Dependent Types, the λ Cube, and Proof Assistants

Stefan Muller

CSE 5095: Types and Programming Languages, Fall 2025
Lecture 25

1 Dependent Types

Remember our type of integer lists $\mathsf{intlist}$, and the hd and tl functions:

$$\begin{array}{lcl} \mathsf{hd} & : & \mathsf{intlist} \to \mathsf{unit} + \mathsf{int} \\ \mathsf{tl} & : & \mathsf{intlist} \to \mathsf{unit} + \mathsf{intlist} \end{array}$$

The functions need to return an option (a sum of an actual value or ()) because they fail if the list is empty. Can we make the types more specific to avoid this kind of failure? Maybe, like we did on the homework with non-zero integers, we can have a type of non-empty lists:

$$\begin{array}{lcl} \mathsf{hd} & : & \mathsf{neintlist} \to \mathsf{int} \\ \mathsf{tl} & : & \mathsf{neintlist} \to \mathsf{intlist} \end{array}$$

This works as far as it goes, but $\mathsf{hd}(\mathsf{tl}\ l)$ is still a problem, since tl may return an empty list. To be fully precise, we want the type of lists to say how many elements it has. Suppose we have the type $\mathsf{intlist}[e]$, where e is an expression of type nat.

$$\tau \quad ::= \quad \cdots \mid \Pi x : \tau.\tau \mid \Sigma x : \tau.\tau$$

The type $\Pi x : \tau_1.\tau_2$ (called the *dependent function* or *dependent product*) indicates a function that takes an x of type τ_1 and returns something of type τ_2, where, unlike in the normal type $\tau_1 \to \tau_2$, x is bound in τ_2. (Note that $\tau_1 \to \tau_2$ is now just a special case of $\Pi x : \tau_1.\tau_2$ where x doesn't appear in τ_2.)

This is a little less useful for us right now, but $\Sigma x : \tau_1.\tau_2$ is a *dependent pair* or *dependent sum.* Its canonical form is a pair whose second component's type can depend on the value of the first component.

Note that the type formation judgment now needs a variable context (if our language still has polymorphic and/or recursive types, it needs a type variable context too—more on this later!).

$$\frac{\Gamma \vdash \tau_1\ \mathsf{ok} \qquad \Gamma, x : \tau \vdash \tau_2\ \mathsf{ok}}{\Gamma \vdash \Pi x : \tau_1.\tau_2\ \mathsf{ok}}\ (\text{OKPi}) \qquad \frac{\Gamma \vdash \tau_1\ \mathsf{ok} \qquad \Gamma, x : \tau \vdash \tau_2\ \mathsf{ok}}{\Gamma \vdash \Sigma x : \tau_1.\tau_2\ \mathsf{ok}}\ (\text{OKSigma})$$

$$\frac{\Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash \mathsf{intlist}[e]\ \mathsf{ok}}\ (\text{OKList}) \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\tau_1 e : \Pi x : \tau_1.\tau_2}\ (\Pi\text{-I}) \qquad \frac{\Gamma \vdash e_1 : \Pi x : \tau_1.\tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : [e_2/x]\tau_2}\ (\Pi\text{-E})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \Gamma \vdash e_2 : [e_1/x]\tau_2}{\Gamma \vdash (e_1, e_2) : \Sigma x : \tau_1.\tau_2}\ (\Sigma\text{-I}) \qquad \frac{\Gamma \vdash e : \Sigma x : \tau_1.\tau_2}{\Gamma \vdash \mathsf{fst}\ e : \tau_1}\ (\Sigma - E_1) \qquad \frac{\Gamma \vdash e : \Sigma x : \tau_1.\tau_2}{\Gamma \vdash \mathsf{snd}\ e : [\mathsf{fst}\ e/x]\tau_2}\ (\Sigma - E_2)$$

Yes, we now need a lemma about substituting expressions into types! And all our lemmas get a lot harder because our type-OK rules have expression typing rules as premises and maybe vice versa! So let's not worry about proving any of them today.

$$\begin{array}{lcl} \mathsf{nil} & : & \mathsf{intlist}[\mathsf{zero}] \\ \mathsf{cons} & : & \Pi n : \mathsf{nat}.\mathsf{int} \to \mathsf{intlist}[n] \to \mathsf{intlist}[\mathsf{succ}\ n] \\ \mathsf{hd} & : & \Pi n : \mathsf{nat}.\mathsf{intlist}[\mathsf{succ}\ n] \to \mathsf{int} \\ \mathsf{tl} & : & \Pi n : \mathsf{nat}.\mathsf{intlist}[\mathsf{succ}\ n] \to \mathsf{intlist}[n] \end{array}$$

The elimination form for types like $\mathsf{intlist}[n]$ is a pattern match, like case, but with a branch for every constructor, in this case nil and cons.

$$\begin{array}{lcl} \mathsf{hd} & \triangleq & \lambda n : \mathsf{nat}.\lambda l : \mathsf{intlist}[\mathsf{succ}\ n].\mathsf{case}\ l\ \mathsf{of}\ \{\mathsf{cons}\ m\ h\ t.\ h\} \\ \mathsf{tl} & \triangleq & \lambda n : \mathsf{nat}.\lambda l : \mathsf{intlist}[\mathsf{succ}\ n].\mathsf{case}\ l\ \mathsf{of}\ \{\mathsf{cons}\ m\ h\ t.\ t\} \end{array}$$

I just said we need a branch for every constructor, so why did we only have one? Clearly, nil isn't a possible constructor for $\mathsf{intlist}[\mathsf{succ}\ n]$ (because zero isn't $\mathsf{succ}\ n$ for any n), so we need not consider it.

We can also define the type of natural numbers less than n:

$$\begin{array}{lcl} \mathsf{z} & : & \mathsf{fin}[\mathsf{succ\ zero}] \\ \mathsf{s} & : & \Pi n : \mathsf{nat}.\mathsf{fin}[n] \to \mathsf{fin}[\mathsf{succ}\ n] \end{array}$$

Now we can define a lookup function that always succeeds!

$$\begin{array}{lcl} \mathsf{lookup} & : & \Pi n : \mathsf{nat}.\mathsf{intlist}[n] \to \mathsf{fin}[n] \to \mathsf{int} \\ \mathsf{lookup} & \triangleq & \mathsf{fix\ lookup} = \lambda n : \mathsf{nat}.\lambda l : \mathsf{intlist}[n].\lambda m : \mathsf{fin}[n]. \\ & & \quad \mathsf{case}\ m\ \mathsf{of}\ \{\mathsf{z}.\ \mathsf{case}\ l\ \mathsf{of}\ \{\mathsf{cons}\ \_\ h\ \_.\ h\}; \mathsf{s}\ m'.\mathsf{case}\ l\ \mathsf{of}\ \{\mathsf{cons}\ n'\ \_\ t.\ \mathsf{lookup}\ n'\ t\ m'\}\} \end{array}$$

As another note, where does the type $\mathsf{intlist}[e]$ come from? Rather than build it up, e.g., with recursive types and sums as we did with int lists before, when working with dependent types, we often just assert the existence of these types and define them with a set of constructors (like nil and cons above, as the ways of building int lists).

2 The λ Cube

When we added polymorphism to STLC, we added $\Lambda\alpha.e$, functions from types to expressions. Above, we added functions from expressions to types.

We've also seen, without really knowing it, a use case for functions from types to *types*. We've talked a lot today about *integer* lists, but we probably want lists of any type, e.g., for any α: α list But we have no way of binding α in this way.

Q: Why would $\forall\alpha.\alpha$ list not be what we want?

What we want is a function *at the level of types* $\lambda\alpha.\tau$. What is such a thing? It's not a type in that it doesn't make sense to talk about whether an expression has it as a type. It's a *type constructor*, for which we use the metavariable c. This leads to a whole new level of our syntax hierarchy: *kinds* are like types of types. They say whether a type constructor is a type or a function. Types are a subset of type constructors and we can use τ for constructors of kind Ty.

$$\begin{array}{lcl} \kappa & ::= & \mathsf{Ty} \mid \kappa \to \kappa \\ c & ::= & \mathsf{unit} \mid \tau \to \tau \mid \cdots \mid \lambda\alpha : \kappa.c \end{array}$$

When we combine this with dependent types, we can get types like below:

$$\begin{array}{lcl} \mathsf{nil} & : & \lambda\alpha : \mathsf{Ty}.\alpha\ \mathsf{list}[\mathsf{zero}] \\ \mathsf{cons} & : & \lambda\alpha : \mathsf{Ty}.\Pi n : \mathsf{nat}.\mathsf{int} \to \alpha\ \mathsf{list}[n] \to \alpha\ \mathsf{list}[\mathsf{succ}\ n] \\ \mathsf{hd} & : & \lambda\alpha : \mathsf{Ty}.\Pi n : \mathsf{nat}.\alpha\ \mathsf{list}[\mathsf{succ}\ n] \to \mathsf{int} \\ \mathsf{tl} & : & \lambda\alpha : \mathsf{Ty}.\Pi n : \mathsf{nat}.\alpha\ \mathsf{intlist}[\mathsf{succ}\ n] \to \alpha\ \mathsf{list}[n] \end{array}$$

At this point, we're tempted to remove the distinction between expressions and types entirely and just allow Π bindings to bind types or expressions. Indeed, that's what calculi of this form often do.

This combination of different features gives rise to the *lambda cube*, formulated in this way by Henk Barendregt. The three axes correspond to whether we include functions from types to expressions (polymorphism), types to types (higher kinds), and/or expressions to types (dependent types).

$$\begin{tikzcd}
 & F_\omega \arrow[rr, dash] \arrow[dl, dash] \arrow[dd, dash] & & \mathrm{CoC} \arrow[dl, dash] \arrow[dd, dash] \\
F \arrow[rr, dash] \arrow[dd, dash] & & \bullet \arrow[dd, dash] & \\
 & \bullet \arrow[rr, dash] \arrow[dl, dash] & & \bullet \arrow[dl, dash] \\
\mathrm{STLC} \arrow[rr, dash] & & \lambda_\Pi &
\end{tikzcd}$$

All of the corners of the cube are type systems; several are notable ones, some of which we've seen:

- STLC is, of course, the simply typed lambda calculus (with just functions and possibly base types to make it non-degenerate).
- System F adds polymorphism to STLC.
- F_ω is a well-known calculus that adds higher kinds to System F. It's a decently powerful model that corresponds reasonably closely to a core model of real functional programming languages.
- λ_Π is the language we consider in the section above with just STLC plus dependent types.
- CoC, the Calculus of Constructions, adds all of these forms of functions and is an extremely powerful type system that forms the foundation of some dependently typed languages and proof assistants (we revisit this connection below), notably Rocq.

3 Proof Assistants

Agda is a dependently typed programming language (its type theory isn't technically CoC, but it's close enough for our purposes today). We can use it to define the intlist type and associated functions from above. It's most interesting though when we view dependent types through the Curry-Howard correspondence:

$$\begin{array}{ccc} \Pi x : \tau_1.\tau_2 & \text{becomes} & \forall x : \tau_1.\tau_2 \\ \Sigma x : \tau_1.\tau_2 & \text{becomes} & \exists x : \tau_1.\tau_2 \end{array}$$

Let's take the property "for all x of type nat, there exists y such that $x * 2 = y$." We can express this as a type:

$$\Pi x : \mathsf{nat}.\Sigma y : \mathsf{nat} x * 2 = y.$$

The proof of this is a function that returns a proof for any x. In each case, the proof of the "there exists" statement is a pair of the actual y and a proof that y meets the requirement.