Lambda-cube and dependent types

Jacques Garrigue, 2018/06/19

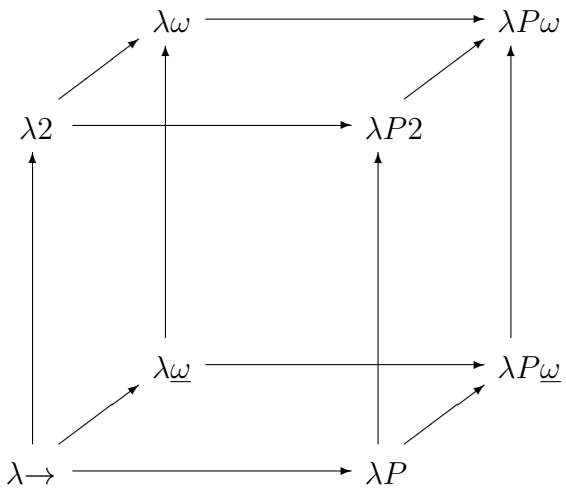
With the simply typed lambda-calculus and System F we have seen two ways in which abstraction can be introduced in the lambda-calculus:

- terms abstracted on terms ($\lambda x : T.M$), or proposition logic
- terms abstracted on types ($\Lambda\alpha.M$), or second-order logic

Just considering terms and types, one can think of two more forms of abstraction

- types abstracted on terms ($\lambda x : T.T$), or predicate logic
- types abstracted on types ($\Lambda\alpha.T$), or higher-order types

Combinations of those can be formalized as the 8 corners of a cube, the lambda-cube.



1 First-order logic and dependent types (λP)

The propositions of first-order logic are similar to polymorphic types.

$$
\begin{array}{rcll}
t & ::= & x \mid a \mid f(t,\ldots) & \text{term} \\
A & ::= & \bot \mid A \to A \mid A \wedge A \mid A \vee A & \\
 & \mid & p(t,\ldots) & \text{predicate} \\
 & \mid & \forall x.A & \text{universal quantification} \\
 & \mid & \exists x.A & \text{existential quantification}
\end{array}
$$

However, here t and x are terms, not types. This means that terms appear in types, rather than types in terms.

$$
\begin{array}{rcll}
T & ::= & b \mid \bot \mid p_T\ M \mid \Pi x{:}T.T \mid \Sigma x{:}T.T & \text{type} \\
M & ::= & x \mid c_T \mid \lambda x{:}T.M \mid (MM) \mid (M, x.M) & \text{term}
\end{array}
$$

If x does not appear in T_2, then $\Pi x{:}T_1.T_2$ (dependent function type) can be written $T_1 \to T_2$. Similarly, $(M_1, x.M_2)$ (dependent pair) and $\Sigma x{:}T_1.T_2$ (dependent pair type) can be written respectively (M_1, M_2) and $T_1 \times T_2$.

We add extra rules, which also check the well-formedness of types.

Type $\dfrac{\Gamma \vdash M : T}{\Gamma \vdash p_T\ M\ \text{ok}} \qquad \dfrac{\Gamma, x : T \vdash T' \text{ ok}}{\Gamma \vdash \Pi x{:}T.T' \text{ ok}} \qquad \dfrac{\Gamma, x : T \vdash T' \text{ ok}}{\Gamma \vdash \Sigma x{:}T.T' \text{ ok}}$

Abs $\dfrac{\Gamma, x : T \vdash M : T'}{\Gamma \vdash \lambda x{:}T.M : \Pi x{:}T.T'} \qquad \dfrac{\Gamma \vdash M_1 : T \quad \Gamma, x : T_1 \vdash M_2 : T_2}{\Gamma \vdash (M_1, x.M_2) : \Sigma x{:}T_1.T_2}$

App $\dfrac{\Gamma \vdash M : \Pi x{:}T.T' \quad \Gamma \vdash N : T}{\Gamma \vdash (M\ N) : [N/x]T'} \qquad \dfrac{\Gamma \vdash M : \Sigma x{:}T_1.T_2}{\Gamma \vdash \mathsf{fst}\ M : T_1}$

Neg $\dfrac{\Gamma \vdash M : \bot \quad \Gamma \vdash T \text{ ok}}{\Gamma \vdash M : T} \qquad \dfrac{\Gamma \vdash M : \Sigma x{:}T_1.T_2}{\Gamma \vdash \mathsf{snd}\ M : [\mathsf{fst}\ M/x]T_2}$

Conv $\dfrac{\Gamma \vdash M : [N/x]T \quad \Gamma \vdash N' : T' \quad N =_{\beta\delta} N'}{\Gamma \vdash M : [N'/x]T}$

We can encode first-order logic in this calculus, but since we do not distinguish between terms of the logic and proofs, this is just a morphism, not an isomorphism.

As an example, here is the encoding of "Humans are mortal, Socrates is a human, so Socrates is mortal".

$$(\Pi x{:}Name.Human\ x \to Mortal\ x) \to Human\ Socrates \to Mortal\ Socrates$$

And here is the proof.

$$\lambda mortal{:}(\Pi x{:}Name.Human\ x \to Mortal\ x).\ \lambda human{:}(Human\ Socrates).mortal\ Socrates\ human$$

We can also write mathematical formulas such as $\forall x.x + x = 2 \times x$.

$$\Pi x{:}\mathsf{Nat}.eqnat(add\ x\ x, mult\ 2\ x),$$

and compute with δ-rules

$$\begin{array}{l} add\ 0\ n \to n \\ add\ (sm)\ n \to s\ (add\ m\ n) \\ mult\ 0\ n \to 0 \\ mult\ (sm)\ n \to add\ n\ (mult\ m\ n) \end{array}$$

Then one can prove properties using the following constants (axioms).

$$\begin{array}{l} add\_sym : \Pi m{:}\mathsf{Nat}.\Pi n{:}\mathsf{Nat}.eqnat(add\ m\ n, add\ n\ m) \\ eq\_sub : \Pi f{:}(\mathsf{Nat} \to \mathsf{Nat}).\Pi m{:}\mathsf{Nat}.\Pi n{:}\mathsf{Nat}.eqnat(m,n) \to eqnat(f\ m, f\ n) \end{array}$$

Here is a proof of the above formula.

$$\lambda x{:}\mathsf{Nat}.eq\_sub\ (\lambda y{:}\mathsf{Nat}.add\ x\ y)\ (add\_sym\ 0\ x)$$

2 Generalized Type Systems

A nice way to define the systems of the lambda-cube in a common framework is to use generalized type systems.

In generalized type systems (or GTS), we do not distinguish between terms and types.

$$\begin{array}{rcll} T & ::= & x \mid c \mid T\ T \mid \lambda x : T.T & \text{terms} \\ & \mid & \Pi x : T.T & \text{product type} \end{array}$$

Terms will be written T, A, B, C. A constant c belongs to the set $\mathbf{C}$ of constants. Again, product types are a generalization of function types, and if $x \notin \mathsf{fv}(T_2)$, we write $T_1 \to T_2$ for $\Pi x : T_1.T_2$.

A specific GTS is determined by a triple $(\mathbf{S}, \mathbf{A}, \mathbf{R})$

1. $\mathbf{S} \subset \mathbf{C}$ is the set of sorts.
2. $\mathbf{A}$ is a set of axioms of the form $c : s \in \mathbf{C} \times \mathbf{S}$.
3. $\mathbf{R}$ is a set of rules of the form $(s_1, s_2, s_3) \in \mathbf{S} \times \mathbf{S} \times \mathbf{S}$.

We use $\Gamma \vdash T : A : s$ as a shorthand for $\Gamma \vdash T : A$ and $\Gamma \vdash A : s$. $A =_\beta B$ means that there exists a C such that $A \xrightarrow{*} C$ and $B \xrightarrow{*} C$.

Axiom $\quad \emptyset \vdash c : s \quad (c : s \in \mathbf{A})$

Start $\quad \dfrac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$

Weaken $\quad \dfrac{\Gamma \vdash B : C \quad \Gamma \vdash A : s}{\Gamma, x : A \vdash B : C}$

Prod $\quad \dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash \Pi x{:}A.B : s_3}\ (s_1, s_2, s_3) \in \mathbf{R}$

Abs $\quad \dfrac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash T : B : s_2}{\Gamma \vdash (\lambda x{:}A.T) : (\Pi x{:}A.B)}\ (s_1, s_2, s_3) \in \mathbf{R}$

App $\quad \dfrac{\Gamma \vdash F : (\Pi x{:}A.B) : s \quad \Gamma \vdash T : A}{\Gamma \vdash F\ T : [T/x]B}$

Conv $\quad \dfrac{\Gamma \vdash T : A \quad \Gamma \vdash B : s \quad A =_\beta B}{\Gamma \vdash T : B}$

The systems of the lambda-cube are a restricted form of GTS such that

1. $\mathbf{S} = \{*, \Box\}$, where $*$ is the sort of terms and $\Box$ the sort of types,
2. $\mathbf{A} = \{* : \Box\}$, meaning that $*$ has type $\Box$,
3. all rules in $\mathbf{R}$ are of the form (s_1, s_2, s_2) (so the we abbreviate them in (s_1, s_2)).

$\lambda\!\to$	$(*,*)$
$\lambda 2$	$(*,*), (\Box,*)$
$\lambda\underline{\omega}$	$(*,*), (\Box,\Box)$
$\lambda\omega$	$(*,*), (\Box,*), (\Box,\Box)$
λP	$(*,*), (*,\Box)$
$\lambda P2$	$(*,*), (\Box,*), (*,\Box)$
$\lambda P\underline{\omega}$	$(*,*), (\Box,\Box), (*,\Box)$
$\lambda P\omega$	$(*,*), (\Box,*), (\Box,\Box), (*,\Box)$

System F ($\lambda 2$) First note that since there is no rule $(*, \Box)$, there is no way a term (of type $A : *$) could appear inside a type (of type $* : \Box$), so that if $A : *$ and $B : *$, then $\Pi x{:}A.B$ is just $A \to B$, as x cannot occur in B. Then **Prod** has the 2 instances:

$$\frac{\Gamma \vdash A : * \quad \Gamma \vdash B : *}{\Gamma \vdash A \to B : *} \qquad \frac{\Gamma \vdash * : \Box \quad \Gamma, A : * \vdash B : *}{\Gamma \vdash \Pi A{:}* .B : *}$$

and **Abs** and **App** have the 4 instances of System F.

Dependent types (λP) Since we have the rule $(*, \Box)$, terms can appear in types.

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi x{:}A.B : *} \qquad \frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : \Box}{\Gamma \vdash \Pi x{:}A.B : \Box}$$

One can use this to encode predicate logic. $\forall x \in T, P(x) \supset Q(x)$ can be derived in the environment $\Gamma = \{T : *, P : T \to *, Q : T \to *\}$, as

$$\dfrac{\dfrac{\emptyset \vdash * : \Box}{\Gamma \vdash T : *} \quad \dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x : T \vdash T : * \quad \Gamma, x : T \vdash * : \Box}{\Gamma, x : T \vdash T \to * : \Box}}{\Gamma, x : T \vdash P : T \to * : \Box} \quad \dfrac{\dfrac{\emptyset \vdash * : \Box}{\Gamma \vdash T : *}}{\Gamma, x : T \vdash x : T}}{\Gamma, x : T \vdash P\ x : *} \quad \dfrac{\cdots}{\Gamma, x : T \vdash Q\ x : *}}{\Gamma, x : T \vdash P\ x \to Q\ x : *}}{\Gamma \vdash (\Pi x{:}T.P\ x \to Q\ x) : *}$$

Predicate logic λPRED While λP is sufficient to encode predicate logic, it may be confusing as it mixes terms of the logic (used as arguments to predicates) with proofs of propositions. By using the full power of GTS, one can separate them, and create a system isomorphic to multi-sorted first-order predicate logic.

$$\begin{array}{ll} \mathbf{S} & *^s, *^p, *^f, \Box^s, \Box^p \\ \mathbf{A} & *^s : \Box^s, *^p : \Box^p \\ \mathbf{R} & (*^p, *^p), (*^s, *^p), (*^s, \Box^p), (*^s, *^f), (*^s, *^s, *^f) \end{array}$$

Here $*^s$ is the sort of sets (the terms used by the logic), $*^p$ the sort of propositions, and $*^f$ the sort of first-order functions between sets of $*^s$.

Inconsistence of $\lambda *$ Out of the lambda-cube, one could define a system with a "type of all types", i.e. $\mathbf{A} = \{* : *\}$ and $\mathbf{R} = \{(*, *)\}$. However, Girard proved it to be inconsistent. This was the starting point for System F.

Calculus of Constructions ($\lambda P\omega$) The full system (allowing all 4 kinds of abstractions) is called Calculus of Constructions. It has no logical counterpart, but it can be seen as a powerful logic by itself. Similarly to System F, it can encode various connectives of logic without adding extra constants.

It is the basis of several *proof assistants* (computer systems that allow one to write proofs and check them). In particular Coq has been used to prove important theorems of mathematics, such as the 4-colour theorem or the Odd-order theorem. Note that Coq actually uses a further extension of $\lambda P\omega$, with a countable hierarchy of sorts above $*$ and $\Box$, to get closer to $\lambda *$ without introducing inconsistency, and allows to define inductive types, to allow using induction without introducing new axioms.