \vdash e_0 + e_1 : \mathbf{int}} \quad \text{etc.}$$

$$\frac{\pi \vdash e_0 : \mathbf{int} \qquad \pi \vdash e_1 : \mathbf{int}}{\pi \vdash e_0 = e_1 : \mathbf{bool}} \quad \text{etc.} \tag{15.1}$$

$$\frac{\pi \vdash e_0 : \mathbf{bool} \qquad \pi \vdash e_1 : \mathbf{bool}}{\pi \vdash e_0 \wedge e_1 : \mathbf{bool}} \quad \text{etc.,}$$

TY RULE: Conditional Expressions

$$\frac{\pi \vdash e_0 : \mathbf{bool} \qquad \pi \vdash e_1 : \theta \qquad \pi \vdash e_2 : \theta}{\pi \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \theta,} \tag{15.2}$$

TY RULE: Variables

$$\frac{}{\pi \vdash v : \theta} \qquad \text{when } \pi \text{ contains } v{:}\theta, \tag{15.3}$$

TY RULE: $\rightarrow$ Introduction

$$\frac{\pi, p{:}\theta \vdash e : \theta'}{\pi \vdash \lambda p.\ e : \theta \rightarrow \theta',} \tag{15.4}$$

TY RULE: $\rightarrow$ Elimination

$$\frac{\pi \vdash e_0 : \theta \rightarrow \theta' \qquad \pi \vdash e_1 : \theta}{\pi \vdash e_0\, e_1 : \theta',} \tag{15.5}$$

TY RULE: Product Introduction

$$\frac{\pi \vdash e_0 : \theta_0 \quad \cdots \quad \pi \vdash e_{n-1} : \theta_{n-1}}{\pi \vdash \langle e_0, \ldots, e_{n-1} \rangle : \mathbf{prod}(\theta_0, \ldots, \theta_{n-1}),} \tag{15.6}$$

TY RULE: Product Elimination

$$\frac{\pi \vdash e : \mathbf{prod}(\theta_0, \ldots, \theta_{n-1})}{\pi \vdash e.k : \theta_k} \qquad \text{when } k < n, \tag{15.7}$$

TY RULE: Sum Introduction

$$\frac{\pi \vdash e : \theta_k}{\pi \vdash @\, k\, e : \mathbf{sum}(\theta_0, \ldots, \theta_{n-1})} \quad \text{when } k < n, \tag{15.8}$$

TY RULE: Sum Elimination

$$\frac{\begin{array}{c}\pi \vdash e : \mathbf{sum}(\theta_0, \ldots, \theta_{n-1}) \\ \pi \vdash e_0 : \theta_0 \to \theta \\ \qquad \ddots \\ \pi \vdash e_{n-1} : \theta_{n-1} \to \theta\end{array}}{\pi \vdash \mathbf{sumcase}\ e\ \mathbf{of}\ (e_0, \ldots, e_{n-1}) : \theta,} \tag{15.9}$$

TY RULE: Patterns

$$\frac{\pi, p_0{:}\,\theta_0, \ldots, p_{n-1}{:}\,\theta_{n-1}, \pi' \vdash e : \theta}{\pi, \langle p_0, \ldots, p_{n-1}\rangle{:}\,\mathbf{prod}(\theta_0, \ldots, \theta_{n-1}), \pi' \vdash e : \theta,} \tag{15.10}$$

TY RULE: **let** Definitions

$$\frac{\begin{array}{c}\pi \vdash e_0 : \theta_0 \\ \qquad \ddots \\ \pi \vdash e_{n-1} : \theta_{n-1} \\ \pi, p_0{:}\,\theta_0, \ldots, p_{n-1}{:}\,\theta_{n-1} \vdash e : \theta\end{array}}{\pi \vdash \mathbf{let}\ p_0 \equiv e_0, \ \ldots\ , p_{n-1} \equiv e_{n-1}\ \mathbf{in}\ e : \theta,} \tag{15.11}$$

TY RULE: **letrec** Definitions

$$\frac{\begin{array}{c}\pi, p_0{:}\,\theta_0, \ldots, p_{n-1}{:}\,\theta_{n-1} \vdash e_0 : \theta_0 \\ \qquad \ddots \\ \pi, p_0{:}\,\theta_0, \ldots, p_{n-1}{:}\,\theta_{n-1} \vdash e_{n-1} : \theta_{n-1} \\ \pi, p_0{:}\,\theta_0, \ldots, p_{n-1}{:}\,\theta_{n-1} \vdash e : \theta\end{array}}{\pi \vdash \mathbf{letrec}\ p_0 \equiv e_0, \ \ldots\ , p_{n-1} \equiv e_{n-1}\ \mathbf{in}\ e : \theta,} \tag{15.12}$$

TY RULE: Fixed Point

$$\frac{\pi \vdash e : \theta \to \theta}{\pi \vdash \mathbf{rec}\ e : \theta,} \tag{15.13}$$

TY RULE: List Introduction

$$\frac{}{\pi \vdash \mathbf{nil} : \mathbf{list}\ \theta} \qquad \frac{\pi \vdash e_0 : \theta \qquad \pi \vdash e_1 : \mathbf{list}\ \theta}{\pi \vdash e_0 :: e_1 : \mathbf{list}\ \theta,} \tag{15.14}$$

TY RULE: List Elimination

$$\frac{\pi \vdash e_0 : \mathbf{list}\ \theta \qquad \pi \vdash e_1 : \theta' \qquad \pi \vdash e_2 : \theta \to \mathbf{list}\ \theta \to \theta'}{\pi \vdash \mathbf{listcase}\ e_0\ \mathbf{of}\ (e_1, e_2) : \theta',} \tag{15.15}$$

TY RULE: **callcc**

$$\frac{\pi \vdash e : \mathbf{cont}\ \theta \to \theta}{\pi \vdash \mathbf{callcc}\ e : \theta,} \tag{15.16}$$

TY RULE: **throw**

$$\frac{\pi \vdash e_0 : \mathbf{cont}\ \theta \qquad \pi \vdash e_1 : \theta}{\pi \vdash \mathbf{throw}\ e_0\ e_1 : \theta',} \tag{15.17}$$

TY RULE: **error**

$$\frac{}{\pi \vdash \mathbf{error} : \theta.} \tag{15.18}$$

(The last rule captures the intention that the result of **error** should not be a type error. In contrast, the absence of any rule for **typeerror** captures the intention that the result of **typeerror** should always be a type error.)

The following is a simple example of the use of these rules in proving a typing judgement:

1.	$\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f} : \mathbf{int} \to \mathbf{int}$	(15.3)
2.	$\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{x} : \mathbf{int}$	(15.3)
3.	$\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f}\,\mathsf{x} : \mathbf{int}$	(15.5,1,2)
4.	$\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int}$	(15.5,1,3)
5.	$\mathsf{f}\colon \mathbf{int} \to \mathbf{int} \vdash \lambda\mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int} \to \mathbf{int}$	(15.4,4)
6.	$\vdash \lambda\mathsf{f}.\ \lambda\mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}$	(15.4,5)

A less trivial example is provided by the recursive definition of the function *append* given in Section 11.5. Let $\pi_1, \ldots, \pi_5$ abbreviate the following contexts:

$$\pi_1 = \mathsf{append}\colon \mathbf{list\ int} \to \mathbf{list\ int} \to \mathbf{list\ int}$$

$$\pi_2 = \pi_1, \mathsf{x}\colon \mathbf{list\ int}$$

$$\pi_3 = \pi_2, \mathsf{y}\colon \mathbf{list\ int}$$

$$\pi_4 = \pi_3, \mathsf{i}\colon \mathbf{int}$$

$$\pi_5 = \pi_4, \mathsf{r}\colon \mathbf{list\ int}.$$

Then the following is a proof of a typing judgement for an expression for the *append* function:

1. $\pi_5 \vdash \mathsf{append} : \mathbf{list\ int} \to \mathbf{list\ int} \to \mathbf{list\ int}$ (15.3)
2. $\pi_5 \vdash \mathsf{r} : \mathbf{list\ int}$ (15.3)
3. $\pi_5 \vdash \mathsf{append\ r} : \mathbf{list\ int} \to \mathbf{list\ int}$ (15.5,1,2)
4. $\pi_5 \vdash \mathsf{y} : \mathbf{list\ int}$ (15.3)
5. $\pi_5 \vdash \mathsf{append\ r\ y} : \mathbf{list\ int}$ (15.5,3,4)
6. $\pi_5 \vdash \mathsf{i} : \mathbf{int}$ (15.3)
7. $\pi_5 \vdash \mathsf{i} :: \mathsf{append\ r\ y} : \mathbf{list\ int}$ (15.14,6,5)
8. $\pi_4 \vdash \lambda\mathsf{r}.\ \mathsf{i} :: \mathsf{append\ r\ y} : \mathbf{list\ int} \to \mathbf{list\ int}$ (15.4,7)
9. $\pi_3 \vdash \lambda\mathsf{i}.\ \lambda\mathsf{r}.\ \mathsf{i} :: \mathsf{append\ r\ y} : \mathbf{int} \to \mathbf{list\ int} \to \mathbf{list\ int}$ (15.4,8)
10. $\pi_3 \vdash \mathsf{y} : \mathbf{list\ int}$ (15.3)
11. $\pi_3 \vdash \mathsf{x} : \mathbf{list\ int}$ (15.3)
12. $\pi_3 \vdash \mathbf{listcase}\ \mathsf{x}\ \mathbf{of}\ (\mathsf{y}, \lambda\mathsf{i}.\ \lambda\mathsf{r}.\ \mathsf{i} :: \mathsf{append\ r\ y}) : \mathbf{list\ int}$ (15.15,11,10,9)
13. $\pi_2 \vdash \lambda\mathsf{y}.\ \mathbf{listcase}\ \mathsf{x}\ \mathbf{of}\ (\mathsf{y}, \lambda\mathsf{i}.\ \lambda\mathsf{r}.\ \mathsf{i} :: \mathsf{append\ r\ y}) :$
    $\mathbf{list\ int} \to \mathbf{list\ int}$ (15.4,12)
14. $\pi_1 \vdash \lambda\mathsf{x}.\ \lambda\mathsf{y}.\ \mathbf{listcase}\ \mathsf{x}\ \mathbf{of}\ (\mathsf{y}, \lambda\mathsf{i}.\ \lambda\mathsf{r}.\ \mathsf{i} :: \mathsf{append\ r\ y}) :$
    $\mathbf{list\ int} \to \mathbf{list\ int} \to \mathbf{list\ int}$ (15.4,13)
15. $\pi_1 \vdash \mathsf{append} : \mathbf{list\ int} \to \mathbf{list\ int} \to \mathbf{list\ int}$ (15.3)
16. $\vdash \mathbf{letrec}\ \mathsf{append} \equiv \lambda\mathsf{x}.\ \lambda\mathsf{y}.$
    $\mathbf{listcase}\ \mathsf{x}\ \mathbf{of}\ (\mathsf{y}, \lambda\mathsf{i}.\ \lambda\mathsf{r}.\ \mathsf{i} :: \mathsf{append\ r\ y})\ \mathbf{in}\ \mathsf{append} :$
    $\mathbf{list\ int} \to \mathbf{list\ int} \to \mathbf{list\ int}$ (15.12,14,15)

A third example illustrates the use of products and compound patterns in typing a function that is similar to append except that it accepts a pair of lists rather than one list after another. Let $\pi_1, \ldots, \pi_5$ abbreviate the contexts:

$$\pi_1 = \mathsf{appendpr}{:}\ \mathbf{list\ int} \times \mathbf{list\ int} \to \mathbf{list\ int}$$

$$\pi_2 = \pi_1, \langle \mathsf{x}, \mathsf{y} \rangle{:}\ \mathbf{list\ int} \times \mathbf{list\ int}$$

$$\pi_3 = \pi_1, \mathsf{x}{:}\ \mathbf{list\ int}, \mathsf{y}{:}\ \mathbf{list\ int}$$

$$\pi_4 = \pi_3, \mathsf{i}{:}\ \mathbf{int}$$

$$\pi_5 = \pi_4, \mathsf{r}{:}\ \mathbf{list\ int}.$$

Then:

1. $\pi_5 \vdash \mathsf{appendpr} : \mathbf{list\ int} \times \mathbf{list\ int} \to \mathbf{list\ int}$ (15.3)
2. $\pi_5 \vdash \mathsf{r} : \mathbf{list\ int}$ (15.3)
3. $\pi_5 \vdash \mathsf{y} : \mathbf{list\ int}$ (15.3)
4. $\pi_5 \vdash \langle \mathsf{r}, \mathsf{y} \rangle : \mathbf{list\ int} \times \mathbf{list\ int}$ (15.6,2,3)
5. $\pi_5 \vdash \mathsf{appendpr}\langle \mathsf{r}, \mathsf{y} \rangle : \mathbf{list\ int}$ (15.5,1,4)
6. $\pi_5 \vdash \mathsf{i} : \mathbf{int}$ (15.3)
7. $\pi_5 \vdash \mathsf{i} :: \mathsf{appendpr}\langle \mathsf{r}, \mathsf{y} \rangle : \mathbf{list\ int}$ (15.14,6,5)
8. $\pi_4 \vdash \lambda \mathsf{r}.\ \mathsf{i} :: \mathsf{appendpr}\langle \mathsf{r}, \mathsf{y} \rangle : \mathbf{list\ int} \to \mathbf{list\ int}$ (15.4,7)
9. $\pi_3 \vdash \lambda \mathsf{i}.\ \lambda \mathsf{r}.\ \mathsf{i} :: \mathsf{appendpr}\langle \mathsf{r}, \mathsf{y} \rangle : \mathbf{int} \to \mathbf{list\ int} \to \mathbf{list\ int}$ (15.4,8)
10. $\pi_3 \vdash \mathsf{y} : \mathbf{list\ int}$ (15.3)
11. $\pi_3 \vdash \mathsf{x} : \mathbf{list\ int}$ (15.3)
12. $\pi_3 \vdash \mathbf{listcase}\ \mathsf{x}\ \mathbf{of}\ (\mathsf{y}, \lambda \mathsf{i}.\ \lambda \mathsf{r}.\ \mathsf{i} :: \mathsf{appendpr}\langle \mathsf{r}, \mathsf{y} \rangle) : \mathbf{list\ int}$ (15.15,11,10,8)
13. $\pi_2 \vdash \mathbf{listcase}\ \mathsf{x}\ \mathbf{of}\ (\mathsf{y}, \lambda \mathsf{i}.\ \lambda \mathsf{r}.\ \mathsf{i} :: \mathsf{appendpr}\langle \mathsf{r}, \mathsf{y} \rangle) : \mathbf{list\ int}$ (15.10,12)
14. $\pi_1 \vdash \lambda \langle \mathsf{x}, \mathsf{y} \rangle.\ \mathbf{listcase}\ \mathsf{x}\ \mathbf{of}\ (\mathsf{y}, \lambda \mathsf{i}.\ \lambda \mathsf{r}.\ \mathsf{i} :: \mathsf{appendpr}\langle \mathsf{r}, \mathsf{y} \rangle) :$
    $\mathbf{list\ int} \times \mathbf{list\ int} \to \mathbf{list\ int}$ (15.4,13)
15. $\pi_1 \vdash \mathsf{appendpr} : \mathbf{list\ int} \times \mathbf{list\ int} \to \mathbf{list\ int}$ (15.3)
16. $\vdash \mathbf{letrec}\ \mathsf{appendpr} \equiv \lambda \langle \mathsf{x}, \mathsf{y} \rangle.$
    $\mathbf{listcase}\ \mathsf{x}\ \mathbf{of}\ (\mathsf{y}, \lambda \mathsf{i}.\ \lambda \mathsf{r}.\ \mathsf{i} :: \mathsf{appendpr}\langle \mathsf{r}, \mathsf{y} \rangle)\ \mathbf{in}\ \mathsf{appendpr} :$
    $\mathbf{list\ int} \times \mathbf{list\ int} \to \mathbf{list\ int}$ (15.12,14,15)

It should be noted that, when using rules 15.4, 15.11, and 15.12, which deal with binding constructs, one must sometimes rename bound variables after applying the rule, to get around the restriction that patterns in contexts must have disjoint variables, For example, there is no instance of 15.4 whose conclusion is $\mathsf{x}\colon \mathbf{bool} \vdash \lambda \mathsf{x}.\ \mathsf{x} : \mathbf{int} \to \mathbf{int}$, since the corresponding premiss, $\mathsf{x}\colon \mathbf{bool}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{x} : \mathbf{int}$, contains an illegal context. But one can use the rule to prove $\mathsf{x}\colon \mathbf{bool} \vdash \lambda \mathsf{x}'.\ \mathsf{x}' : \mathbf{int} \to \mathbf{int}$ and then rename x' in the lambda expression.

Finally, it should be emphasized that the introduction of simple types changes the status of the constructions **letrec** and **rec** for recursive definition and fixed points. As mentioned in Section 14.4, in a untyped language using normal-order evaluation, these constructions can be defined as syntactic sugar by using the expression Y that maps functions into their least fixed-points. (One can also desugar **letrec** in untyped eager-evaluation languages.) Such definitions, however, are ill-typed in most type systems. In fact, it has been shown for the simple type system that, in the absence of any explicit construction for recursion, the

15.5 The Intrinsic View

We now turn to the *intrinsic* view of the semantics of types, which is associated with the logician Alonzo Church and is sometimes called the "ontological" view.

In the extrinsic view of the previous section, the meaning of an expression is independent of the type system, while the meaning of a type is a partial equivalence relation whose domain is the set of values of expressions of the type. For example, the meaning of $\lambda x.\ x$ is the identity function on V_*, which belongs to the domains of the meanings of both $\mathbf{int} \to \mathbf{int}$ and $\mathbf{bool} \to \mathbf{bool}$ (since, roughly speaking, it maps integers into integers and booleans into booleans).

In contrast, in the *intrinsic* view, an expression only has a meaning when it satisfies a typing judgement, the kind of meaning depends on the judgement, and an expression satisfying several judgements will have several meanings. For example, corresponding to the judgement $\vdash \lambda x.\ x : \mathbf{int} \to \mathbf{int}$ the value of $\lambda x.\ x$ is the identity function on the domain that is the meaning of **int**, while corresponding to the judgement $\vdash \lambda x.\ x : \mathbf{bool} \to \mathbf{bool}$ the value of $\lambda x.\ x$ is the identity function on the domain that is the meaning of **bool**. On the other hand, ill-typed expressions such as $\mathbf{true} + x$ or $x\,x$ have no meaning at all.

As in the previous section, to keep things simple we limit our semantic description to normal-order evaluation; we disregard lists, **let**, **letrec**, and compound patterns; and we give nontype errors the same meaning as nontermination.

The first step in developing an intrinsic semantics is to define the meaning of each type θ to be a domain, which we will denote by $\mathcal{D}(\theta)$. The function $\mathcal{D}$ is defined by syntax-directed equations on the syntax of types. For primitive types we have the obvious flat domains

$$\mathcal{D}(\mathbf{int}) = \mathbf{Z}_\perp \qquad \mathcal{D}(\mathbf{bool}) = \mathbf{B}_\perp,$$

while for compound types we express each type constructor in terms of the analogous constructor for domains:

$$\mathcal{D}(\theta \to \theta') = \mathcal{D}(\theta) \to \mathcal{D}(\theta')$$

$$\mathcal{D}(\mathbf{prod}(\theta_0, \ldots, \theta_{n-1})) = \mathcal{D}(\theta_0) \times \cdots \times \mathcal{D}(\theta_{n-1})$$

$$\mathcal{D}(\mathbf{sum}(\theta_0, \ldots, \theta_{n-1})) = (\mathcal{D}(\theta_0) + \cdots + \mathcal{D}(\theta_{n-1}))_\perp.$$

(Notice the use of lifting in the last equation to convert a predomain into a domain. A lifted disjoint union is often called a "separated sum" of domains.)

The next step is to define the meaning of a context π to be a domain, which we will denote by $\mathcal{D}^*(\pi)$. To do this, we regard the context $\pi = v_0{:}\,\theta_0, \ldots, v_{n-1}{:}\,\theta_{n-1}$ as the function on the finite set $\{v_0, \ldots, v_{n-1}\}$ that maps each v_i into the type θ_i. Then the meaning of π is the iterated product, over $\{v_0, \ldots, v_{n-1}\}$, of the domains that are the meanings of the θ_i:

$$\mathcal{D}^*(\pi) = \prod_{v \in \mathrm{dom}\,\pi} \mathcal{D}(\pi\, v).$$

As with all products of domains, $\mathcal{D}^*(\pi)$ is ordered componentwise. Its members are environments that act on finite sets of variables (rather than the environments used previously in this book, which are defined for all variables). For example, $[\mathsf{x}\colon 7 \mid \mathsf{y}\colon \mathbf{true} \mid \mathsf{z}\colon \lambda x\colon \mathbf{Z}_\perp.\ x]$ is an environment in the domain $\mathcal{D}^*(\mathsf{x}\colon \mathbf{int}, \mathsf{y}\colon \mathbf{bool}, \mathsf{z}\colon \mathbf{int} \to \mathbf{int})$.

Finally, we consider the intrinsic semantics of expressions, which is fundamentally different from either untyped or extrinsic typed semantics. When we say that an expression has a meaning for every typing judgement that it satisfies, we really mean that meanings are attached to judgements rather than to expressions. For every valid typing judgement $\pi \vdash e : \theta$ there is a meaning

$$[\![\pi \vdash e : \theta]\!] \in \mathcal{D}^*(\pi) \to \mathcal{D}(\theta)$$

that is a continuous function from environments in the meaning of π to values in the meaning of θ. One can think of this as the meaning of e corresponding to $\pi \vdash e : \theta$, but fundamentally it is just the meaning of $\pi \vdash e : \theta$ itself. For example (where $[\,]$ denotes the empty environment),

$$[\![\vdash \lambda \mathsf{x}.\ \mathsf{x} : \mathbf{int} \to \mathbf{int}]\!][\,] = \lambda z \in \mathbf{Z}_\perp.\ z$$

$$[\![\vdash \lambda \mathsf{x}.\ \mathsf{x} : \mathbf{bool} \to \mathbf{bool}]\!][\,] = \lambda z \in \mathbf{B}_\perp.\ z.$$

When we attach meaning to judgements rather than expressions, however, it no longer makes sense to define the meaning by syntax-directed semantic equations, which are based on structural induction over expressions. Instead, we define the meaning of judgements by structural induction on the proofs (viewed as trees) of judgements.

For each type inference rule, we give a typed semantic equation that, for every instance of the rule, expresses the meaning of the conclusion of the instance in terms of the meanings of the premisses of the instance. Within these semantic equations, we will use the metavariables:

θ	⟨type⟩	z	values in some $\mathcal{D}(\theta)$
π	⟨context⟩	η	environments in some $\mathcal{D}^*(\pi)$
e	⟨exp⟩	k, n	$\mathbf{N}$
v	⟨var⟩	i	$\mathbf{Z}$
		b	$\mathbf{B}$.

In the typed semantic equations themselves (abbreviated TY SEM EQ), we indicate the correspondence with the type inference rules in Section 15.2 by parenthesized references. (Notice that the equation for $\wedge$ defines short-circuit evaluation.)

TY SEM EQ: Constants and Primitive Operations (15.1)

$$[\![\pi \vdash \mathbf{true}\colon \mathbf{bool}]\!]\eta = \iota_\uparrow \mathbf{true}$$

$$[\![\pi \vdash 0\colon \mathbf{int}]\!]\eta = \iota_\uparrow 0$$

$$[\![\pi \vdash e_0 + e_1\colon \mathbf{int}]\!]\eta = (\lambda i.\ (\lambda i'.\ \iota_\uparrow(i + i'))_{\perp\!\!\perp}([\![\pi \vdash e_1\colon \mathbf{int}]\!]\eta))_{\perp\!\!\perp}([\![\pi \vdash e_0\colon \mathbf{int}]\!]\eta)$$

$$[\![\pi \vdash e_0 = e_1\colon \mathbf{bool}]\!]\eta = (\lambda i.\ (\lambda i'.\ \iota_\uparrow(i = i'))_{\perp\!\!\perp}([\![\pi \vdash e_1\colon \mathbf{int}]\!]\eta))_{\perp\!\!\perp}([\![\pi \vdash e_0\colon \mathbf{int}]\!]\eta)$$

$$[\![\pi \vdash e_0 \wedge e_1\colon \mathbf{bool}]\!]\eta = (\lambda b.\ \mathbf{if}\ b\ \mathbf{then}\ [\![\pi \vdash e_1\colon \mathbf{bool}]\!]\eta\ \mathbf{else}\ \iota_\uparrow \mathbf{false})_{\perp\!\!\perp}([\![\pi \vdash e_0\colon \mathbf{bool}]\!]\eta),$$

TY SEM EQ: Conditional Expressions (15.2)

$$[\![\pi \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \theta]\!]\eta = (\lambda b.\ \mathbf{if}\ b\ \mathbf{then}\ [\![\pi \vdash e_1 : \theta]\!]\eta\ \mathbf{else}\ [\![\pi \vdash e_2 : \theta]\!]\eta)_{\perp\!\!\perp}([\![\pi \vdash e_0 : \mathbf{bool}]\!]\eta),$$

TY SEM EQ: Variables (15.3)

$$[\![\pi \vdash v : \theta]\!]\eta = \eta\, v \qquad \text{when } \pi \text{ contains } v\colon\theta,$$

TY SEM EQ: $\rightarrow$ Introduction (15.4)

$$[\![\pi \vdash \lambda v.\ e : \theta \rightarrow \theta']\!]\eta = \lambda z \in \mathcal{D}(\theta).\ [\![\pi, v\colon\theta \vdash e : \theta']\!][\,\eta \mid v\colon z\,],$$

TY SEM EQ: $\rightarrow$ Elimination (15.5)

$$[\![\pi \vdash e_0\, e_1 : \theta']\!]\eta = ([\![\pi \vdash e_0 : \theta \rightarrow \theta']\!]\eta)([\![\pi \vdash e_1 : \theta]\!]\eta),$$

TY SEM EQ: Product Introduction (15.6)

$$[\![\pi \vdash \langle e_0, \ldots, e_{n-1}\rangle : \mathbf{prod}(\theta_0, \ldots, \theta_{n-1})]\!]\eta = \langle [\![\pi \vdash e_0 : \theta_0]\!]\eta, \ldots, [\![\pi \vdash e_{n-1} : \theta_{n-1}]\!]\eta\rangle,$$

TY SEM EQ: Product Elimination (15.7)

$$[\![\pi \vdash e.k : \theta_k]\!]\eta = ([\![\pi \vdash e : \mathbf{prod}(\theta_0, \ldots, \theta_{n-1})]\!]\eta).k \qquad \text{when } k < n,$$

TY SEM EQ: Sum Introduction (15.8)

$$[\![\pi \vdash @\, k\, e : \mathbf{sum}(\theta_0, \ldots, \theta_{n-1})]\!]\eta = \iota_\uparrow\langle k, [\![\pi \vdash e : \theta_k]\!]\eta\rangle \qquad \text{when } k < n,$$

TY SEM EQ: Sum Elimination (15.9)

$$[\![\pi \vdash \mathbf{sumcase}\ e\ \mathbf{of}\ (e_0, \ldots, e_{n-1}) : \theta]\!]\eta = (\lambda\langle k, z\rangle \in \mathcal{D}(\theta_0) + \cdots + \mathcal{D}(\theta_{n-1}).\ [\![\pi \vdash e_k : \theta_k \rightarrow \theta]\!]\eta z)_{\perp\!\!\perp} ([\![\pi \vdash e : \mathbf{sum}(\theta_0, \ldots, \theta_{n-1})]\!]\eta),$$

TY SEM EQ: Fixed Point (15.13)

$$[\![\pi \vdash \mathbf{rec}\ e : \theta]\!]\eta = \mathbf{Y}_{\mathcal{D}(\theta)}([\![\pi \vdash e : \theta \to \theta]\!]\eta).$$

To see how these rules define the meaning of expressions, consider the first proof of a typing judgement giving in Section 15.2:

1. $\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f} : \mathbf{int} \to \mathbf{int}$ (15.3)
2. $\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{x} : \mathbf{int}$ (15.3)
3. $\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f}\,\mathsf{x} : \mathbf{int}$ (15.5,1,2)
4. $\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int}$ (15.5,1,3)
5. $\mathsf{f}\colon \mathbf{int} \to \mathbf{int} \vdash \lambda\mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int} \to \mathbf{int}$ (15.4,4)
6. $\vdash \lambda\mathsf{f}.\ \lambda\mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}$ (15.4,5)

For each step of this proof, corresponding to the type inference rule used to infer the step, there is a semantic equation expressing the meaning of the judgement at that step in terms of the meanings of the earlier judgements that were premisses to the inference. Thus one can obtain the meaning of the judgements in the proof by considering the steps in succession:

1. $[\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f} : \mathbf{int} \to \mathbf{int}]\!]\eta = \eta\,\mathsf{f}$ (15.3)

2. $[\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{x} : \mathbf{int}]\!]\eta = \eta\,\mathsf{x}$ (15.3)

3. $[\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f}\,\mathsf{x} : \mathbf{int}]\!]\eta$ (15.5,1,2)
   $= ([\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f} : \mathbf{int} \to \mathbf{int}]\!]\eta)([\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{x} : \mathbf{int}]\!]\eta)$
   $= (\eta\,\mathsf{f})(\eta\,\mathsf{x})$

4. $[\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int}]\!]\eta$ (15.5,1,3)
   $= ([\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f} : \mathbf{int} \to \mathbf{int}]\!]\eta)([\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f}\,\mathsf{x} : \mathbf{int}]\!]\eta)$
   $= (\eta\,\mathsf{f})((\eta\,\mathsf{f})(\eta\,\mathsf{x}))$

5. $[\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int} \vdash \lambda\mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int} \to \mathbf{int}]\!]\eta$ (15.4,4)
   $= \lambda z \in \mathbf{Z}_\perp.\ [\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int}, \mathsf{x}\colon \mathbf{int} \vdash \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int}]\!][\eta \mid \mathsf{x}\colon z]$
   $= \lambda z \in \mathbf{Z}_\perp.\ ([\eta \mid \mathsf{x}\colon z]\,\mathsf{f})(([\eta \mid \mathsf{x}\colon z]\,\mathsf{f})([\eta \mid \mathsf{x}\colon z]\,\mathsf{x}))$
   $= \lambda z \in \mathbf{Z}_\perp.\ (\eta\,\mathsf{f})((\eta\,\mathsf{f})\,z)$

6. $[\![\vdash \lambda\mathsf{f}.\ \lambda\mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : (\mathbf{int} \to \mathbf{int}) \to \mathbf{int} \to \mathbf{int}]\!]\eta$ (15.4,5)
   $= \lambda z' \in \mathbf{Z}_\perp \to \mathbf{Z}_\perp.\ [\![\mathsf{f}\colon \mathbf{int} \to \mathbf{int} \vdash \lambda\mathsf{x}.\ \mathsf{f}(\mathsf{f}\,\mathsf{x}) : \mathbf{int} \to \mathbf{int}]\!][\eta \mid \mathsf{f}\colon z']$
   $= \lambda z' \in \mathbf{Z}_\perp \to \mathbf{Z}_\perp.\ \lambda z \in \mathbf{Z}_\perp.\ ([\eta \mid \mathsf{f}\colon z']\,\mathsf{f})(([\eta \mid \mathsf{f}\colon z']\,\mathsf{f})\,z)$
   $= \lambda z' \in \mathbf{Z}_\perp \to \mathbf{Z}_\perp.\ \lambda z \in \mathbf{Z}_\perp.\ z'(z'\,z).$

A subtle question remains: What if there are several proofs of the same judgement (which are distinct as trees, not just sequences)? Clearly, in a sensible intrinsic semantics, all such proofs must lead to the same meaning for the judgement — this property is called *coherence*. For our simply typed language, it is relatively straightforward to establish coherence, since the type inference rules are syntax-directed: For each constructor of the abstract syntax, there is a unique rule that shows how to infer a judgement about a constructed expression from judgements about its subphrases. Thus there is a one-to-one correspondence between proofs (viewed as trees) and abstract syntax trees. In Section 16.6, however, when we consider the intrinsic semantics of subtyping and intersection types, coherence will become a serious constraint.

In Section 10.5, when we discussed the denotational semantics of normal-order evaluation for the untyped lambda calculus, we emphasized the distinction between a closed expression that diverges and one that evaluates to an abstraction whose applications always diverge; in the language of this chapter, an example is provided by $\mathbf{rec}\,(\lambda \mathsf{f}.\ \mathsf{f})$ and $\lambda \mathsf{x}.\ (\mathbf{rec}\,(\lambda \mathsf{f}.\ \mathsf{f})\,\mathsf{x}$, both of type $\mathbf{int} \to \mathbf{int}$ (or, more generally, of any type of the form $\theta \to \theta'$). In our untyped denotational semantics, these expressions had the distinct meanings $\perp_{V_*}$ and $\iota_{\text{norm}}(\iota_{\text{fun}} \perp_{V_{\text{fun}}})$, respectively. In our intrinsic typed semantics, however, they both have the same meaning, which is the least element of $\mathcal{D}\,\theta \to \mathcal{D}\,\theta'$. (Analogously, in our extrinsic semantics, the results $\perp_{V_*}$ and $\iota_{\text{norm}}(\iota_{\text{fun}} \perp_{V_{\text{fun}}})$ are identified by the partial equivalence relation $\mathcal{P}(\theta \to \theta')$.)

This raises the fear that the intrinsic semantics might be unsound, in the sense of Section 2.8. But in fact, it is sound if one restricts the observable phrases to closed phrases of *primitive* type (i.e. **int** and **bool**) and only observes whether such a phrase terminates and, if so, its value. It can be shown that, for such phases, this behavior is determined by their intrinsic semantics: When $\vdash e : \mathbf{int}$,

$$e \Rightarrow n \text{ if and only if } [\![\vdash e : \mathbf{int}]\!] = \iota_{\uparrow} n$$
$$e \uparrow \text{ if and only if } [\![\vdash e : \mathbf{int}]\!] = \perp,$$

and similarly for **bool**. Thus, if O is an observation of termination or of a value, and C is a context that takes on primitive type γ when filled with any phrase e satisfying $\pi \vdash e : \theta$, then

$$[\![\pi \vdash e : \theta]\!] = [\![\pi \vdash e' : \theta]\!]$$

implies

$$[\![\vdash C[e] : \gamma]\!] = [\![\vdash C[e'] : \gamma]\!]$$

(by the compositionality of the intrinsic semantics), which in turn implies

$$O(C[e]) = O(C[e']).$$

On the other hand, the intrinsic semantics is not fully abstract. This surprising result is a consequence of two facts:

- There is a value in $\mathbf{B}_\perp \to \mathbf{B}_\perp \to \mathbf{B}_\perp$,

$$\begin{aligned} \mathit{por}\ b\ b' = \ & \textbf{if } b = \textbf{true} \text{ or } b' = \textbf{true then true else} \\ & \textbf{if } b = \perp \text{ or } b' = \perp \textbf{ then } \perp \textbf{ else false} \end{aligned}$$

  (called the *parallel or*), that is not the meaning of any closed expression of type **bool** $\to$ **bool** $\to$ **bool**. Roughly speaking, this is because evaluation for our language is purely sequential, so that if a function begins to evaluate an argument of primitive type, it can terminate only if the argument terminates. In contrast, to evaluate an application of *por*, one would need to evaluate the arguments in parallel, terminating if either evaluation terminates with the value **true**. (Despite this need for parallelism, *por* is still a determinate function.)
- There are two closed expressions, both of type (**bool** $\to$ **bool** $\to$ **bool**) $\to$ **bool**,

$$\begin{aligned} T \stackrel{\text{def}}{=} \lambda \mathsf{p}.\ & \textbf{if } \mathsf{p}\ \textbf{true}\ (\textbf{rec } \lambda \mathsf{x}.\ \mathsf{x})\ \textbf{then} \\ & \quad \textbf{if } \mathsf{p}\ (\textbf{rec } \lambda \mathsf{x}.\ \mathsf{x})\ \textbf{true then} \\ & \quad\quad \textbf{if } \mathsf{p}\ \textbf{false false then rec } \lambda \mathsf{x}.\ \mathsf{x}\ \textbf{else true} \\ & \quad \textbf{else rec } \lambda \mathsf{x}.\ \mathsf{x} \\ & \textbf{else rec } \lambda \mathsf{x}.\ \mathsf{x} \end{aligned}$$

$$T' \stackrel{\text{def}}{=} \lambda \mathsf{p}.\ \textbf{rec } \lambda \mathsf{x}.\ \mathsf{x},$$

  whose meanings are functions that give distinct results when applied to *por*, but give the same result when applied to any other member of the domain $\mathbf{B}_\perp \to \mathbf{B}_\perp \to \mathbf{B}_\perp$.

Clearly, T and T' have distinct denotational semantics. But it can be shown that $C[T]$ and $C[T']$ have the same semantics for all closed contexts C of primitive type; the basic reason is that, when T and T' are applied to any expression, they will yield the same result.

15.6 Set-Theoretic Semantics

At the end of Section 15.2, we remarked that, in the absence of the **letrec** and **rec** constructions, all simply typed expressions of our functional language terminate. This suggests that the sublanguage where recursive definitions and fixed points are eliminated should possess an intrinsic semantics in which partially defined domain elements, and indeed the whole apparatus of an approximation ordering, play no role.

In fact, there is such an intrinsic semantics, in which types denote ordinary sets and $S \to S'$ denotes the set of all functions from S to S'. Indeed, it is this *set-theoretic* semantics that justifies the use of typed lambda expressions (and the other nonrecursive constructions of our simply typed functional language) in ordinary mathematical discourse, as for example in Sections A.3 or 2.5.

It is straightforward to obtain the set-theoretic model by modifying the intrinsic semantics given in the previous section. Equations for the functions $\mathcal{D}$ and $\mathcal{D}^*$, which now map types and contexts into sets, are obtained by dropping the lifting operations:

$$\begin{aligned}
\mathcal{D}(\mathbf{int}) &= \mathbf{Z} \\
\mathcal{D}(\mathbf{bool}) &= \mathbf{B} \\
\mathcal{D}(\theta \to \theta') &= \mathcal{D}(\theta) \to \mathcal{D}(\theta') \\
\mathcal{D}(\mathbf{prod}(\theta_0, \ldots, \theta_{n-1})) &= \mathcal{D}(\theta_0) \times \cdots \times \mathcal{D}(\theta_{n-1}) \\
\mathcal{D}(\mathbf{sum}(\theta_0, \ldots, \theta_{n-1})) &= \mathcal{D}(\theta_0) + \cdots + \mathcal{D}(\theta_{n-1}) \\
\mathcal{D}^*(\pi) &= \prod_{v \in \mathrm{dom}\, \pi} \mathcal{D}(\pi\, v).
\end{aligned}$$

Here $\to$, $\times$, $+$, and $\prod$ are given their conventional meanings for sets, as defined in Sections A.3 and A.4.

The semantic equations for judgements are obtained from those in the previous section by eliminating the injections $\iota_\uparrow$ and the function extensions $(-)_{\perp\!\!\perp}$. (Several equations can then be simplified by β-contraction.)

TY SEM EQ: Constants and Primitive Operations (15.1)

$$\begin{aligned}
[\![\pi \vdash \mathbf{true}\colon \mathbf{bool}]\!]\eta &= \mathbf{true} \\
[\![\pi \vdash 0\colon \mathbf{int}]\!]\eta &= 0 \\
[\![\pi \vdash e_0 + e_1\colon \mathbf{int}]\!]\eta &= ([\![\pi \vdash e_0\colon \mathbf{int}]\!]\eta) + ([\![\pi \vdash e_1\colon \mathbf{int}]\!]\eta) \\
[\![\pi \vdash e_0 = e_1\colon \mathbf{bool}]\!]\eta &= ([\![\pi \vdash e_0\colon \mathbf{int}]\!]\eta) = ([\![\pi \vdash e_1\colon \mathbf{int}]\!]\eta) \\
[\![\pi \vdash e_0 \wedge e_1\colon \mathbf{bool}]\!]\eta &= \mathbf{if}\ [\![\pi \vdash e_0\colon \mathbf{bool}]\!]\eta\ \mathbf{then}\ [\![\pi \vdash e_1\colon \mathbf{bool}]\!]\eta\ \mathbf{else}\ \mathbf{false},
\end{aligned}$$

TY SEM EQ: Conditional Expressions (15.2)

$$\begin{aligned}
&[\![\pi \vdash \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \theta]\!]\eta \\
&\quad = \mathbf{if}\ [\![\pi \vdash e_0 : \mathbf{bool}]\!]\eta\ \mathbf{then}\ [\![\pi \vdash e_1 : \theta]\!]\eta\ \mathbf{else}\ [\![\pi \vdash e_2 : \theta]\!]\eta,
\end{aligned}$$

TY SEM EQ: Variables (15.3)

$$[\![\pi \vdash v : \theta]\!]\eta = \eta\, v \qquad \text{when } \pi \text{ contains } v\colon \theta,$$

TY SEM EQ: $\rightarrow$ Introduction (15.4)

$$[\![\pi \vdash \lambda v.\, e : \theta \rightarrow \theta']\!]\eta = \lambda z \in \mathcal{D}(\theta).\, [\![\pi, v{:}\theta \vdash e : \theta']\!][\,\eta \mid v{:}\,z\,],$$

TY SEM EQ: $\rightarrow$ Elimination (15.5)

$$[\![\pi \vdash e_0\, e_1 : \theta']\!]\eta = ([\![\pi \vdash e_0 : \theta \rightarrow \theta']\!]\eta)([\![\pi \vdash e_1 : \theta]\!]\eta),$$

TY SEM EQ: Product Introduction (15.6)

$$\begin{aligned}[\![\pi \vdash \langle e_0, \ldots, e_{n-1}\rangle : \mathbf{prod}(\theta_0, \ldots, \theta_{n-1})]\!]\eta \\ = \langle [\![\pi \vdash e_0 : \theta_0]\!]\eta, \ldots, [\![\pi \vdash e_{n-1} : \theta_{n-1}]\!]\eta\rangle,\end{aligned}$$

TY SEM EQ: Product Elimination (15.7)

$$[\![\pi \vdash e.k : \theta_k]\!]\eta = ([\![\pi \vdash e : \mathbf{prod}(\theta_0, \ldots, \theta_{n-1})]\!]\eta).k \qquad \text{when } k < n,$$

TY SEM EQ: Sum Introduction (15.8)

$$[\![\pi \vdash @\, k\, e : \mathbf{sum}(\theta_0, \ldots, \theta_{n-1})]\!]\eta = \langle k, [\![\pi \vdash e : \theta_k]\!]\eta\rangle \qquad \text{when } k < n,$$

TY SEM EQ: Sum Elimination (15.9)

$$\begin{aligned}[\![\pi \vdash \mathbf{sumcase}\ e\ \mathbf{of}\ (e_0, \ldots, e_{n-1}) : \theta]\!]\eta \\ = (\lambda\langle k, z\rangle \in \mathcal{D}(\theta_0) + \cdots + \mathcal{D}(\theta_{n-1}).\, [\![\pi \vdash e_k : \theta_k \rightarrow \theta]\!]\eta z) \\ ([\![\pi \vdash e : \mathbf{sum}(\theta_0, \ldots, \theta_{n-1})]\!]\eta).\end{aligned}$$

The essence of set-theoretic semantics is that no restriction, such as continuity, is imposed on the notion of function. Because of paradoxes such as Russell's, such a semantics is possible only for a sufficiently restrictive typing discipline.

15.7 Recursive Types

In developing types we have tacitly assumed that, like other syntactic phrases, they are finite. In fact, however, our syntactic development extends smoothly to infinite types. For example, as defined in Section 11.4, a list of integers is either $@\,0\,\langle\rangle$ or $@\,1\,\langle i, r\rangle$, where i is an integer and r is a list of integers. Thus the type **list int** is really an abbreviation for the infinite type

$$\begin{aligned}&\mathbf{sum}(\mathbf{unit}, \mathbf{int} \times \\ &\quad \mathbf{sum}(\mathbf{unit}, \mathbf{int} \times \\ &\qquad \mathbf{sum}(\mathbf{unit}, \mathbf{int} \times \\ &\qquad\quad \cdots))).\end{aligned}$$