Lambda-cube, part 3

Lambda-cube crew

June 28, 2023

So far we have seen three versions of typed lambda-calculus: simply-typed, polymorphic, and weak omega. To sum these up,

In	simply-typed lambda-calculus	$\lambda\to$	terms depend on terms
In	system F	$\lambda 2$	terms depend on types
In	weak-omega	$\lambda\underline{\omega}$	types depend on types

Let us now introduce the missing version of lambda-calculus: dependent types λP, where types will depend on terms. Once we have defined this, we will have all three edges departing from the simply-typed vertex on the lambda cube, as displayed in Figure **??**. After that, we can combine our different approaches to complete the whole cube.

1 Dependent types

In λP, types are allowed to depend on arbitrary terms, and are therefore refered to as *dependent types*.

The canonical example of a dependent type is vectors of a set length, where the size of a vector is baked into its type as an integer: the empty vector has type $\mathbf{vec}_0$, a vector of size 1 has type $\mathbf{vec}_1$, etc. The type $\mathbf{vec}_n$ is dependent on n, an integer.

More generally, for any type A and any kind k, we will now authorize building types of kind k that depend on terms of type A, i.e. $A \to k$ is now a kind.

$$\begin{tikzcd}
\lambda 2 & & \\
& \lambda\underline{\omega} & \\
\lambda\to \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P
\end{tikzcd}$$

Figure 1: Half lambda-cube

For instance, $A \to \star\colon \Box$. If $f\colon A \to \star$ and $a\colon A$, then we can apply f to a, and $f(a)\colon \star$, i.e. $f(a)$ is a type depending on the term a. Likewise, we can use abstractions to create arrow-types: $\lambda x\colon A.f(x)\colon A \to \star$. Taking $\mathbb{N}$ for A and **vec** for f, we get that $\mathbf{vec}_n$ is a type, for any integer n.

Now imagine we wish to write a function that, given an integer n, constructs a vector of size n initialize at some value. What type should that function have? If one writes $\mathbf{N} \to \mathbf{vec}_n$, the n in the subscript of the result-type is ill-defined: we wish that n to be exactly the input provided to the function. In order to express the type of this function, we must generalise the notion of arrow-type to now allow the argument to be named and used in the right-hand-side of the arrow. We do this but introducing $\Pi-$types of the form

$$\Pi x : A.B$$

meaning a function that expects type A as an argument, and yields a result of type B. Importantly, the type B will be allowed to reference x, the argument given to the function. If B does not depend on x, then $\Pi x : A.B$ is exactly the type we previously denoted by $A \to B$. Given these $\Pi-$types, the function we mentioned earlier, that takes an integer argument and returns a vector of that given type, now can be typed to type $\Pi n : \mathbb{N}.\mathbf{vec}_n$.

Let us now formalise the way we define λP. The terms in this language are defined by

$$\mathcal{T} \ ::= \ \mathcal{V} \mid \mathcal{C} \mid \mathcal{T}\mathcal{T} \mid \lambda\mathcal{V}\colon \mathcal{T}.\mathcal{T} \mid \Pi\mathcal{V}\colon \mathcal{T}.\mathcal{T}$$

Just like in $\lambda\underline{\omega}$, there is no distinction between types and terms. We have generalised $\mathcal{T} \to \mathcal{T}$ to $\Pi\mathcal{V} : \mathcal{T}.\mathcal{T}$. Just like in $\lambda\underline{\omega}$, the set of constants $\mathcal{C}$ contains the set of sorts $\mathcal{S} = \{\star, \Box\}$. The statements are still of the form $M\colon A$ where $M \in \mathcal{T}$ and $A \in \mathcal{T}$, and bases are still linearily ordered in the form $\Gamma =< [x_1\colon A_1, \ldots, x_n\colon A_n] >$.

We can still prove statements like $\alpha\colon \star, x\colon \alpha \vdash x\colon \alpha$, but now where in $\lambda\underline{\omega}$ we wrote $\alpha\colon \star \vdash \lambda x\colon \alpha.x\colon \alpha \to \alpha$ to type the identity function, we now write

$$\alpha\colon \star \vdash \lambda x\colon \alpha.x\colon \Pi x\colon \alpha.\alpha$$

More precisely, our typing rules are now:

$$\textsc{Ax/sort}\quad \frac{}{\vdash \star\colon \Box} \qquad \textsc{var}\quad \frac{\Gamma \vdash A\colon s \qquad x \notin \Gamma}{\Gamma, x\colon A \vdash x\colon A} \qquad \textsc{Weak}\quad \frac{\Gamma \vdash M\colon B \qquad \Gamma \vdash A\colon s \qquad x \notin \Gamma}{\Gamma, x\colon A \vdash M\colon B}$$

$$\textsc{Lambda}\quad \frac{\Gamma, x\colon A \vdash M\colon B \qquad \Gamma \vdash \Pi x\colon A.B\colon s}{\Gamma \vdash \lambda x\colon A.M\colon \Pi x\colon A.B} \qquad \textsc{App}\quad \frac{\Gamma \vdash M\colon \Pi x\colon A.B \qquad \Gamma \vdash N\colon A}{\Gamma \vdash MN\colon B[N/x]}$$

$$\textsc{ConvBeta}\quad \frac{\Gamma \vdash M\colon A \qquad \Gamma \vdash B\colon s \qquad A =_\beta B}{\Gamma \vdash M\colon B} \qquad \textsc{Type/Kind Pi}\quad \frac{\Gamma \vdash A\colon \star \qquad \Gamma, x\colon A \vdash B\colon s}{\Gamma \vdash \Pi x\colon A.B\colon s}$$

Where s stands for $\star$ or $\Box$.

These rules are very close to the ones in $\lambda\underline{\omega}$, with two main differences:

$$\dfrac{\textsc{Var}\;\dfrac{\textsc{Ax/sort}\;\dfrac{}{\vdash \star\colon \Box}}{A\colon \star \vdash A\colon\ \star} \qquad \dfrac{\textsc{Weak}\;\dfrac{\textsc{Ax/sort}\;\dfrac{}{\vdash \star\colon \Box}}{A\colon \star \vdash \star\colon \Box} \quad \dfrac{\dfrac{}{\vdash \star\colon \Box}\;\textsc{Ax/sort}}{A\colon \star \vdash A\colon\ \star}\;\textsc{Var}}{A\colon \star, x\colon A \vdash \star\colon \Box}\;\textsc{Weak}}{A\colon \star \vdash (A \to \star)\colon \Box}\;\textsc{Type/Kind Pi}$$

Figure 2: Proof of example 1

- The arrow type has now become the more general $\Pi-$type. This means that the application rule now has a term substitution in the result type, to account for types depending on terms.
- The Type/Kind rule now allows creating a $\Pi-$type from types to types, or a $\Pi-$kind from types to kind, where in $\lambda\underline{\omega}$, we could build an arrow-type from type to type or an arrow-kind from kind to kind.

Using these rules, we can prove for example (using $A \to B$ as a shorthand for $\Pi x\colon A.B$ when x isn't free in B):

1. $$A\colon \star \vdash (A \to \star)\colon \Box$$

   We show the proof tree in Figure **??**

2. $$A\colon \star, P\colon A \to \star, a\colon A \vdash Pa\colon \star$$

   To prove this, we use the following proof derivation:

   $$\dfrac{\dfrac{\text{Weak, Var and 1.}}{A\colon \star, P\colon A \to \star, a\colon A \vdash P\colon A \to \star} \qquad \dfrac{\text{Weak and 1.}}{A\colon \star, P\colon A \to \star, a\colon A \vdash A \to \star\colon \Box}}{A\colon \star, P\colon A \to \star, a\colon A \vdash Pa\colon\ \star}\;\textsc{App}$$

3. $$A\colon \star, P\colon A \to \star, a\colon A \vdash Pa \to \star\colon \Box$$

   To prove this, we use the following proof derivation:

   $$\dfrac{\dfrac{2.}{A\colon \star, P\colon A \to \star, a\colon A \vdash Pa\colon\ \star} \qquad \dfrac{\text{Weak, Var, 1. and Ax/sort}}{A\colon \star, P\colon A \to \star, a\colon A, x\colon \star \vdash \star\colon \Box}}{A\colon \star, P\colon A \to \star, a\colon A \vdash Pa \to \star\colon \Box}\;\textsc{Type/Kind Pi}$$

4. $$A\colon \star, P\colon A \to \star \vdash \Pi a\colon A.Pa \to \star\colon \Box$$

   To prove this, we use the following proof derivation:

   $$\dfrac{\dfrac{\text{Weak, Var and 1.}}{A\colon \star, P\colon A \to \star \vdash A\colon\ \star} \qquad \dfrac{3.}{A\colon \star, P\colon A \to \star, a\colon A \vdash Pa \to \star\colon \Box}}{A\colon \star, P\colon A \to \star \vdash \Pi a\colon A.Pa \to \star\colon \Box}\;\textsc{Type/Kind Pi}$$

5.
$$A\colon \star, P\colon A \to \star \vdash (\lambda a\colon A.\lambda x\colon Pa.x)\colon (\Pi a\colon A.(Pa \to Pa))$$

To prove this, we apply rule Lambda, leaving us with 2 proof obligations. The second one is similar to 4. and can be proved through similar methods. To prove the first one, we use the following proof derivation:

$$\dfrac{\textsc{Var}\ \dfrac{\dfrac{2.}{A\colon \star, P\colon A \to \star, a\colon A \vdash Pa\colon \star}}{A\colon \star, P\colon A \to \star, a\colon A, x\colon Pa \vdash x\colon Pa} \qquad \dfrac{\text{Similar to 3.}}{A\colon \star, P\colon A \to \star, a\colon A \vdash Pa \to Pa\colon \Box}}{A\colon \star, P\colon A \to \star, a\colon A \vdash \lambda x\colon Pa.x\colon Pa \to Pa}\ \textsc{Lambda}$$

The Curry-Howard correspondance holds for λP, and because types can depend on terms, we can now reason about logical predicates that depend on simpler types. A predicate P on set A is represented as $P\colon A \to \star$. For $a \in A$, Pa is valid if, and only if, it is inhabited as a type. $\forall x \in A.Px$ is translated as $\Pi x\colon A.Px$, and $A \to B$ is translated as $\Pi x\colon A.B$.

For example, the formula

$$(\forall x \in A, \forall y \in A, Pxy) \implies (\forall x \in A, Pxx)$$

is valid because its translation in λP is inhabited:

$$A\colon \star, P\colon A \to A \to \star \vdash [\lambda z\colon (\Pi x\colon A.\Pi y\colon A.Pxy).\lambda x\colon A.zxx]\colon$$
$$((\Pi x\colon A.\Pi y\colon A.Pxy) \to (\Pi x\colon A.Pxx))$$

2 The lambda-cube

We are now ready to close the lamdba-cube. We present a visual representation of the cube in Figure **??**: the simply-typed lambda-calculus is our base vertex, and walking along the edges adds extra features. Going vertically up allows for polymorphism (terms depending on types), going back in depth allows for type functions (types depending on types), and going right horizontally allows for dependent types (types depending on terms).

The beauty of the lambda-cube is that we can define one language that encompasses all systems. The language has the same syntax than dependent types presented in the previous sections, and all rules are the same, except for the last rule Type/Kind Pi.

Instead of that rule, we introduce rule Pi:

$$\textsc{Pi}\quad \dfrac{\Gamma \vdash A\colon s_1 \qquad \Gamma, x\colon A \vdash B\colon s_2 \qquad (s_1, s_2) \in \mathcal{R}}{\Gamma \vdash \Pi x\colon A.B\colon s_2}$$

To switch from different systems of lambda-calculus, we simply change the definition of relation $\mathcal{R}$. For simply-typed lambda-calculus, only $(\star, \star)$ is in $\mathcal{R}$: only simple arrow-types are allowed. For polymorphism in System F, we also allow $(\Box, \star)$, i.e. types can depend on kinds. In $\lambda\underline{\omega}$, $\mathcal{R}$ is the relation $\{(\star, \star); (\Box, \Box)\}$, i.e. we can only create simple arrow types or arrows from kinds

$$\begin{tikzcd}
 & \lambda\omega \arrow[rr] & & \lambda P\omega \\
\lambda 2 \arrow[ur] \arrow[rr] & & \lambda P2 \arrow[ur] & \\
 & \lambda\underline{\omega} \arrow[uu] \arrow[rr] & & \lambda P\underline{\omega} \arrow[uu] \\
\lambda{\to} \arrow[uu] \arrow[ur] \arrow[rr] & & \lambda P \arrow[uu] \arrow[ur] &
\end{tikzcd}$$

Figure 3: Lambda cube

to kinds. And in accordance to the rules presented in the previous section, the relation for dependent types is $\{(\star,\star);(\star,\Box)\}$.

To combine these feature, one can simply augment $\mathcal{R}$. The following table shows how to define $\mathcal{R}$ for each system:

System	$\mathcal{R}$			
$\lambda\to$	$(\star,\star)$			
$\lambda 2$	$(\star,\star)$	$(\Box,\star)$		
λP	$(\star,\star)$		$(\star,\Box)$	
$\lambda\underline{\omega}$	$(\star,\star)$			$(\Box,\Box)$
$\lambda P2$	$(\star,\star)$	$(\Box,\star)$	$(\star,\Box)$	
$\lambda\omega$	$(\star,\star)$	$(\Box,\star)$		$(\Box,\Box)$
$\lambda P\underline{\omega}$	$(\star,\star)$		$(\star,\Box)$	$(\Box,\Box)$
$\lambda P\omega$	$(\star,\star)$	$(\Box,\star)$	$(\star,\Box)$	$(\Box,\Box)$

These are the related systems:

System		
$\lambda\to$	simple type theory	Church (1940)
$\lambda 2$	system $\mathcal{F}$	Girard (1972)
		Reynolds (1974)
λP	AUT-QE (AUTOMAT)	de Bruijn (1970)
	logical frameworks (LF)	Harper et al. (1987)
$\lambda\underline{\omega}$	POLYREC	Renardel de Lavalette (1991)
$\lambda P2$		Longo and Moggi (1988)
$\lambda P\omega$	$\mathcal{F}\omega$	Girard (1972)
$\lambda P\underline{\omega}$	calculus of constructions	Coquand and Huet (1988)
$\lambda P\underline{\omega}$		

All eight systems in the cube are strongly normalising.