1.8 Dependent Types

We have encountered a number of constructors for propositions and types. Generally, propositions are constructed from simpler propositions, and types are constructed from simpler types. Furthermore, propositions refer to types (such as $\forall x{\in}\tau.\ A(x)$), and propositions refer to terms (such as $n =_N m$). However, we have not seen a type that refers to either a term or a proposition. In this section we consider the former. As we will see, allowing types to be constructed from terms has a number of applications, but it also creates a number of problems.

As an example we consider lists. Rather than simply keeping track of the types of their elements as we have done so far, we keep track of the length of the list as part of the type. We obtain the following formation rule:

$$\frac{\Gamma \vdash \tau\ \mathit{type} \qquad \Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash \tau\,\mathbf{list}(n)\ \mathit{type}}\,\mathbf{list}F$$

Note that we now make a context Γ explicit in this judgment, since the term n which occurs inside the type $\tau\,\mathbf{list}(n)$ may contain variables. We call $\tau\,\mathbf{list}$ a *type family* and n its *index term*.

The meaning of the type $\tau\,\mathbf{list}(n)$ is the type of lists with elements of type τ and length n. The introduction rules for this type track the length of the constructed list.

$$\frac{}{\Gamma \vdash \mathbf{nil}^\tau \in \tau\,\mathbf{list}(\mathbf{0})}\,\mathbf{list}I_n \qquad \frac{\Gamma \vdash s \in \tau \qquad \Gamma \vdash l \in \tau\,\mathbf{list}(n)}{\Gamma \vdash s :: l \in \tau\,\mathbf{list}(\mathbf{s}(n))}\,\mathbf{list}I_c$$

The elimination rule now must track the length of the list as well. Written as a schema of primitive recursion, we obtain

$$\begin{array}{lrcl} f & (\mathbf{0}, \mathbf{nil}) & = & s_n \\ f & (\mathbf{s}(n'), x :: l') & = & s_c(n', x, l', f(n', l')) \end{array}$$

where s_n contains no occurrence of f, and all occurrences of f in s_c have the indicated form of $f(n', l')$. Note that coupling occurrences of n and l in the schema guarantees that the typing remains consistent: even occurrence of $f(n, l)$ contains a list l in the second argument and its length in the first argument. Transforming this rule into an elimination rule yields

$$\frac{\begin{array}{l}\Gamma \vdash l \in \tau\,\mathbf{list}(n) \\ \Gamma \vdash s_n \in \sigma(\mathbf{0}, \mathbf{nil}) \\ \Gamma, n'{\in}\mathbf{nat}, x{\in}\tau, l'{\in}\tau\,\mathbf{list}(n'), f(n', l'){\in}\sigma(n', l') \vdash s_c \in \sigma(\mathbf{s}(n'), x :: l')\end{array}}{\Gamma \vdash (\mathbf{rec}\ l\ \mathbf{of}\ f(\mathbf{0}, \mathbf{nil}) \Rightarrow s_n \mid f(\mathbf{s}(n'), x :: l') \Rightarrow s_c) \in \sigma(n, l)}\,\mathbf{list}E$$

Here we have written the premises on top of each other for typographical reasons. There are two complications in this rule. The first is that we have to iterate over the lists and its length at the same time. The second is that now types may depend on terms. Therefore the type σ may actually depend on both n

and l, and this must be reflected in the rule. In fact, it looks very much like a rule of induction if we read the type $\sigma(n,l)$ as a proposition $A(n,l)$. Allowing types to depend on terms make types look even more like propositions than before. In fact, we are close to extending the Curry-Howard isomorphism from the propositional to the first-order case.

Next we consider how to use elements of this new type in some examples. The first is appending of two lists. We would like to say

$$\mathit{app} \in \tau\,\mathbf{list}(n) \to \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(n+m)$$

that is, *app* takes a list of length n and a list of length m and returns a list of length $n+m$. But what is the status of n and m in this declaration? We can see that at least n cannot be a global parameter (as τ, for example, since it changes during the recursion. Instead, we make it explicit in the type, using a new type constructor Π. This constructor acts on types exactly the way that $\forall$ acts on propositions. With it, we can write

$$\mathit{app} \in \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(n+m)$$

so that *app* is now a function of four arguments: a number n, a list of length n, a number m, and then a list of length m. The function returns a list of length $n+m$.

The rules for Π are constructed in complete analogy with $\forall$.

$$\frac{\Gamma \vdash \tau\ \mathit{type} \qquad \Gamma, x{\in}\tau \vdash \sigma(x)\ \mathit{type}}{\Gamma \vdash \Pi x{\in}\tau.\ \sigma(x)\ \mathit{type}}\,\Pi F$$

$$\frac{\Gamma, x{\in}\tau \vdash s \in \sigma(x)}{\Gamma \vdash \lambda x{\in}\tau.\ s \in \Pi x{\in}\tau.\ \sigma(x)}\,\Pi I \qquad \frac{\Gamma \vdash s \in \Pi x{\in}\tau.\ \sigma(x) \qquad \Gamma \vdash t \in \tau}{\Gamma \vdash s\,t \in \sigma(t)}\,\Pi E$$

$\Pi x \in \tau.\ \sigma(x)$ is called a *dependent function type*, because it denotes a function whose result type depends on the value of the argument. As for universal quantification, substitution is required in the elimination rule. With this in mind, we can first write and then type-check the specification of *app*.

$$\mathit{app} \in \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(n+m)$$

$$\begin{array}{llllll} \mathit{app} & \mathbf{0} & \mathbf{nil} & m & k & = & k \\ \mathit{app} & (\mathbf{s}(n')) & (x :: l') & m & k & = & x :: (\mathit{app}\ n'\ l'\ m\ k) \end{array}$$

For each equation in this specification we type-check both the left- and the right-hand side and verify that they are the same. We show the checking of subterm to help the understanding of the type-checking process. First, the left-hand side of the first equation.

$$\begin{array}{lll} \mathit{app}\ \mathbf{0} & \in & \tau\,\mathbf{list}(\mathbf{0}) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(\mathbf{0}+m) \\ \mathit{app}\ \mathbf{0}\ \mathbf{nil} & \in & \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(\mathbf{0}+m) \\ \mathit{app}\ \mathbf{0}\ \mathbf{nil}\ m & \in & \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(\mathbf{0}+m) \qquad \text{for } m{\in}\mathbf{nat} \\ \mathit{app}\ \mathbf{0}\ \mathbf{nil}\ m\ k & \in & \tau\,\mathbf{list}(\mathbf{0}+m) \qquad \text{for } k{\in}\tau\,\mathbf{list}(m) \\ \\ k & \in & \tau\,\mathbf{list}(m) \end{array}$$

While the two types are different, the first one can be reduced to the second. Just like previously for propositions, we therefore need rules of computation for types.

$$\frac{\Gamma \vdash s : \sigma \qquad \sigma \Longrightarrow \sigma' \qquad \Gamma \vdash \sigma \; \mathit{type}}{\Gamma \vdash s : \sigma'} \; \mathit{conv}$$

$$\frac{\Gamma \vdash s : \sigma' \qquad \sigma \Longrightarrow \sigma' \qquad \Gamma \vdash \sigma \; \mathit{type}}{\Gamma \vdash s : \sigma} \; \mathit{conv}'$$

Next, we consider the second equation, first the left-hand and then the right-hand side.

$$\begin{array}{lcl}
\mathit{app}\ (\mathbf{s}(n')) & \in & \tau\,\mathbf{list}(\mathbf{s}(n')) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(\mathbf{s}(n') + m) \\
 & & \qquad \text{for } n' \in \mathbf{nat} \\
\mathit{app}\ (\mathbf{s}(n'))\ (x :: l') & \in & \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(\mathbf{s}(n') + m) \\
 & & \qquad \text{for } x \in \tau \text{ and } l' \in \tau\,\mathbf{list}(n') \\
\mathit{app}\ (\mathbf{s}(n'))\ (x :: l')\ m & \in & \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(\mathbf{s}(n') + m) \qquad \text{for } m{\in}\mathbf{nat} \\
\mathit{app}\ (\mathbf{s}(n'))\ (x :: l')\ m\ k & \in & \tau\,\mathbf{list}(\mathbf{s}(n') + m) \qquad \text{for } k{\in}\tau\,\mathbf{list}(m) \\
 & & \\
\mathit{app}\ n'\ l'\ m\ k & \in & \tau\,\mathbf{list}(n' + m) \\
x :: (\mathit{app}\ n'\ l'\ m\ k) & \in & \tau\,\mathbf{list}(\mathbf{s}(n' + m))
\end{array}$$

Again, we can obtain the right-hand side by computation from the left-hand side

$$\tau\,\mathbf{list}(\mathbf{s}(n') + m) \Longrightarrow \tau\,\mathbf{list}(\mathbf{s}(n' + m))$$

since addition is defined by primitive recursion over the first argument.

For the sake of completeness, we now show an explicit definition of *app* by primitive recursion.

$$\begin{array}{lcl}
\mathit{app} & = & \lambda n{\in}\mathbf{nat}.\ \lambda l{\in}\tau\,\mathbf{list}(n). \\
 & & \quad \mathbf{rec}\ l \\
 & & \quad\ \mathbf{of}\ f(\mathbf{0}, \mathbf{nil}) \Rightarrow \lambda m{\in}\mathbf{nat}.\ \lambda k{\in}\tau\,\mathbf{list}(m).\ k \\
 & & \quad\quad \mid\ f(\mathbf{s}(n'), x :: l') \Rightarrow \lambda m{\in}\mathbf{nat}.\ \lambda k{\in}\tau\,\mathbf{list}(m).\ x :: (f(n', l')\ m\ k)
\end{array}$$

From the practical point of view, we would like to avoid passing the lengths of the lists as arguments to *app*. In the end, we are interested in the list as a result, and not its length. In order to capture this, we extend the erasure notation $[A]$ and $[M]$ from propositions and proof terms to types $[\tau]$ and terms $[t]$. The meaning is completely analogous. Since we don't want to pass length information, we obtain

$$\mathit{app} \in \Pi[n]{\in}[\mathbf{nat}].\ \tau\,\mathbf{list}[n] \to \Pi[m]{\in}[\mathbf{nat}].\ \tau\,\mathbf{list}[m] \to \tau\,\mathbf{list}[n + m]$$

$$\begin{array}{lcccccl}
\mathit{app} & [\mathbf{0}] & \mathbf{nil} & [m] & k & = & k \\
\mathit{app} & [\mathbf{s}(n')] & (x :: l') & [m] & k & = & x :: (\mathit{app}\ [n']\ l'\ [m]\ k)
\end{array}$$

Fortunately, this annotation is consistent: we never use a bracketed variable outside of brackets. That is, we never try to construct an answer out of a

variable that will not be carried at runtime. After erasure of the bracketed terms and types and simplification, we obtain the prior definition of *app* on lists that are not indexed by their length.

But not every function can be consistently annotated. As a simple counterexample consider the following length function:

$$\begin{array}{rrcl} & \mathit{length} & \in & \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \mathbf{nat} \\ \mathit{length} & n\ \ l & = & n \end{array}$$

This is a perfectly valid implementation of *length*: from type-checking we know that l must have length n. However, if we try to annotate this function

$$\begin{array}{rrcl} & \mathit{length} & \in & \Pi[n]{\in}[\mathbf{nat}].\ \tau\,\mathbf{list}[n] \to \mathbf{nat} \\ \mathit{length} & [n]\ \ l & = & n \end{array}$$

we observe a use of n outside of brackets which is illegal. Indeed, if n is not passed at run-time, then we cannot "compute" the length in this way. Fortunately, there is another obvious definition of length that can be annotated in the desired way.

$$\begin{array}{rrcl} \mathit{length}\ \ [\mathbf{0}]\ \ \mathbf{nil} & & = & \mathbf{0} \\ \mathit{length}\ \ [\mathbf{s}(n')]\ \ (x :: l') & & = & \mathbf{s}(\mathit{length}\ [n']\ l') \end{array}$$

which has the property that the bracketed variable n' from the left-hand side also occurs only bracketed on the right-hand side. Note that dependent type-checking does *not* verify the correctness of this second implementation of *length* in the sense that the type does not exhibit a connection between the length argument n and the natural number that is returned.

The use of dependent types goes very smoothly for the examples above, but what happens when the length of an output list to a function is unknown? Consider the *filter* function which retains only those elements of a list that satisfy a given predicate p. We first give the definition with the ordinary lists not indexed by their length.

$$\begin{array}{rrrcl} & & \mathit{filter} & \in & (\tau \to \mathbf{bool}) \to \tau\,\mathbf{list} \to \tau\,\mathbf{list} \\ \mathit{filter} & p & \mathbf{nil} & = & \mathbf{nil} \\ \mathit{filter} & p & (x :: l') & = & \mathbf{if}\ p\,x \\ & & & & \mathbf{then}\ \ x :: \mathit{filter}\ p\ l' \\ & & & & \mathbf{else}\ \ \mathit{filter}\ p\ l' \end{array}$$

There is no type of the form

$$\mathit{filter} \quad \in \quad (\tau \to \mathbf{bool}) \to \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \tau\,\mathbf{list}(?)$$

we can assign to *filter*, since the length of the result depends on p and the length of the input. For this we need an *existential type*. It works analogously to the existential quantifier on propositions and is written as $\Sigma x{\in}\tau.\ \sigma(x)$. With it, we would specify the type as

$$\mathit{filter} \quad \in \quad (\tau \to \mathbf{bool}) \to \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \Sigma m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m)$$

We can read this as "*the function returns a list of length m for some m*" or as "*the function returns a pair consisting of an m and a list of length m*", depending on whether we intend to carry the lengths are runtime. Before we show the specification of *filter* with this new type, we give the rules for Σ types.

$$\frac{\Gamma \vdash \tau \ \mathit{type} \qquad \Gamma, x{\in}\tau \vdash \sigma(x) \ \mathit{type}}{\Gamma \vdash \Sigma x{\in}\tau.\ \sigma(x)}\,\Sigma F$$

$$\frac{\Gamma \vdash t \in \tau \qquad \Gamma \vdash s \in \sigma(t)}{\Gamma \vdash \langle t, s\rangle \in \Sigma x{\in}\tau.\ \sigma(x)}\,\Sigma I$$

$$\frac{\Gamma \vdash t \in \Sigma x{\in}\tau.\ \sigma(x) \qquad \Gamma, x{\in}\tau, y{\in}\sigma(x) \vdash r \in \rho}{\Gamma \vdash (\mathbf{let}\ \langle x, y\rangle = t\ \mathbf{in}\ r) \in \rho}\,\Sigma E$$

If we read $\sigma(x)$ as a proposition $A(x)$ instead of a type, we obtain the usual rules for the existential quantifier. Returning to the function *filter*, we have

$$\mathit{filter} \in (\tau \to \mathbf{bool}) \to \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \Sigma m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m)$$

$$\begin{array}{llrrcl}
\mathit{filter} & p & \mathbf{0} & \mathbf{nil} & = & \langle \mathbf{0}, \mathbf{nil}\rangle \\
\mathit{filter} & p & (\mathbf{s}(n')) & (x :: l') & = & \mathbf{let}\ \langle m', k'\rangle = \mathit{filter}\ p\ n'\ l' \\
 & & & & & \mathbf{in}\ \mathbf{if}\ p\,x \\
 & & & & & \quad \mathbf{then}\ \langle \mathbf{s}(m'), x :: k'\rangle \\
 & & & & & \quad \mathbf{else}\ \langle m', k'\rangle
\end{array}$$

In this code, k' stands for the list resulting from the recursive call, and m' for its length. Now type-checking succeeds, since each branch in each case has type $\Sigma m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m)$. Again, we can annotate the type and implementation to erase the part of the code which is not computationally relevant.

$$\mathit{filter} \in (\tau \to \mathbf{bool}) \to \Pi[n]{\in}[\mathbf{nat}].\ \tau\,\mathbf{list}[n] \to \Sigma[m]{\in}[\mathbf{nat}].\ \tau\,\mathbf{list}[m]$$

$$\begin{array}{llrrcl}
\mathit{filter} & p & [\mathbf{0}] & \mathbf{nil} & = & \langle [\mathbf{0}], \mathbf{nil}\rangle \\
\mathit{filter} & p & [\mathbf{s}(n')] & (x :: l') & = & \mathbf{let}\ \langle [m'], k'\rangle = \mathit{filter}\ p\ [n']\ l' \\
 & & & & & \mathbf{in}\ \mathbf{if}\ p\,x \\
 & & & & & \quad \mathbf{then}\ \langle [\mathbf{s}(m')], x :: k'\rangle \\
 & & & & & \quad \mathbf{else}\ \langle [m'], k'\rangle
\end{array}$$

This annotation is consistent, and erasure followed by simplification produces the previous version of *filter* with lists not carrying their length.

Existential types solve a number of potential problems, but they incur a loss of information which may render dependent type-checking less useful than it might first appear. Recall the function *rev*, the generalized version of *reverse* carrying an accumulator argument a.

$$\mathit{rev} \in \tau\,\mathbf{list} \to \tau\,\mathbf{list} \to \tau\,\mathbf{list}$$

$$\begin{array}{llrcl}
\mathit{rev} & \mathbf{nil} & a & = & a \\
\mathit{rev} & (x :: l') & a & = & \mathit{rev}\ l'\ (x :: a)
\end{array}$$

We would like to verify that

$$rev \quad \in \quad \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \tau\,\mathbf{list}(n+m)$$

where

$$\begin{array}{llllllll} rev & \mathbf{0} & \mathbf{nil} & m & a & = & a \\ rev & (\mathbf{s}(n')) & (x :: l') & m & a & = & rev\ n'\ l'\ (\mathbf{s}(m))\ (x :: a) \end{array}$$

While everything goes according to plan for the first equation, the second equation yields

$$rev\ (\mathbf{s}(n'))\ (x :: l')\ m\ a \quad \in \quad \tau\,\mathbf{list}(\mathbf{s}(n') + m)$$

for the left-hand side, and

$$rev\ n'\ l'\ (\mathbf{s}(m))\ (x :: a) \quad \in \quad \tau\,\mathbf{list}(n' + \mathbf{s}(m))$$

for the right hand side. There is no way to bridge this gap by comptutation alone; we need to *prove* that $\mathbf{s}(n')+m =_N n'+\mathbf{s}(m)$ by induction. Clearly, type-checking can not accomplish this—it would require type-checking to perform theorem proving which would not be feasible inside a compiler.

What can we do? One option is the simply hide the length of the output list by using an existential type.

$$rev \quad \in \quad \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \to \Sigma x{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(x)$$

However, this means type-checking guarantees much less about our function than we might hope for. The other is to reintroduce propositions and change our type to something like

$$\begin{array}{rcl} rev & \in & \Pi n{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(n) \to \Pi m{\in}\mathbf{nat}.\ \tau\,\mathbf{list}(m) \\ & & \quad \to \Sigma x{\in}\mathbf{nat}.\ [x =_N n+m] \times \tau\,\mathbf{list}(x). \end{array}$$

That is, we allow the output to be list of length x which is provably, but not necessarily computationally equal to the sum of n and m. Here we consider $[x =_N n + m]$ as a type, even though $x =_N n + m$ is a proposition. This is consistent with our interpretation of erasure, which converts propositions to types before running a program.

As a practical matter, in extensions of programming language with some limited form of dependent types, there are other ways to ensure feasibility of type-checking. Rather than base the comparison of types entirely on computation of the terms embedded in them, we can base it instead on any decidable theory (which is feasible in practice). This is the approach we have taken in the design of DML [**?**, **?**]. In the simplest application, index objects may contain only linear equalities and inequalities between integers, which can be solved effectively during type-checking. As we have seen in the examples above, dependent types (especially when we can also mention propositions $[A]$) permit a continuum of properties of programs to be expressed and verified at type-checking time, all the way from simple types to full specifications. For the

latter, the proof objects either have to be expressed directly in the program or extracted as obligations and verified separately.

We now briefly reexamine the Curry-Howard isomorphism, when extended to the first-order level. We have the following correspondence:

Propositions	$\wedge$	$\supset$	$\top$	$\vee$	$\bot$	$\forall$	$\exists$
Types	$\times$	$\rightarrow$	$\mathbf{1}$	$+$	$\mathbf{0}$	Π	Σ

Note that under erasure, $\forall$ is related to $\rightarrow$ and $\exists$ is related to $\times$. The analogous property holds for Π and Σ: $\Pi x{:}\tau.\ \sigma$ corresponds to $\tau \rightarrow \sigma$ if x does not occur in σ, and $\Sigma x{:}\tau.\ \sigma$ simplifies to $\tau \times \sigma$ if x does not occur in σ.

In view of this strong correspondence, one wonders if propositions are really necessary as a primitive concept. In some systems, they are introduced in order to distinguish those elements with computational contents from those without. However, we have introduced the bracket annotation to serve this purpose, so one can streamline and simplify type theory by eliminating the distinction between propositions and types. Similarly, there is no need to distinguish between terms and proof terms. In fact, we have already used identical notations for them. Propositional constructs such as $n =_N m$ are then considered as types (namely: the types of their proof terms).

Because of the central importance of types and their properties in the design and theory of programming languages, there are many other constructions that are considered both in the literature and in practical languages. Just to name some of them, we have polymorphic types, singleton types, intersection types, union types, subtypes, record types, quotient types, equality types, inductive types, recursive types, linear types, strict types, modal types, temporal types, etc. Because of the essentially open-ended nature of type theory, all of these could be considered in the context of the machinery we have built up so far. We have seen most of the principles which underly the design of type systems (or corresponding logics), thereby providing a foundation for understanding the vast literature on the subject.

Instead of discussing these (which could be subject of another course) we consider one further application of dependent types and then consider theorem proving in various fragments of the full type theory.

1.9 Data Structure Invariants

An important application of dependent types is capturing representation invariants of data structures. An invariant on a data structure restricts valid elements of a type. Dependent types can capture such invariants, so that only valid elements are well-typed.

Our example will be an efficient implementation of finite sets of natural numbers. We start with a required lemma and auxiliary function.

$$\forall x{\in}\mathbf{nat}.\ \forall y{\in}\mathbf{nat}.\ [x < y] \vee [x =_N y] \vee [x > y]$$