on the second argument y. So the result of the first application of *minus* must be function, which is directly represented in the definition below.

$$\begin{array}{lcl} \mathit{minus} & = & \lambda x \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & & \quad \mathbf{of}\ m(\mathbf{0}) \Rightarrow \lambda y \in \mathbf{nat}.\ 0 \\ & & \quad \mid m(\mathbf{s}(x')) \Rightarrow \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ y \\ & & \qquad\qquad \mathbf{of}\ p(\mathbf{0}) \Rightarrow \mathbf{s}(x') \\ & & \qquad\qquad \mid p(\mathbf{s}(y')) \Rightarrow (m\,(x'))\,y' \end{array}$$

Note that m is correctly applied only to x', while p is not used at all. So the inner recursion could have been written as a **case**-expression instead.

Functions defined by primitive recursion terminate. This is because the behavior of the function on $\mathbf{s}(n)$ is defined in terms of the behavior on n. We can therefore count down to $\mathbf{0}$, in which case no recursive call is allowed. An alternative approach is to take **case** as primitive and allow arbitrary recursion. In such a language it is much easier to program, but not every function terminates. We will see that for our purpose about integrating constructive reasoning and functional programming it is simpler if all functions one can write down are *total*, that is, are defined on all arguments. This is because total functions can be used to provide witnesses for propositions of the form $\forall x \in \mathbf{nat}.\ \exists y \in \mathbf{nat}.\ P(x, y)$ by showing how to compute y from x. Functions that may not return an appropriate y cannot be used in this capacity and are generally much more difficult to reason about.

1.6 Booleans

Another simple example of a data type is provided by the Boolean type with two elements **true** and **false**. This should *not* be confused with the propositions $\top$ and $\bot$. In fact, they correspond to the unit type $\mathbf{1}$ and the empty type $\mathbf{0}$. We recall their definitions first, in analogy with the propositions.

$$\frac{}{\mathbf{1}\ \mathit{type}}\ \mathbf{1}F$$

$$\frac{}{\Gamma \vdash \langle\rangle \in \mathbf{1}}\ \mathbf{1}I \qquad\qquad \text{no } \mathbf{1} \text{ elimination rule}$$

$$\frac{}{\mathbf{0}\ \mathit{type}}\ \mathbf{0}F$$

$$\text{no } \mathbf{0} \text{ introduction rule} \qquad\qquad \frac{\Gamma \vdash t \in \mathbf{0}}{\Gamma \vdash \mathbf{abort}^{\tau}\ t \in \tau}\ \mathbf{0}E$$

There are no reduction rules at these types.

The Boolean type, **bool**, is instead defined by two introduction rules.

$$\frac{}{\mathbf{bool}\ \mathit{type}}\ \mathbf{bool}F$$

$$\frac{}{\Gamma \vdash \mathbf{true} \in \mathbf{bool}}\ \mathbf{bool}I_1 \qquad\qquad \frac{}{\Gamma \vdash \mathbf{false} \in \mathbf{bool}}\ \mathbf{bool}I_0$$

The elimination rule follows the now familiar pattern: since there are two introduction rules, we have to distinguish two cases for a given Boolean value. This could be written as

$$\mathbf{case}\, t\, \mathbf{of}\, \mathbf{true} \Rightarrow s_1 \mid \mathbf{false} \Rightarrow s_0$$

but we typically express the same program as an **if** t **then** s_1 **else** s_0.

$$\frac{\Gamma \vdash t \in \mathbf{bool} \qquad \Gamma \vdash s_1 \in \tau \qquad \Gamma \vdash s_0 \in \tau}{\Gamma \vdash \mathbf{if}\, t\, \mathbf{then}\, s_1\, \mathbf{else}\, s_0 \in \tau}\,\mathbf{bool}E$$

The reduction rules just distinguish the two cases for the subject of the **if**-expression.

$$\begin{array}{rcl} \mathbf{if}\,\mathbf{true}\,\mathbf{then}\, s_1\, \mathbf{else}\, s_0 & \Longrightarrow & s_1 \\ \mathbf{if}\,\mathbf{false}\,\mathbf{then}\, s_1\, \mathbf{else}\, s_0 & \Longrightarrow & s_0 \end{array}$$

Now we can define typical functions on booleans, such as *and*, *or*, and *not*.

$$\begin{array}{rcl} \mathit{and} & = & \lambda x \in \mathbf{bool}.\, \lambda y \in \mathbf{bool}. \\ & & \quad \mathbf{if}\, x\, \mathbf{then}\, y\, \mathbf{else}\, \mathbf{false} \\ \mathit{or} & = & \lambda x \in \mathbf{bool}.\, \lambda y \in \mathbf{bool}. \\ & & \quad \mathbf{if}\, x\, \mathbf{then}\, \mathbf{true}\, \mathbf{else}\, y \\ \mathit{not} & = & \lambda x \in \mathbf{bool}. \\ & & \quad \mathbf{if}\, x\, \mathbf{then}\, \mathbf{false}\, \mathbf{else}\, \mathbf{true} \end{array}$$

1.7 Lists

Another more interesting data type is that of lists. Lists can be created with elements from any type whatsoever, which means that $\tau\,\mathbf{list}$ is a type for any type τ.

$$\frac{\tau\ \mathit{type}}{\tau\,\mathbf{list}\ \mathit{type}}\,\mathbf{list}F$$

Lists are built up from the empty list (**nil**) with the operation :: (pronounced "cons"), written in infix notation.

$$\frac{}{\Gamma \vdash \mathbf{nil}^{\tau} \in \tau\,\mathbf{list}}\,\mathbf{list}I_n \qquad\qquad \frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash s \in \tau\,\mathbf{list}}{\Gamma \vdash t :: s \in \tau\,\mathbf{list}}\,\mathbf{list}I_c$$

The elimination rule implements the schema of primitive recursion over lists. It can be specified as follows:

$$\begin{array}{rcl} f\,(\mathbf{nil}) & = & s_n \\ f\,(x :: l) & = & s_c(x, l, f(l)) \end{array}$$

where we have indicated that s_c may mention x, l, and $f(l)$, but no other occurrences of f. Again this guarantees termination.

$$\frac{\Gamma \vdash t \in \tau\,\mathbf{list} \qquad \Gamma \vdash s_n \in \sigma \qquad \Gamma, x \in \tau, l \in \tau\,\mathbf{list}, f(l) \in \sigma \vdash s_c \in \sigma}{\Gamma \vdash \mathbf{rec}\, t\, \mathbf{of}\, f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c \in \sigma}\,\mathbf{list}E$$

We have overloaded the **rec** constructor here—from the type of t we can always tell if it should recurse over natural numbers or lists. The reduction rules are once again recursive, as in the case for natural numbers.

$$\begin{array}{rcl}
(\mathbf{rec}\,\mathbf{nil}\,\mathbf{of}\,f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) & \Longrightarrow & s_n \\
(\mathbf{rec}\,(h :: t)\,\mathbf{of}\,f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) & \Longrightarrow & \\
\multicolumn{3}{c}{[(\mathbf{rec}\,t\,\mathbf{of}\,f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c)/f(l)]\,[h/x]\,[t/l]\,s_c}
\end{array}$$

Now we can define typical operations on lists via primitive recursion. A simple example is the *append* function to concatenate two lists.

$$\begin{array}{rcl}
\mathit{append}\,\mathbf{nil}\,k & = & k \\
\mathit{append}\,(x :: l')\,k & = & x :: (\mathit{append}\,l'\,k)
\end{array}$$

In the notation of primitive recursion:

$$\begin{array}{rcl}
\mathit{append} & = & \lambda l \in \tau\,\mathbf{list}.\ \lambda k \in \tau\,\mathbf{list}.\ \mathbf{rec}\ l \\
& & \qquad\qquad\qquad\qquad \mathbf{of}\ a(\mathbf{nil}) \Rightarrow k \\
& & \qquad\qquad\qquad\qquad \mid a(x :: l') \Rightarrow x :: (a\,l') \\
\vdash \mathit{append} & \in & \tau\,\mathbf{list} \to \tau\,\mathbf{list} \to \tau\,\mathbf{list}
\end{array}$$

Note that the last judgment is parametric in τ, a situation referred to as *parametric polymorphism*. In means that the judgment is valid for every type τ. We have encountered a similar situation, for example, when we asserted that $(A \wedge B) \supset A$ *true*. This judgment is parametric in A and B, and every instance of it by propositions A and B is evident, according to our derivation.

As a second example, we consider a program to reverse a list. The idea is to take elements out of the input list l and attach them to the front of a second list a one which starts out empty. The first list has been traversed, the second has accumulated the original list in reverse. If we call this function *rev* and the original one *reverse*, it satisfies the following specification.

$$\begin{array}{rcl}
\mathit{rev} & \in & \tau\,\mathbf{list} \to \tau\,\mathbf{list} \to \tau\,\mathbf{list} \\
\mathit{rev}\ \mathbf{nil}\ a & = & a \\
\mathit{rev}\ (x :: l')\ a & = & \mathit{rev}\ l'\ (x :: a) \\[1ex]
\mathit{reverse} & \in & \tau\,\mathbf{list} \to \tau\,\mathbf{list} \\
\mathit{reverse}\ l & = & \mathit{rev}\ l\ \mathbf{nil}
\end{array}$$

In programs of this kind we refer to a as the *accumulator argument* since it accumulates the final result which is returned in the base case. We can see that except for the additional argument a, the *rev* function is primitive recursive. To make this more explicit we can rewrite the definition of *rev* to the following equivalent form:

$$\begin{array}{rcl}
\mathit{rev}\ \mathbf{nil} & = & \lambda a.\ a \\
\mathit{rev}\ (x :: l) & = & \lambda a.\ \mathit{rev}\ l\ (x :: a)
\end{array}$$

Now the transcription into our notation is direct.

$$\begin{array}{rcl}
\mathit{rev} & = & \lambda l \in \tau\,\mathbf{list}.\ \mathbf{rec}\ l \\
& & \qquad\qquad \mathbf{of}\ r(\mathbf{nil}) \Rightarrow \lambda a \in \tau\,\mathbf{list}.\ a \\
& & \qquad\qquad \mid r(x :: l') \Rightarrow \lambda a \in \tau\,\mathbf{list}.\ r\ (l')\ (x :: a) \\
\mathit{reverse}\ l & = & \mathit{rev}\ l\ \mathbf{nil}
\end{array}$$

Finally a few simple functions which mix data types. The first counts the number of elements in a list.

$$\begin{array}{rcl} \mathit{length} & \in & \tau\,\mathbf{list} \rightarrow \mathbf{nat} \\ \\ \mathit{length}\ \mathbf{nil} & = & \mathbf{0} \\ \mathit{length}\ (x :: l') & = & \mathbf{s}(\mathit{length}\ (l')) \end{array}$$

$$\begin{array}{rcl} \mathit{length} & = & \lambda x \in \tau\,\mathbf{list}.\ \mathbf{rec}\ x \\ & & \qquad\qquad\qquad \mathbf{of}\ \mathit{le}(\mathbf{nil}) \Rightarrow \mathbf{0} \\ & & \qquad\qquad\qquad \mid \mathit{le}(x :: l') \Rightarrow \mathbf{s}(\mathit{le}\ (l')) \end{array}$$

The second compares two numbers for equality.

$$\begin{array}{rcl} \mathit{eq} & \in & \mathbf{nat} \rightarrow \mathbf{nat} \rightarrow \mathbf{bool} \\ \\ \mathit{eq}\ \mathbf{0}\ \mathbf{0} & = & \mathbf{true} \\ \mathit{eq}\ \mathbf{0}\ (\mathbf{s}(y')) & = & \mathbf{false} \\ \mathit{eq}\ (\mathbf{s}(x'))\ \mathbf{0} & = & \mathbf{false} \\ \mathit{eq}\ (\mathbf{s}(x'))\ (\mathbf{s}(y')) & = & \mathit{eq}\ x'\ y' \end{array}$$

As in the example of subtraction, we need to distinguish two levels.

$$\begin{array}{rcl} \mathit{eq} & = & \lambda x \in \mathbf{nat}.\ \mathbf{rec}\ x \\ & & \quad \mathbf{of}\ e(\mathbf{0}) \Rightarrow \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ y \\ & & \qquad\qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{true} \\ & & \qquad\qquad\qquad \mid f(\mathbf{s}(y')) \Rightarrow \mathbf{false} \\ & & \quad \mid e(\mathbf{s}(x')) \Rightarrow \lambda y \in \mathbf{nat}.\ \mathbf{rec}\ y \\ & & \qquad\qquad\qquad \mathbf{of}\ f(\mathbf{0}) \Rightarrow \mathbf{false} \\ & & \qquad\qquad\qquad \mid f(\mathbf{s}(y')) \Rightarrow e(x')\ y' \end{array}$$

We will see more examples of primitive recursive programming as we proceed to first order logic and quantification.

1.8 Summary of Data Types

Judgments.

τ *type*	τ is a type
$t \in \tau$	t is a term of type τ

Type Formation.

$$\frac{}{\mathbf{nat}\ \mathit{type}}\,\mathbf{nat}F \qquad \frac{}{\mathbf{bool}\ \mathit{type}}\,\mathbf{bool}F \qquad \frac{\tau\ \mathit{type}}{\tau\,\mathbf{list}\ \mathit{type}}\,\mathbf{list}F$$

Term Formation.

$$\frac{}{\mathbf{0} \in \mathbf{nat}}\,\mathbf{nat}I_0 \qquad \frac{n \in \mathbf{nat}}{\mathbf{s}(n) \in \mathbf{nat}}\,\mathbf{nat}I_s$$

$$\frac{\Gamma \vdash t \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat}, f(x) \in \tau \vdash t_s \in \tau}{\Gamma \vdash \mathbf{rec}\ t\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s \in \tau}\,\mathbf{nat}E$$

$$\frac{}{\Gamma \vdash \mathbf{true} \in \mathbf{bool}}\,\mathbf{bool}I_1 \qquad \frac{}{\Gamma \vdash \mathbf{false} \in \mathbf{bool}}\,\mathbf{bool}I_0$$

$$\frac{\Gamma \vdash t \in \mathbf{bool} \qquad \Gamma \vdash s_1 \in \tau \qquad \Gamma \vdash s_0 \in \tau}{\Gamma \vdash \mathbf{if}\ t\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 \in \tau}\,\mathbf{bool}E$$

$$\frac{}{\Gamma \vdash \mathbf{nil}^{\tau} \in \tau\,\mathbf{list}}\,\mathbf{list}I_n \qquad \frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash s \in \tau\,\mathbf{list}}{\Gamma \vdash t :: s \in \tau\,\mathbf{list}}\,\mathbf{list}I_c$$

$$\frac{\Gamma \vdash t \in \tau\,\mathbf{list} \qquad \Gamma \vdash s_n \in \sigma \qquad \Gamma, x \in \tau, l \in \tau\,\mathbf{list}, f(l) \in \sigma\,\mathbf{list} \vdash s_c \in \sigma}{\Gamma \vdash \mathbf{rec}\ t\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c \in \sigma}\,\mathbf{list}E$$

Reductions.

$$\begin{array}{rcl}
(\mathbf{rec}\ \mathbf{0}\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) & \Longrightarrow & t_0 \\
(\mathbf{rec}\ \mathbf{s}(n)\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) & \Longrightarrow & \\
\multicolumn{3}{r}{[(\mathbf{rec}\ n\ \mathbf{of}\ f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s)/f(x)]\,[n/x]\,t_s} \\
\mathbf{if}\ \mathbf{true}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 & \Longrightarrow & s_1 \\
\mathbf{if}\ \mathbf{false}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_0 & \Longrightarrow & s_0 \\
(\mathbf{rec}\ \mathbf{nil}\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) & \Longrightarrow & s_n \\
(\mathbf{rec}\ (h :: t)\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c) & \Longrightarrow & \\
\multicolumn{3}{r}{[(\mathbf{rec}\ t\ \mathbf{of}\ f(\mathbf{nil}) \Rightarrow s_n \mid f(x :: l) \Rightarrow s_c)/f(l)]\,[h/x]\,[t/l]\,s_c}
\end{array}$$

1.9 Predicates on Data Types

In the preceding sections we have introduced the concept of a type which is determined by its elements. Examples were natural numbers, Booleans, and lists. In the next chapter we will explicitly quantify over elements of types. For example, we may assert that every natural number is either even or odd. Or we may claim that any two numbers possess a greatest common divisor. In order to formulate such statements we need some basic propositions concerned with data types. In this section we will define such predicates, following our usual methodology of using introduction and elimination rules to define the meaning of propositions.