3.5 Primitive Recursion

In the preceding sections we have developed an interpretation of propositions as types. This interpretation yields function types (from implication), product types (from conjunction), unit type (from truth), sum types (from disjunction) and the empty type (from falsehood). What is missing for a reasonable programming language are basic data types such as natural numbers, integers, lists, trees, etc. There are several approaches to incorporating such types into our framework. One is to add a general definition mechanism for *recursive types* or *inductive types*. We return to this option later. Another one is to specify each type in a way which is analogous to the definitions of the logical connectives via introduction and elimination rules. This is the option we pursue in this section. A third way is to use the constructs we already have to define data. This was Church's original approach culminating in the so-called *Church numerals*. We will not discuss this idea in these notes.

After spending some time to illustrate the interpretation of propositions as types, we now introduce types as a first-class notion. This is not strictly necessary, but it avoids the question what, for example, **nat** (the type of natural numbers) means as a proposition. Accordingly, we have a new judgment τ *type* meaning "τ *is a type*". To understand the meaning of a type means to understand what elements it has. We therefore need a second judgment $t \in \tau$ (read: "t *is an element of type* τ") that is defined by introduction rules with their corresponding elimination rules. As in the case of logical connectives, computation arises from the meeting of elimination and introduction rules. Needless to say, we will continue to use our mechanisms of hypothetical judgments.

Before introducing any actual data types, we look ahead at their use in logic. We will introduce new propositions of the form $\forall x \in \tau.\ A(x)$ (A *is true for every element* x *of type* τ) and $\exists x \in \tau.\ A(x)$ (A *is true some some element* x *of type* τ). This will be the step from propositional logic to first-order logic. This logic is called *first-order* because we can quantify (via $\forall$ and $\exists$) only over elements of data types, but not propositions themselves.

We begin our presentation of data types with the natural numbers. The formation rule is trivial: **nat** is a type.

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

Now we state two of Peano's famous axioms in judgmental form as introduction rules: (1) **0** is a natural numbers, and (2) if n is a natural number then its successor, $\mathbf{s}(n)$, is a natural number. We write $\mathbf{s}(n)$ instead of $n+1$, since addition and the number 1 have yet to be defined.

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

The elimination rule is a bit more difficult to construct. Assume have a natural number n. Now we cannot directly take its predecessor, for example, because we do not know if n was constructed using $\mathbf{nat}I_0$ or $\mathbf{nat}I_s$. This is similar to the case of disjunction, and our solution is also similar: we distinguish cases. In general, it turns out this is not sufficient, but our first approximation for an elimination rule is

$$\dfrac{n \in \mathbf{nat} \qquad t_0 \in \tau \qquad \begin{array}{c}\dfrac{}{x \in \mathbf{nat}}x \\ \vdots \\ t_s \in \tau\end{array}}{\mathbf{case}\, n\ \mathbf{of}\ \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s \in \tau}x$$

Note that x is introduced in the third premise; its scope is t_s. First, we rewrite this using our more concise notation for hypothetical judgments. For now, Γ contains assumptions of the form $x \in \tau$. Later, we will add logical assumptions of the form $u{:}A$.

$$\dfrac{\Gamma \vdash n \in \mathbf{nat} \qquad \Gamma \vdash t_0 \in \tau \qquad \Gamma, x \in \mathbf{nat} \vdash t_s \in \tau}{\Gamma \vdash \mathbf{case}\, n\ \mathbf{of}\ \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s \in \tau}x$$

This elimination rule is sound, and under the computational interpretation of terms, type preservation holds. The reductions rules are

$$\begin{array}{rcl}(\mathbf{case}\, \mathbf{0}\ \mathbf{of}\ \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s) & \Longrightarrow & t_0 \\ (\mathbf{case}\, \mathbf{s}(n)\ \mathbf{of}\ \mathbf{0} \Rightarrow t_0 \mid \mathbf{s}(x) \Rightarrow t_s) & \Longrightarrow & [n/x]t_s\end{array}$$

Clearly, this is the intended reading of the case construct in programs.

In order to use this in writing programs independently of the logic developed earlier, we now introduce function types in a way that is isomorphic to implication.

$$\dfrac{\tau\ \mathit{type} \qquad \sigma\ \mathit{type}}{\tau \to \sigma\ \mathit{type}}{\to}F$$

$$\dfrac{\Gamma, x \in \sigma \vdash t \in \tau}{\Gamma \vdash \lambda x \in \sigma.\, t \in \sigma \to \tau}{\to}I^x \qquad\qquad \dfrac{\Gamma \vdash s \in \tau \to \sigma \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash s\, t \in \sigma}{\to}E$$

$$(\lambda x \in \sigma.\, s)\, t \Longrightarrow [t/x]s$$

Now we can write a function for truncated predecessor: the predecessor of $\mathbf{0}$ is defined to be $\mathbf{0}$; otherwise the predecessor of $n+1$ is simply n. We phrase this as a notational definition.

$$\mathit{pred} \quad = \quad \lambda x \in \mathbf{nat}.\, \mathbf{case}\, x\ \mathbf{of}\ \mathbf{0} \Rightarrow \mathbf{0} \mid \mathbf{s}(y) \Rightarrow y$$

Then $\vdash \mathit{pred} \in \mathbf{nat} \to \mathbf{nat}$ and we can formally calculate the predecessor of 2.

$$\begin{array}{rcl}\mathit{pred}(\mathbf{s}(\mathbf{s}(\mathbf{0}))) & = & (\lambda x \in \mathbf{nat}.\, \mathbf{case}\, x\ \mathbf{of}\ \mathbf{0} \Rightarrow \mathbf{0} \mid \mathbf{s}(y) \Rightarrow y)\, (\mathbf{s}(\mathbf{s}(\mathbf{0}))) \\ & \Longrightarrow & \mathbf{case}\, \mathbf{s}(\mathbf{s}(\mathbf{0}))\ \mathbf{of}\ \mathbf{0} \Rightarrow \mathbf{0} \mid \mathbf{s}(y) \Rightarrow y \\ & \Longrightarrow & \mathbf{s}(\mathbf{0})\end{array}$$

As a next example, we consider a function which doubles its argument. The behavior of the *double* function on an argument can be specified as follows:

$$\begin{array}{rcl} \mathit{double}(\mathbf{0}) & = & \mathbf{0} \\ \mathit{double}(\mathbf{s}(n)) & = & \mathbf{s}(\mathbf{s}(\mathit{double}(n))) \end{array}$$

Unfortunately, there is no way to transcribe this definition into an application of the **case**-construct for natural numbers, since it is *recursive*: the right-hand side contains an occurrence of *double*, the function we are trying to define.

Fortunately, we can generalize the elimination construct for natural numbers to permit this kind of recursion which is called *primitive recursion.* Note that we can define the value of a function on $\mathbf{s}(n)$ only in terms of n and the value of the function on n. We write

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

Here, f may not occur in t_0 and can only occur in the form $f(x)$ in t_s to denote the result of the recursive call. Essentially, $f(x)$ is just the mnemonic name of a new variable for the result of the recursive call. Moreover, x is bound with scope t_s. The reduction rules are now recursive:

$$\begin{array}{rcl} (\mathbf{rec}\;\mathbf{0}\;\mathbf{of}\; f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) & \Longrightarrow & t_0 \\ (\mathbf{rec}\;\mathbf{s}(n)\;\mathbf{of}\; f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s) & \Longrightarrow & \\ \multicolumn{3}{l}{[(\mathbf{rec}\; n\;\mathbf{of}\; f(\mathbf{0}) \Rightarrow t_0 \mid f(\mathbf{s}(x)) \Rightarrow t_s)/f(x)]\,[n/x]\,t_s} \end{array}$$

As an example we revisit the double function and give it as a notational definition.

$$\begin{array}{rcl} \mathit{double} & = & \lambda x \in \mathbf{nat}.\;\mathbf{rec}\; x \\ & & \qquad\qquad \mathbf{of}\; d(\mathbf{0}) \Rightarrow \mathbf{0} \\ & & \qquad\qquad\;\; \mid d(\mathbf{s}(x')) \Rightarrow \mathbf{s}(\mathbf{s}(d(x'))) \end{array}$$

Now *double* $(\mathbf{s}(\mathbf{0}))$ can be computed as follows

$$\begin{array}{rl} & (\lambda x \in \mathbf{nat}.\;\mathbf{rec}\; x \\ & \qquad\qquad \mathbf{of}\; d(\mathbf{0}) \Rightarrow \mathbf{0} \\ & \qquad\qquad\;\; \mid d(\mathbf{s}(x')) \Rightarrow \mathbf{s}(\mathbf{s}(d(x')))) \\ & \;(\mathbf{s}(\mathbf{0})) \\ \Longrightarrow & \mathbf{rec}\;(\mathbf{s}(\mathbf{0})) \\ & \;\mathbf{of}\; d(\mathbf{0}) \Rightarrow \mathbf{0} \\ & \;\; \mid d(\mathbf{s}(x')) \Rightarrow \mathbf{s}(\mathbf{s}(d(x')))) \\ \Longrightarrow & \mathbf{s}(\mathbf{s}(\mathbf{rec}\;\mathbf{0} \\ & \qquad \mathbf{of}\; d(\mathbf{0}) \Rightarrow \mathbf{0} \\ & \qquad\; \mid d(\mathbf{s}(x')) \Rightarrow \mathbf{s}(\mathbf{s}(d(x'))))) \\ \Longrightarrow & \mathbf{s}(\mathbf{s}(\mathbf{0})) \end{array}$$

As some other examples, we consider the functions for addition and multiplication. These definitions are by no means uniquely determined. In each case we first give an implicit definition, describing the intended behavior of the function, and then the realization in our language.

$$\begin{array}{rcl} \mathit{plus}\,\mathbf{0}\,y & = & y \\ \mathit{plus}\,(\mathbf{s}(x'))\,y & = & \mathbf{s}(\mathit{plus}\,x'\,y) \end{array}$$

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

$$\begin{array}{rcl} \mathit{times}\,\mathbf{0}\,y & = & \mathbf{0} \\ \mathit{times}\,(\mathbf{s}(x'))\,y & = & \mathit{plus}\,y\,(\mathit{times}\,x'\,y) \end{array}$$

$$\begin{array}{rcl} \mathit{times} & = & \lambda x \in \mathbf{nat}.\, \lambda y \in \mathbf{nat}.\, \mathbf{rec}\ x \\ & & \quad \mathbf{of}\ t(\mathbf{0}) \Rightarrow \mathbf{0} \\ & & \quad\ \mid\ t(\mathbf{s}(x')) \Rightarrow \mathit{plus}\,y\,(t(x')\,y) \end{array}$$

The next example requires pairs in the language. We therefore introduce pairs which are isomorphic to the proof terms for conjunction from before.

$$\frac{\Gamma \vdash s \in \sigma \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash \langle s, t\rangle \in \sigma \times \tau}\times I$$

$$\frac{\Gamma \vdash t \in \tau \times \sigma}{\Gamma \vdash \mathbf{fst}\,t \in \tau}\times E_L \qquad \frac{\Gamma \vdash t \in \tau \times \sigma}{\Gamma \vdash \mathbf{snd}\,t \in \sigma}\times E_R$$

$$\begin{array}{c} \mathbf{fst}\,\langle t, s\rangle \Longrightarrow t \\ \mathbf{snd}\,\langle t, s\rangle \Longrightarrow s \end{array}$$

Next the function *half*, rounding down if necessary. This is slightly trickier then the examples above, since we would like to count down by *two* as the following specification indicates.

$$\begin{array}{rcl} \mathit{half}\,\mathbf{0} & = & \mathbf{0} \\ \mathit{half}\,(\mathbf{s}(\mathbf{0})) & = & \mathbf{0} \\ \mathit{half}\,(\mathbf{s}(\mathbf{s}(x'))) & = & \mathbf{s}(\mathit{half}\,(x')) \end{array}$$

The first step is to break this function into two, each of which steps down by one.

$$\begin{array}{rcl} \mathit{half}_1\,\mathbf{0} & = & \mathbf{0} \\ \mathit{half}_1\,(\mathbf{s}(x')) & = & \mathit{half}_2(x') \\ \mathit{half}_2\,\mathbf{0} & = & \mathbf{0} \\ \mathit{half}_2\,(\mathbf{s}(x'')) & = & \mathbf{s}(\mathit{half}_1(x'')) \end{array}$$

Note that half_1 calls half_2 and vice versa. This is an example of so-called *mutual recursion*. This can be modeled by one function half_{12} returning a pair such that $\mathit{half}_{12}(x) = \langle \mathit{half}_1(x), \mathit{half}_2(x)\rangle$.

$$\begin{array}{rcl} \mathit{half}_{12}\,\mathbf{0} & = & \langle \mathbf{0}, \mathbf{0}\rangle \\ \mathit{half}_{12}\,(\mathbf{s}(x)) & = & \langle \mathbf{snd}\,(\mathit{half}_{12}(x)), \mathbf{s}(\mathbf{fst}\,(\mathit{half}_{12}(x)))\rangle \\ \mathit{half}\,x & = & \mathbf{fst}\,(\mathit{half}\,x) \end{array}$$

In our notation this becomes

$$\begin{array}{rcl} \mathit{half}_{12} & = & \lambda x \in \mathbf{nat}.\,\mathbf{rec}\,x \\ & & \qquad\qquad \mathbf{of}\,h(\mathbf{0}) \Rightarrow \langle \mathbf{0}, \mathbf{0}\rangle \\ & & \qquad\qquad\ \mid h(\mathbf{s}(x')) \Rightarrow \langle \mathbf{snd}\,(h(x)), \mathbf{s}(\mathbf{fst}\,(h(x)))\rangle \\ \mathit{half} & = & \lambda x \in \mathbf{nat}.\,\mathbf{fst}\,(\mathit{half}_{12}\,x) \end{array}$$

As a last example in the section, consider the subtraction function which cuts off at zero.

$$\begin{array}{rcl} \mathit{minus}\,\mathbf{0}\,y & = & \mathbf{0} \\ \mathit{minus}\,(\mathbf{s}(x'))\,\mathbf{0} & = & \mathbf{s}(x') \\ \mathit{minus}\,(\mathbf{s}(x'))\,(\mathbf{s}(y')) & = & \mathit{minus}\,x'\,y' \end{array}$$

To be presented in the schema of primitive recursion, this requires two nested case distinctions: the outermost one on the first argument x, the innermost one on the second argument y.

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

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.

3.6 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 **1** and the empty type **0**. We recall their definitions first, in analogy with the propositions.

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

$$\frac{}{\Gamma \vdash \langle\,\rangle \in \mathit{type}}\ \mathbf{1}I \qquad\qquad \textit{no } \mathbf{1} \textit{ elimination rule}$$

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

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

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

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

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.

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

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

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

3.7 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.

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

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.

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

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.

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

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:

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

Note that the last judgment is parametric in τ, a situation referred to as *parametric polymorphism.*