Dependent types

Paul Stansifer

March 16, 2012

1 You've seen this before

I hope you like abstraction, because we're about to use it a lot. Here's the simply-typed lambda calculus with built-in list operations and one minor, completely unmotivated change to the type notation:

$$\begin{array}{rcl} e & ::= & ee \mid \lambda x{:}\tau.e \mid x \mid n \mid \textbf{cons}\ e\ e \mid \textbf{car}\ e \mid \textbf{cdr}\ e \mid \text{nil} \\ v & ::= & \lambda x{:}\tau.e \mid n \mid \textbf{cons}\ v\ v \mid \text{nil} \\ \tau & ::= & \Pi x{:}\tau.\tau \mid \text{Nat} \mid \textbf{ListNat} \end{array}$$

That $\Pi x{:}\tau.\tau$ is just $\tau \to \tau$. Pay no attention to the x behind the curtain! We have the standard type rules:

$$\frac{\Gamma, x{:}\tau_0 \vdash e : \tau_1}{\Gamma \vdash \lambda x{:}\tau_0.e : \Pi x{:}\tau_0.\tau_1}\ \text{ST}\lambda\text{C-Abs}$$

$$\frac{\Gamma \vdash e_0 : \Pi x{:}\tau_0.\tau_1 \qquad \Gamma \vdash e_1 : \tau_0}{\Gamma \vdash e_0 e_1 : \tau_1}\ \text{ST}\lambda\text{C-App}$$

Let's assume that the rest of the rules are trivial, and make an operational semantics:

$$\textbf{cdr}\ \textbf{cons}\ v_1\ v_2 \to v_1$$

$$\textbf{cdr}\ \text{nil} \to \text{uh...}$$

Some people deal with this problem by adding errors to the language. We call them weaklings. We're going to make **cdr** nil ill-typed. A "vector", we'll say, is a list with known length.

$$\begin{array}{rcl} e & ::= & ee \mid \lambda x{:}\tau.e \mid x \mid n \mid \textbf{cons}_{\ell}\ e\ e \mid \textbf{car}_{\ell}\ e \mid \textbf{cdr}_{\ell}\ e \mid \text{nil} \\ v & ::= & \lambda x{:}\tau.e \mid n \mid \textbf{cons}_{\ell}\ v\ v \mid \text{nil} \\ \tau & ::= & \Pi x{:}\tau.\tau \mid \text{Nat} \mid \textbf{VecNat}\ \ell \\ \ell & ::= & 0 \mid \textbf{succ}\ \ell \end{array}$$

$$\frac{\Gamma \vdash e_{\text{lst}} : \textbf{VecNat succ}\ \ell}{\Gamma \vdash \textbf{car}_{\ell}\ e_{\text{lst}} : \text{Nat}}\ \text{T-Car}$$

$$\frac{\Gamma \vdash e_{\text{lst}} : \textbf{VecNat succ}\ \ell}{\Gamma \vdash \textbf{cdr}_{\ell}\ e_{\text{lst}} : \textbf{VecNat}\ \ell}\ \text{T-Cdr}$$

$$\frac{\Gamma \vdash x : \text{Nat} \qquad \Gamma \vdash e_{\text{lst}} : \textbf{VecNat}\ n}{\Gamma \vdash \textbf{cons}_{\ell}\ e_{\text{elt}}\ e_{\text{lst}} : \textbf{VecNat succ}\ n}\ \text{T-Cons}$$

$$\frac{}{\Gamma \vdash \text{nil} : \textbf{VecNat}\ 0}\ \text{T-Nil}$$

(By the way, you should think of those subscripted things as being perfectly normal arguments. We're writing them that way because, in some cases, they could be inferred, saving us a certain amount of pain.)

Now, our types prevent us from taking the tail of an empty list. But the broader situation seems very bad. Not only have we built lists into the language, but we're going to have to do the same for future list operations. And operations like **filter** are probably going to require re-vamping the whole type system.

None of these things are going to be easy. But with dependent types, at least they can be possible.

2 Bring out the big guns

So far, we've talked about type systems that are, in some sense, utterly insensitive to terms. Sure, the structure of a term guides typechecking (or even type synthesis), but once we've typed a term, the only information the type system sees about it is its type. But that means that we have to go into increasingly desparate lengths to encode information into types. Dependent types exist to remove that restriction. It is now possible to statically look into terms! This power comes at a cost, of course.

First, let's fold ℓ into the syntax for expressions:

$$\begin{array}{rcl} e & ::= & ee \mid \lambda x{:}\tau.e \mid x \mid n \mid \mathbf{cons}\ e\ e \mid \mathbf{car}\ e \mid \mathbf{cdr}\ e \mid \text{nil} \mid 0 \mid \mathbf{succ}\ e \\ \tau & ::= & \Pi x{:}\tau.\tau \mid \text{Nat} \mid \alpha \mid \tau\ e \end{array}$$

Because I want to avoid talking about kindedness, I won't talk much about α, except to say that **VecNat** and Length are both αs, and that using them in the expected way is well-kinded.

Now, let's update our types so that they talk about es instead of ℓs:

$$\frac{\Gamma \vdash e : \text{Length} \qquad \Gamma \vdash e_{\text{lst}} : \mathbf{VecNat}\ \mathbf{succ}\ e}{\Gamma \vdash \mathbf{car}_e\ e_{\text{lst}} : \text{Nat}}\ \text{T-Car}$$

$$\frac{\Gamma \vdash e : \text{Length} \qquad \Gamma \vdash e_{\text{lst}} : \mathbf{VecNat}\ \mathbf{succ}\ e}{\Gamma \vdash \mathbf{cdr}_e\ e_{\text{lst}} : \mathbf{VecNat}\ e}\ \text{T-Cdr}$$

$$\frac{\Gamma \vdash e : \text{Length} \qquad \Gamma \vdash x : \text{Nat} \qquad \Gamma \vdash e_{\text{lst}} : \mathbf{VecNat}\ n}{\Gamma \vdash \mathbf{cons}_e\ e_{\text{elt}}\ e_{\text{lst}} : \mathbf{VecNat}\ \mathbf{succ}\ n}\ \text{T-Cons}$$

$$\frac{}{\Gamma \vdash \text{nil} : \mathbf{VecNat}\ 0}\ \text{T-Nil}$$

Oh, let's express the kindedness restriction on our αs, and well-typedness for lengths:

$$\frac{}{\Gamma \vdash \text{Length} :: *}\ \text{K-Length}$$

$$\frac{\Gamma \vdash e : \text{Length}}{\Gamma \vdash \mathbf{VecNat}\ e :: *}\ \text{K-VecNat}$$

$$\frac{}{\Gamma \vdash 0 : \text{Length}}\ \text{T-Zero}$$

$$\frac{\Gamma \vdash e : \text{Length}}{\Gamma \vdash \mathbf{succ}\ e : \text{Length}}\ \text{T-Succ}$$

Now it's easy to construct a proof tree showing that that $\mathbf{cons}_8\ x\ (\mathbf{cdr}_8\ xs)$ has the same length as xs, under the assumptions (presumably provided by Γ) that x is a natural number, and xs doesn't at least have one element.

Okay, but these things that should be functions (**cons** and **cdr**, etc.) are still built in to the language. That's where the Πs come in. With them, people will be able to effectively write types as inference rules. The abstraction rule will be pretty much unchanged (but note that τ_1 will now be able to reference x):

$$\frac{\Gamma, x{:}\tau_0 \vdash e : \tau_1 \qquad \Gamma \vdash \tau_0 :: *}{\Gamma \vdash \lambda x{:}\tau_0.e : \Pi x{:}\tau_0.\tau_1}\ \textsc{Dep-Abs}$$

Read $\Gamma \vdash \tau_0 :: *$ as "τ_0 is a well-formed type.". It doesn't have to be inhabited/true. We need this because, now that we care about the names that are in our types, we want any $x$es in our newly introduced type to be bound. As usual, abstraction takes proofs of τ_0 to proofs of τ_1.

In the application rule, you can kinda see the function's Π type being looked up, and then unpacked so that the τ_0 is on the top and the τ_1 is on the bottom:

$$\frac{\Gamma \vdash e_0 : \Pi x{:}\tau_0.\tau_1 \qquad \Gamma \vdash e_1 : \tau_0}{\Gamma \vdash e_0\ e_1 : [x \mapsto e_1]\tau_1}\ \textsc{Dep-App}$$

Now, if you wrote:

$$\mathbf{cdr} : \Pi e{:}\text{Length}.\ (\Pi xs{:}\mathbf{VecNat\ succ}\ e.\mathbf{VecNat}\ e)$$

... it would mean (modulo currying) the same thing as the rule above:

$$\frac{\Gamma \vdash e : \text{Length} \qquad \Gamma \vdash xs : \mathbf{VecNat\ succ}\ e}{\Gamma \vdash \mathbf{cdr}_e\ e_{\text{lst}} : \mathbf{VecNat}\ e}\ \textsc{Def-Cdr}$$

It used to be that ℓ was a special form, and you couldn't write code that worked on vectors of different length. Now it's a term, and we can abstract over terms, so we can do it now.

We'll continue writing function types as inference rules, just know that, henceforth they'll be inference rules *provided by the user*.

2.1 Type comparison

One of the nice things about types is that it's easy to compare them to each other. Unless you have subtyping, you'd typically just look at them syntactically. But you can't do this with terms! Surely, you'd agree that $(\lambda x.x)5$ and 5 are equivalent terms? It'd be a shame to reject a program because it used the former instead of the later. In fact, it'd sort of prevent you from programming. We need to be able to compare terms.

But wait! Aren't we in a strongly-normalizing language? Comparing things is what normalizing is good for, so all we need to do is normalize the things we wish to compare, and then check to see if they're α-equivalent. For now, this solution is good enough.

3 Typechecker needs proofs badly

Here's the definition of the function that appends lists of natural numbers (it's time to mention that I'm using bold to mean "is an abstraction over terms"):

$$\underline{\text{let}}\ \frac{m : \text{Length} \qquad n : \text{Length} \qquad xs : \mathbf{VecNat}\ m \qquad ys : \mathbf{VecNat}\ n}{\mathbf{append}_{m,n}\ xs\ ys : \mathbf{VecNat}\ (\mathbf{sum}\ m\ n)} \quad =$$

$$\begin{array}{rcl} \mathbf{append}_{0,n}\ \text{nil}\ ys & \Rightarrow & ys \\ \mathbf{append}_{\mathbf{succ}\ m',n}\ (\mathbf{cons}_{m'}\ x\ xs')\ ys & \Rightarrow & \mathbf{cons}_{\mathbf{sum}\ m'\ n}\ x\ \left(\mathbf{append}_{m',n}\ xs'\ ys\right) \end{array}$$

Will this typecheck? Let's look at a fragment of the proof tree:

$$\dfrac{\dfrac{\dfrac{???}{\Gamma \vdash \mathbf{sum}\ m'\ n : \text{Length}}\ ??? \qquad \dfrac{\vdots}{\Gamma \vdash x : \text{Nat}} \qquad \dfrac{\vdots}{\Gamma \vdash \mathbf{append}_{m',n}\ xs'\ ys : \mathbf{sum}\ m'\ n}}{\Gamma \vdash \mathbf{cons}_{\mathbf{sum}\ m'\ n}\ x\ \left(\mathbf{append}_{m',n}\ xs'\ ys\right) : \mathbf{VecNat\ succ}\ (\mathbf{sum}\ m'\ n)}\ \textsc{Def-Cons}}{\Gamma \vdash \mathbf{cons}_{\mathbf{sum}\ m'\ n}\ x\ \left(\mathbf{append}_{m',n}\ xs'\ ys\right) : \mathbf{VecNat}\ (\mathbf{sum\ succ}\ m'\ n)}\ ???$$

(Remember that DEF-CONS is actually an abbreviation for of two applications of DEP-APP to the definition of the **cons** function.)

Our problem is that, given that we're in the body of $\mathbf{append}_{m,n}\ xs\ ys$, we need to get a result type of $\mathbf{VecNat}\ \mathbf{sum}\ (\mathbf{succ}\ m')\ n$, not $\mathbf{VecNat}\ \mathbf{succ}\ (\mathbf{sum}\ m'\ n)$. And **sum** isn't even defined! Let's fix that:

$$\underline{\text{let}}\ \frac{m : \text{Length} \qquad n : \text{Length}}{\mathbf{sum}\ m\ n : \text{Length}} = \begin{array}{rcl} \mathbf{sum}\ 0\ n & \Rightarrow & n \\ \mathbf{sum}\ (\mathbf{succ}\ m')\ n & \Rightarrow & \mathbf{succ}\ (\mathbf{sum}\ m'\ n) \end{array}$$

Witness the power of our fully armed and operational dependent type system! We provided a necessary definition, and it included a reduction which the type system can use to prove that the two mismatched types we were worried about were equal. The situation is actually even better than that; in real life, we'd've used Nat instead of Length, and addition would have already been defined.

But we were lucky that the structure of $\mathbf{append}_{m,n}\ xs\ ys$ and $\mathbf{sum}\ m\ n$ were so similar. If we were to write a function that combines two lists in a slightly different fashion...

$$\underline{\text{let}}\ \frac{m : \text{Length} \qquad n : \text{Length} \qquad xs : \mathbf{VecNat}\ m \qquad ys : \mathbf{VecNat}\ n}{\mathbf{aepnpd}_{m,n}\ xs\ ys : \mathbf{VecNat}\ (\mathbf{sum}\ m\ n)} = \begin{array}{rcl} \mathbf{aepnpd}_{0,n}\ \text{nil}\ ys & \Rightarrow & ys \\ \mathbf{aepnpd}_{\mathbf{succ}\ m',n}\ (\mathbf{cons}_{m'}\ x\ xs')\ ys & \Rightarrow & \mathbf{cons}_{\mathbf{sum}\ n\ m'}\ x\ (\mathbf{aepnpd}_{n,m'}\ ys\ xs') \end{array}$$

...it won't typecheck at all. For any particular m and n, it would be possible for the typechecker to normalize to observe the commutitivity of addition, but the typechecker is operating at compile-time, and we don't know what values of m and n we're interested in. In this case, we'll need to prove that addition is commutitive. The Curry-Howard isomorphism guides us to a type, and we'll fill in an implementation that proves it:

$$\underline{\text{let}}\ \frac{m : \text{Length} \qquad n : \text{Length}}{\mathbf{sum\text{-}comm}\ m\ n : \mathbf{Equal}\ (\mathbf{sum}\ m\ n)\ (\mathbf{sum}\ n\ m)} = \begin{array}{rcl} \mathbf{sum}\ 0\ n & \Rightarrow & \mathbf{zero\text{-}sum\text{-}comm}\ n \\ \mathbf{sum}\ (\mathbf{succ}\ m')\ n & \Rightarrow & [\mathbf{sum\text{-}comm}\ m'\ n\rangle \end{array}$$

The weird $[\rangle$ operator takes a proof that $\mathbf{Equal}\ e_1\ e_2$ and turns it into a proof that $\mathbf{Equal}\ c[e_1]\ c[e_2]$ for whichever c gives the right type of result (in this case, $\mathbf{succ}\ []$). Definitions for **Equal** and **zero-sum-comm** have been ommitted, but hopefully it's clear that the process will eventually bottom out.

Now, repairing our **aepnpd** function requires a type ascription operator, which uses a proof of equality to change something's type:

$$\frac{Q : \mathbf{Equal}\ \tau_1\ \tau_2 \qquad e : \tau_1}{\{Q\}\ e : \tau_2}\ \text{T-ASCRIBE}$$

We can now use **sum-comm** to generate proofs of commutivity. And thanks to those annoying ms and ns we've been carrying around, we know exactly what proof to generate.

$$\underline{\text{let}}\ \frac{m : \text{Length} \qquad n : \text{Length} \qquad xs : \mathbf{VecNat}\ m \qquad ys : \mathbf{VecNat}\ n}{\mathbf{aepnpd}_{m,n}\ xs\ ys : \mathbf{VecNat}\ (\mathbf{sum}\ m\ n)} = \begin{array}{rcl} \mathbf{aepnpd}_{0,n}\ \text{nil}\ ys & \Rightarrow & ys \\ \mathbf{aepnpd}_{\mathbf{succ}\ m',n}\ (\mathbf{cons}_{m'}\ x\ xs')\ ys & \Rightarrow & [\{\mathbf{sum\text{-}comm}\ n\ m'\}\rangle \mathbf{cons}_{\mathbf{sum}\ n\ m'}\ x\ \left(\mathbf{aepnpd}_{n,m'}\ ys\ xs'\right) \end{array}$$

4 On choosing types

Consider the following function, missing its return type:

$$\underline{\text{let}}\ \frac{m : \text{Length} \qquad x : \text{Nat} \qquad xs : \textbf{VecNat}\ m}{\textbf{uniq-cons}_m\ x\ xs : ???}$$
$$\textbf{uniq-cons}_m\ x\ xs \Rightarrow \underline{\text{match}}\ \textbf{member?}\ x\ xs$$
$$\text{true} \Rightarrow xs$$
$$\text{false} \Rightarrow \textbf{cons}_m\ x\ xs$$

What should go there? We could define **nonmember-as-nat** to be like **member?** , but returning 0 for membership and 1 for nonmembership. Then its type could be **VecNat sum** m **nonmember-as-nat** $x\ xs$. Does that seem good?

Well, for one thing, it's going to be a pain to thread through the necessary ingredients to prove that that's really the return type of **uniq-cons** But the more serious problem is going to come up when we try to deal with the following:

$$\textbf{cdr}_{???}\ \textbf{uniq-cons}_m\ x\ xs$$

We're going to need to know whether x is in xs before we can figure out if this is legal, or what length to give as an argument to **cdr**. We could, in theory, do this. The end result of this strategy leaves us carting around everything we've ever proved, in much the same manner that the protaganist of a point-and-click adventure game pockets every object they encounter that is not nailed down.

The point of this is that, unlike pretty much every other type system that we've talked about, dependently-typed languages do not have principal types in any meaningful way. And therefore, there isn't an obviously-correct type for **uniq-cons**. **VecNat** is not a meaningful fit for its return value. Perhaps a type for vectors with length at least m would work better, but even then we have choices. Do we want to mention that it returns a vector at least as long as xs is: **AtLeastVecNat** m? Or that it never returns an empty vector: **AtLeastVecNat** 1? Or both: (**AtLeastVecNat** m) $\cap$ (**AtLeastVecNat** 1)? It depends on the situation.

5 Σ types

We've been talking about using dependent types to characterize the behavior of functions. But that's not all we're interested in. It would also be nice to talk about relationships amongst data. Suppose that we're writing crypto software, and we'd like a type to represent a public/private keypair. We could write this:

$$\Sigma pv_0{:}\text{Prime}\ .\ (\Sigma pv_1{:}\text{Prime}\ .\ \textbf{TheNaturalNumber}\ \ (\textbf{prod}\ pv_0\ pv_1))$$

5.1 Wherefore Π and Σ?

To explain the choice of Greek letters here (it's especially confusing to see Σ used to talk about a *product*), we need to cross the Curry-Howard isomorphism.

$$\begin{aligned} \Pi x{:}\tau_0.\tau_1 &\equiv \forall x.\,(x\ \textbf{proves}\ \tau_0) \rightarrow \tau_1 \\ \Sigma x{:}\tau_0.\tau_1 &\equiv \exists x.\,(x\ \textbf{proves}\ \tau_0) \wedge \tau_1 \end{aligned}$$

Remember that the truth of the proposition corresponds to the inhabitedness of the isomorphic type. Dependent types are isomorphic to a pretty weird logic, one where the universe of discourse is the set of proofs of its own theorems.

6 The lambda cube

I haven't shown a complete definition for dependent types yet. There's a reason for that; it's a little tiresome. The big problem is the kinding rules, which I've almost completely omitted; if you squint, they look an awful

lot like the typing rules. Seriously, go to page 51 of ATTAPL and squint, and you'll understand half of the motivation for this section.

$$\begin{array}{rcl} e & ::= & s \mid x \mid \lambda x{:}e.e \mid ee \mid \Pi x{:}e.e \\ s & ::= & * \mid \Box \\ \Gamma & ::= & \varnothing \mid \Gamma, x{:}e \end{array}$$

$$\frac{}{\Gamma \vdash * : \Box}\ \text{T-Star}$$

$$\frac{x{:}e_0 \in \Gamma}{\Gamma \vdash x : e_0}\ \text{T-Var}$$

$$\frac{\Gamma \vdash e_0 : s_i \qquad \Gamma, x{:}e_0 \vdash e_1 : e_2}{\Gamma \vdash \lambda x{:}e_0.e_1 : \Pi x{:}e_0.e_2}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash e_0 : \Pi x{:}e_2.e_3 \qquad \Gamma \vdash e_1 : e_2}{\Gamma \vdash e_0 e_1 : [x \mapsto e_1]e_3}\ \text{T-App}$$

$$\frac{\Gamma \vdash e_0 : s_i \qquad \Gamma, x{:}e_0 \vdash e_1 : s_j}{\Gamma \vdash \Pi x{:}e_0.e_1 : s_j}\ \text{T-Pi}$$

$$\frac{\Gamma \vdash e_0 : e_1 \qquad e_1 \equiv e_2 \qquad \Gamma \vdash e_2 : s}{\Gamma \vdash e_0 : e_2}\ \text{T-Conv}$$

$$(s_i, s_j) \in \{(*,*), (*,\Box)\}$$

These rules are not unlike the ones we've seen before, but now the distinction between types and terms has been completely erased. The way that you recover them is by looking T-Pi. Notice that it's strictly a judgement about well-formedness of things that aren't terms. It controls what kind of abstractions can occur.

And if we want different sets of powers, we need only change the allowable sorts in that last line:

$$\begin{array}{rl} \lambda_{\to} & \{(*,*)\} \\ \lambda P & \{(*,*), (*,\Box)\} \\ F & \{(*,*), (\Box,*)\} \\ F^{\omega} & \{(*,*), (\Box,*), (\Box,\Box)\} \\ CC & \{(*,*), (\Box,*), (*,\Box), (\Box,\Box)\} \end{array}$$

In fact, each of (□,∗), (∗,□), and (□, □) can be toggled independently, forming the axes of a cube: