Notes: Barendregt's cube and programming with dependent types

Eric Lu

I. ANNOTATED BIBLIOGRAPHY

A. Lambda Calculi with Types

```
@incollection{barendregt:1992
    author = {Barendregt, Henk},
    title = {Lambda Calculi with Types},
    booktitle = {Handbook of Logic in Computer Science},
    publisher = {Oxford University Press},
    year = {1992},
    editor = {Abramsky, S. and Gabbay, D.M. and Maibaum, T.S.E.},
    volume = {2},
    pages = {117--309},
    address = {New York, New York}
}
```

Summary: As setup for introducing the lambda cube, Barendregt introduces the type-free lambda calculus and reviews some of its properties. Subsequently he presents accounts of typed lambda calculi from Curry and Church, including some Curry-style systems not belonging to the lambda cube (λU and $\lambda\mu$). He then describes the lambda cube construction that was first noted by Barendregt in 1991. The lambda cube describes an inclusion relation amongst eight typed lambda calculi. Moreover it explains a fine structure for the calculus of constructions arising from the presence or absence of three axiomatic additions to the simply-typed lambda calculus. The axes of the lambda cube are intuitively justified by the notion of dependency. After introducing the lambda cube, Barendregt gives properties of the languages belonging to the lambda cube, as well as some properties of pure type systems, which generalize the means of description of the axioms of the systems in the lambda cube.

Evaluation: This work originates as a reference from the logical/mathematical community, but its content was important in providing a firm foundation for types in programming languages. The lambda cube explains prior work on a number of type systems by observing a taxonomy and relating them to logical systems via the Curry-Howard isomorphism. This enabled further work in type systems, leading to the extensive developments of the 1990s.

B. Cayenne—a language with dependent types

```
@article{augustsson:cayenne
    author = {Augustsson, Lennart},
    title = {Cayenne\&Mdash;a Language with Dependent Types},
    journal = {SIGPLAN Notices},
    volume = {34},
    pages = {239--250},
    year = {1998},
    month = sep,
    address = {New York, New York}
}
```

Summary: Augustsson presents a dependently typed programming language with syntax similar to Haskell that is structured to enable practical programming with dependent types. Cayenne was deliberately designed to reduce the number of syntactic constructs used to express type, value, and module expressions in order to ease usage. The paper opens with a number of code examples demonstrating what is enabled by dependent typing and what this looks like in Cayenne. Augustsson then describes the core syntax of the language. Noting that this can be difficult to program with, Augustsson separately describes a number of pieces of syntactic sugar that make programming in Cayenne more practical. At this point Augustsson describes and analyzes the type system of Cayenne. One note is that type checking is undecidable; in some cases, it must be checked whether two types are equivalent, but this is hard

because Cayenne is not strongly normalizing. The type checker addresses this by attempting to show equivalence but giving up after some number of reduction steps—evidently these "don't know" cases are fairly uncommon. Finally, Augustsson discusses the implementation of the language. The hierarchy of sorts permits the compiler to know when types may be erased.

Evaluation: Cayenne was part of a body of work on dependent types. Earlier work incorporating (e.g. AUTOMATH) had demonstrated the use of dependent types for encoding logic in the form of proof systems. Cayenne showed how dependent types might be useful for practical programming (i.e. a system concerned with the production of programs for execution rather than the production of constructive proofs) in a user-friendly manner. This was an important step in relating a fairly abstract notion from type theory to programming languages.

II. SIMPLY TYPED LAMBDA CALCULUS

We'll give a re-introduction of simply typed lambda calculus. This is what Barendregt calls the Church version of $\lambda \to$. (Historical comment: Church introduced this in 1940; Curry did an implicit typing version in 1934 for combinators, and 1958 Curry and Feys, 1972 Curry et. al. modified this to λ.) We saw something very similar to $\lambda \to$ in Christos's presentation around a month ago (in fact on the 26th of September). This may be a bit of a review.

Syntax of $\lambda \to$:

$$\mathbb{T} = \quad \mathbb{T} \quad | \quad \mathbb{T} \to \mathbb{T} \tag{1}$$

$$\Lambda_{\mathbb{T}} = \quad V \quad | \quad \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \quad | \quad \lambda V : \mathcal{T}.\Lambda_{\mathbb{T}} \tag{2}$$

V is a basis of "term variables"; $\mathbb{V}$ is the corresponding basis of "type variables". The resulting set $\Lambda_{\mathbb{T}}$ is a set of "pseudoterms" (the reason they're not just the terms being that this syntax doesn't describe correctly typed ones, only all the "syntactically well-formed ones").

The "contraction rules" (which I guess are reduction rules in modern terms) consist of β reduction only:

$$(\lambda x : A.M)N \to_\beta M[x := N] \tag{3}$$

To address the question: "what of capture avoidance?" Barendregt actually discusses on page 8: "For reasons of hygiene it will always be assumed that the bound variables that occur in a certain expression are different from the free ones." Thereafter we don't really worry about this issue.

Typing judgments. We have statements $M : \sigma$, where $M \in \Lambda$ is the subject and $\sigma \in \mathbb{T}$ is the predicate. Declarations are statements with subject a term variable. Basis is a set of declarations about different variables. And finally a statement $M : \sigma$ is derivable from the basis Γ, as in $\Gamma \vdash M : \sigma$ ($\vdash$ is pronounced "yields"), if the following rules produce it:

$$\text{(start)} \quad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \tag{4}$$

$$(\to \text{-elimination}) \quad \frac{\Gamma \vdash M : (A \to B) \quad \Gamma \vdash N : A}{\Gamma \vdash (MN) : B} \tag{5}$$

$$(\to \text{-introduction}) \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash (\lambda x : A.M) : (A \to B)} \tag{6}$$

The "legal" pseudoterms are those $M \in \Lambda_{\mathbb{T}}$ such that $\exists \Gamma, A$ such that $\Gamma \vdash M : A$.

Anyone need examples? I guess we can go quickly do something with the naturals if necessary. Barendregt provides these type assignment examples:

$$\vdash \quad (\lambda a : A.a) : (A \to A) \tag{7}$$

$$b : B \quad \vdash \quad (\lambda a : A.b) : (A \to B) \tag{8}$$

$$b : A \quad \vdash \quad ((\lambda a : A.a)b) : A \tag{9}$$

$$c : A, b : B \quad \vdash \quad (\lambda a : A.b)c : B \tag{10}$$

$$\vdash \quad (\lambda a : A.\lambda b : B.a) : (A \to B \to A) \tag{11}$$

III. POLYMORPHIC TYPED LAMBDA CALCULUS

Christos also presented polymorphically typed λ on the third of October. This was Girard's System F of 1972, also studied by Reynolds 1974. Barendregt calls it $\lambda 2$.

The syntax is just a little different; we add support for type-level functions...

$$\mathbb{T} = \quad \mathbb{T} \quad | \quad \mathbb{T} \to \mathbb{T} \quad | \quad \forall \mathbb{V}\mathbb{T} \tag{12}$$

$$\Lambda_{\mathbb{T}} = \quad V \quad | \quad \Lambda_{\mathbb{T}}\Lambda_{\mathbb{T}} \quad | \quad \lambda V : \mathcal{T}.\Lambda_{\mathbb{T}} \quad | \quad \Lambda_{\mathbb{T}}\mathbb{T} \quad | \quad \Lambda \mathbb{V} \Lambda_{\mathbb{T}} \tag{13}$$

This latter rule lets us bind a type variable.

The contraction rules are extended to operate these...

$$(\lambda x : A.M)N \to_\beta M[x := N] \tag{14}$$
$$(\Lambda\alpha.M)A \to_\beta M[\alpha := A] \tag{15}$$

Finally, we add two more rules:

$$(\forall\text{-elimination}) \quad \frac{\Gamma \vdash M : (\forall\alpha.A)}{\Gamma \vdash MB : A[\alpha := B]}, B \in \mathbb{T} \tag{16}$$

$$(\forall\text{ -introduction}) \quad \frac{\Gamma \vdash M : A}{\Gamma \vdash (\Lambda\alpha.M) : (\forall\alpha.A)}, \alpha \notin FV(\Gamma) \tag{17}$$

FV appears to be "free variables"; this avoids like type name capture or something; particular reason is that the basis for A shouldn't have repeated variables. This is averted in the $\to$-introduction rule just because it's specified x shouldn't be in Γ already either.

Some type assignment examples!

$$\vdash \quad (\lambda a : \alpha.a) : (\alpha \to \alpha) \tag{18}$$
$$\vdash \quad (\Lambda\alpha\lambda a : \alpha.a) : (\forall\alpha.\alpha \to \alpha) \tag{19}$$
$$\vdash \quad (\Lambda\alpha\lambda a : \alpha.a)A : (A \to A) \tag{20}$$
$$b : A \quad \vdash \quad (\Lambda\alpha\lambda a : \alpha.a)Ab : A \tag{21}$$
$$\vdash \quad (\Lambda\beta\lambda a : (\forall\alpha.\alpha).a((\forall\alpha.\alpha) \to \beta)a) : (\forall\beta.(\forall\alpha.\alpha) \to \beta) \tag{22}$$
$$\vdash \quad (\Lambda\beta\lambda a : (\forall\alpha : \alpha).a\beta) : (\forall\beta.(\forall\alpha.\alpha) \to \beta) \tag{23}$$

The last two look weird (and related; the second is a simpler version of the first). In particular what could possibly have type $\forall\alpha.\alpha$? (This is some weird expression that once you feed it a type it just is that type. So in particular in (22) we can pass in $(\forall\alpha.\alpha) \to \beta$—say, to get $p = a((\forall\alpha.\alpha) \to \beta)$ (p for peculiar) and now $pa : \beta$.) Does it not seem like β came out of nowhere? The thing a has no idea how to construct a β or anything. So to dip into the Curry-Howard correspondence for a moment: when interpreted as a proposition, $\forall\alpha.\alpha$ is inconsistent. As it happens $\forall\alpha.\alpha$ is equivalent logically to the statement that anything is true, and no consistent system should contain something of this type. The reason is: given any type A, $(\forall\alpha.\alpha)A : A$ produces a term with that type—meaning for any proposition, $\forall\alpha.\alpha$ constructs a proof of that proposition.

IV. DEPENDENCIES

Now, we can get more general than that. Let's follow the generalization out to get an intuition for what Barendregt was doing constructing all these systems. The observation Barendregt makes is this. In $\lambda \to$ we see terms that "depend on" terms:

$$F : A \to B \quad M : A \implies FM : B \tag{24}$$

where FM "depends on" the term M. Then in $\lambda 2$ we have:

$$G : \forall\alpha.\alpha \to \alpha \quad A \in \mathbb{T} \implies GA : A \to A \tag{25}$$

So given $G = \Lambda\alpha\lambda a : \alpha.a$ for instance, we get a GA that "depends on" the type A.

Note that a dependency gives rise to function abstraction:

$$\lambda m : A.Fm : A \to B \tag{26}$$
$$\Lambda\alpha.G\alpha : \forall\alpha.\alpha \to \alpha \tag{27}$$

This is perhaps the most formal way of encoding the notion of "dependency".

So... given that we have:

1. terms depending on terms
2. terms depending on types

why don't we try constructing systems that get us...?

1. types depending on types

2. types depending on terms

So Barendregt will define $\lambda\underline{\omega}$ which has types that depend on types in the form FA, and λP with types that depend on terms in the form FM.

V. TYPE OPERATORS: TYPES DEPENDING ON TYPES

Here's one natural example for a type that depends on another type; we would like to somehow define $f = \lambda\alpha \in \mathbb{T}.\alpha \to \alpha$. We will also produce the setup for discussing kinds in order to introduce $\lambda\underline{\omega}$.

We're about to begin to mix terms and types up, which suggests that we have to sort of merge what we've been so far referring to separately as $\mathbb{T}$ and $\Lambda_{\mathbb{T}}$. Let's consider a unified set of "pseudo-expressions" $\mathcal{T}$ consisting of:

$$\mathcal{T} = \quad V \quad | \quad C \quad | \quad \mathcal{T}\mathcal{T} \quad | \quad \lambda V : \mathcal{T}.\mathcal{T} \quad | \quad \mathcal{T} \to \mathcal{T} \tag{28}$$

We introduce a constant $* \in C$ so that we say $\sigma : *$ when $\sigma \in \mathbb{T}$. We're now able to say:

$$\alpha, \beta \in \mathbb{T} \implies (\alpha \to \beta) \in \mathbb{T} \tag{29}$$

formally WITHIN the type system as

$$\alpha : *, \beta : * \vdash (\alpha \to \beta) : * \tag{30}$$

Now f can be written $\lambda\alpha : *.\alpha \to \alpha$. But what is f? Something else; it is not a term or a type (in particular, we don't get $f : *$). This is what kinds are for:

$$\mathbb{K} = \quad * \quad | \quad \mathbb{K} \to \mathbb{K} \tag{31}$$

We will also have a constant $\Box$ such that $k : \Box$ means $k \in \mathbb{K}$. We call f a constructor of kind k when $\vdash k : \Box$ and $\vdash f : k$. Now we may write $\vdash (\lambda\alpha : *.\alpha \to \alpha) : (* \to *)$. What we used to call elements of $\mathbb{T}$ are all constructors of kind $*$. $*$ and $\Box$ are called sorts and "are the main reason to introduce constants".

We must modify our notion of basis to be linearly ordered. Guessing that we have a rule that looks like $\to$-introduction, without ordering, we could go:

$$x : \alpha, \alpha : * \vdash x : \alpha \tag{32}$$

$$x : \alpha \vdash (\lambda\alpha : *.x) : (* \to \alpha) \tag{33}$$

but then α is bound inside λ and also is free, which makes no sense. Instead we need to enforce the ordering so that:

$$\alpha : *, x : \alpha \vdash x : \alpha \tag{34}$$

$$\alpha : * \vdash (\lambda x : \alpha.x) : (\alpha \to \alpha) \tag{35}$$

Therefore we define "contexts" to be finite, linearly ordered sets of statements. Statements are still $M : A$, but now $M, A \in \mathcal{T}$ both. Finally we write $<>$ for the empty context.

We are now in a position to give the typing rules for $\lambda\underline{\omega}$. Take $s \in \{*, \Box\}$. (In particular, type/kind formation is

only two rules, rather than like eight.)

$$\text{(axiom)} \quad <> \vdash * : \square \tag{36}$$

$$\text{(start rule)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}, x \notin \Gamma \tag{37}$$

$$\text{(weakening rule)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}, x \notin \Gamma \tag{38}$$

$$\text{(type/kind formation)} \quad \frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s}{\Gamma \vdash (A \to B) : s} \tag{39}$$

$$\text{(application rule)} \quad \frac{\Gamma \vdash F : (A \to B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B} \tag{40}$$

$$\text{(abstraction rule)} \quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (A \to B) : s}{\Gamma \vdash (\lambda x : A.b) : (A \to B)} \tag{41}$$

Let us compare this with the previous sets of rules we've seen. The start rule is like the start rule in the previous two systems we've seen. The weakening rule for them was to be derived; here, it is evidently helpful to include it as an axiom. Barendregt offers proofs of consistency for these type systems: in particular, typed terms are "strongly normalizing", i.e. all reduction sequences beginning with a typed term will terminate. The type/kind formation rule governs both the formation of types $A \to B : *$ and kinds, as discussed. The application and abstraction rules are generalizations of the elimination and introduction rules from earlier, respectively. Finally the conversion rule replaces the conversion rules from before, in particular encoding β reduction.

Examples! These are getting increasingly open-ended so perhaps I read them off wholly.

$$\alpha : *, \beta : * \vdash (\alpha \to \beta) : * \quad \vdash \quad (\alpha \to \beta) : * \tag{42}$$

$$\alpha : *, \beta : *, x : (\alpha \to \beta) \quad \vdash \quad x : (\alpha \to \beta) \tag{43}$$

$$\alpha : *, \beta : * \quad \vdash \quad (\lambda x : (\alpha \to \beta).x) : ((\alpha \to \beta) \to (\alpha \to \beta)) \tag{44}$$

In the first step we apply weakening with a variable x. In the second step we apply abstraction.

Those were a little straightforward I guess. Take $D = \lambda\beta : *.\beta \to \beta$. Then:

$$\vdash \quad D : (* \to *) \tag{45}$$

$$\alpha : * \quad \vdash \quad (\lambda x : D\alpha.x) : D(D\alpha) \tag{46}$$

Look at that last one: what we're saying is that if $x : (\alpha \to \alpha)$ then that lambda term has type $((\alpha \to \alpha) \to (\alpha \to \alpha))$.

Some clarifications from questions and otherwise...

- Some discussion on D. Under Curry-Howard it's a little strange. It takes in a proposition and gives back the same proposition, but it doesn't constitute any sort of statement of $A \implies A$. It's a type operator; perhaps we could call it the identity operator. $D : * \to *$, which also means there are no terms $? : D$—D is not a $*$.
- More explicitly, the type/kind formation rule unfolds into these two rules:

$$\frac{\Gamma \vdash A : * \quad \Gamma \vdash B : *}{\Gamma \vdash (A \to B) : *} \tag{47}$$

$$\frac{\Gamma \vdash A : \square \quad \Gamma \vdash B : \square}{\Gamma \vdash (A \to B) : \square} \tag{48}$$

  The first rule gives us functions, which is the only way of constructing types from other types. The second rule is responsible for what we observe of sort construction: we get our sort hierarchy, $*$, $* \to *$, $* \to * \to *$ ... et cetera.

- How does this differ from $\lambda 2$? Consider our example $D = \lambda\beta : *.\beta \to \beta$. This is not exactly the same as $\forall\alpha.\alpha$ from $\lambda 2$. In particular, when D is given some type γ, $D\gamma$ returns another TYPE γ. On the other hand, $(\forall\alpha.\alpha)\gamma$ is an actual TERM of type γ. In general, $\lambda\underline{\omega}$ allows for "computation in type-space" that eventually generates a type, but can't generate terms that are different depending on some type input like $\lambda 2$ can. The logical correspondence is something like: though we can now construct propositions more generally, the result needs to be proven on its own. We don't get to present proofs for bunches of propositions, like $\forall$ corresponds to. This logical system is weird and doesn't seem to have a name.

As a historical note, Barendregt gives that a similar system under the name POLYREC had been discussed by Renardel de Lavalette, 1991.

VI. DEPENDENT TYPES: TYPES DEPENDING ON TERMS

Finally we will introduce the system λP, which has types that can depend on terms! The motivation Barendregt uses is functions $A^n \to B$, where n is a natural number. In particular we don't yet have a way of describing exactly those A^n. Generally, though, dependency lets you do crazy things, like $(n : n > 0) \to bool$. You can see how dependent types lead roughly to program specifications; we'll explore that later, with Cayenne.

The way we'll do this is by extending kinds: if A is a type and k is a kind, then $A \to k$ will be a kind also. In particular we get $A \to *$. Then if $f : A \to *$ and $a : A$, we have $fa : *$. This fa is exactly a type that depends on a term, like we wanted to look at here.

Look at what we get then. Suppose for each $a : A$ we have a type B_a, and $b_a : B_a$. Then we might write $\lambda a : A.b_a$. What's it's type? The Cartesian product of all these B_a, it seems. We'll use a Cartesian product notation to encode this:

$$(\lambda a : A.b_a) : \Pi a : A.B_a \tag{49}$$

This is some complex thing, especially when all the B_a are different. This is quite general; one restriction of this lets us write:

$$(A \to B) = \Pi a : A.B \tag{50}$$

a "dependent product", which is informally B^A by the way—that functions can be associated with exponentiations is a neat observation. We can use this in a redefinition of pseudoexpressions:

$$\mathcal{T} = \quad V \quad | \quad C \quad | \quad \mathcal{T}\mathcal{T} \quad | \quad \lambda V : \mathcal{T}.\mathcal{T} \quad | \quad \Pi V : \mathcal{T}.\mathcal{T} \tag{51}$$

With everything else the same as for $\lambda\underline{\omega}$, here are the typing rules:

$$\text{(axiom)} \quad <> \vdash * : \Box \tag{52}$$

$$\text{(start rule)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}, x \notin \Gamma \tag{53}$$

$$\text{(weakening rule)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}, x \notin \Gamma \tag{54}$$

$$\text{(type/kind formation)} \quad \frac{\Gamma \vdash A : * \quad \Gamma \vdash B : s}{\Gamma \vdash (\Pi x : A.B) : s} \tag{55}$$

$$\text{(application rule)} \quad \frac{\Gamma \vdash F : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]} \tag{56}$$

$$\text{(abstraction rule)} \quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \tag{57}$$

Wow, this looks very similar, except we've replaced arrows with the product notation as discussed. Besides that, what is different? Only the change in the type/kind formation rule, where we fix $A : *$ for both s, rather than having it vary the same way.

- To be explicit, we get functions from $(*,*)$ as before, plus this rule:

$$\frac{\Gamma \vdash A : * \quad \Gamma \vdash B : \Box}{\Gamma \vdash (\Pi x : A.B) : \Box} \tag{58}$$

  which gets us such things as $A \to *$, $A \to B \to *$, etc..

Examples!

$$A : * \quad \vdash \quad (A \to *) : \Box \tag{59}$$
$$A : *, P : A \to *, a : A \quad \vdash \quad Pa : * \tag{60}$$
$$A : *, P : A \to *, a : A \quad \vdash \quad Pa \to * : \Box \tag{61}$$
$$A : *, P : A \to * \quad \vdash \quad (\Pi a : A.Pa \to *) : \Box \tag{62}$$
$$A : *, P : A \to * \quad \vdash \quad (\lambda a : A \lambda x : Pa.x) : (\Pi a : A.(Pa \to Pa)) \tag{63}$$

That's abstraction and application, then type/kind formation, then type/kind formation, then abstraction.

The second to last one there has just abstracted the a away; note this is still a constructor, and we get the third to last one by supplying an a. That last one is giving an identity function for a dependent type; again one supplies an a.

Historically, N.G. de Bruijn introduced this 1970, 1980 to represent theorems and proofs. Elide description because Curry-Howard is a little outside the scope. Suffice it to say that propositions are types and proofs are the terms that inhabit them; inhabited types are proven propositions. And λP is actually named thusly because it can interpret predicate logic. Near to that time, it was expressed in a PL AUTOMATH, intended to automatically verify proofs. Other similar projects in NUPRL (Constable et. al. 1986), LF (Harper et. al. 1987), the calculus of constructions (Coquand and Huet 1988) which forms the basis for Coq. Martin-Löf 1984 does foundational math using this interpretation of types.

VII. TYPING RULE REVISIONISM

Equipped with this product notation, we can actually go back and rewrite $\lambda \to$ and $\lambda 2$. The particular rule we should swap out is the type/kind formation rule. For $\lambda \to$ we only have:

$$\frac{\Gamma \vdash A : * \quad \Gamma \vdash B : *}{\Gamma \vdash (\Pi x : A.B) : *} \tag{64}$$

as in normal function constructor. A note: we kinda sorta have terms depending on types as $\lambda x : A.x$ may be seen to "depend on" A, and kinda have types depending on types as $A \to A$ depends on A. But we do not get function abstraction for this, as in we cannot pull the dependency out into a usable function (in e.g. the manner of $f = \lambda A : *.(A \to A)$) for use. We also do not have types depending on terms in any of the non-P systems since the types must be specified beforehand.

How are these the same? We get the function arrow back by taking

$$A \to B \equiv \Pi x : A.B \tag{65}$$

See that using the pseudoexpression ruleset, any A with $\Gamma \vdash A : *$ is built up from some set $\{B | (B : *) \in \Gamma\}$ using only the $\to$ we defined just now. As stated, application is equivalent to $\to$-elimination.

For $\lambda 2$ we get:

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : *}{\Gamma \vdash (\Pi x : A.B) : s} \tag{66}$$

- That is, besides the typical $(*,*)$ rule that gives functions, also:

$$\frac{\Gamma \vdash A : \Box \quad \Gamma \vdash B : *}{\Gamma \vdash (\Pi x : A.B) : \Box} \tag{67}$$

  which produces $* \to A$, $* \to * \to A$, etc..

This manages to encode both function types and the kind of arrow that you get from a quantification. We see they're the same by beginning with:

$$\forall \alpha.A \equiv \Pi \alpha : *.A \tag{68}$$
$$\Lambda \alpha.M \equiv \lambda \alpha : *.M \tag{69}$$

VIII. THE LAMBDA CUBE

Note now an implicit structure on the type/kind formation rules for all these systems. What happens when we combine them? (One may now draw the lambda cube. Right-handed with x up and y right, get $(\square, *)$ as x, $(*, \square)$ as y, and $(\square, \square)$ as z.)

The systems are defined based on pseudoexpressions:

$$\mathcal{T} = \quad V \quad | \quad C \quad | \quad \mathcal{T}\mathcal{T} \quad | \quad \lambda V : \mathcal{T}.\mathcal{T} \quad | \quad \Pi V : \mathcal{T}.\mathcal{T} \tag{70}$$

There are two constants called sorts $\mathcal{S}$, which are $*$ and $\square$. We have β-conversion and β-reduction from:

$$(\lambda x : A.B)C \to B[x := C] \tag{71}$$

We have statements $A : B$ with $A, B \in \mathcal{T}$, where A is the subject and B is the predicate. A declaration is $x : A$ with $x \in V$ and $A \in \mathcal{T}$. A pseudo-context is a finite ordered sequence of declarations with distinct subjects. The type assignment rules define the axioms for deriving $\Gamma \vdash A : B$. When this is derivable, A and B are full-fledged legal expressions and Γ is a legal context.

Then let us write the type assignment rules for a given λ cube system.

$$\text{(axiom)} \quad <> \vdash * : \square \tag{72}$$

$$\text{(start rule)} \quad \frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}, x \notin \Gamma \tag{73}$$

$$\text{(weakening rule)} \quad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}, x \notin \Gamma \tag{74}$$

$$\text{(application rule)} \quad \frac{\Gamma \vdash F : (\Pi x : A.B) \quad \Gamma \vdash a : A}{\Gamma \vdash Fa : B[x := a]} \tag{75}$$

$$\text{(abstraction rule)} \quad \frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A.B) : s}{\Gamma \vdash (\lambda x : A.b) : (\Pi x : A.B)} \tag{76}$$

Systems also get specific rules of the form:

$$(s_1, s_2) \text{ rule} \quad \frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash B : s_2}{\Gamma \vdash (\Pi x : A.B) : s_2} \tag{77}$$

A listing of specific rules for the systems are:

$$\lambda \to \quad (*, *) \tag{78}$$

$$\lambda 2 \quad (*, *) \quad (\square, *) \tag{79}$$

$$\lambda P \quad (*, *) \quad (*, \square) \tag{80}$$

$$\lambda P2 \quad (*, *) \quad (\square, *) \quad (*, \square) \tag{81}$$

$$\lambda \underline{\omega} \quad (*, *) \quad (\square, \square) \tag{82}$$

$$\lambda \omega \quad (*, *) \quad (\square, *) \quad (\square, \square) \tag{83}$$

$$\lambda P \underline{\omega} \quad (*, *) \quad (*, \square) \quad (\square, \square) \tag{84}$$

$$\lambda P \omega \quad (*, *) \quad (\square, *) \quad (*, \square) \quad (\square, \square) \tag{85}$$

$$\tag{86}$$

So this may seem to have arisen clearly enough at this point. I'd think this is because of the way Barendregt introduced and motivated each system—his motivations for deriving $\lambda \underline{\omega}$ and λP implicitly encoded the lambda cube in terms of "dependency". Actually many systems related to these ones were independently the subjects of previous study: here's a listing Barendregt gives.

1. $\lambda \to$: the simply-typed lambda calculus λ^τ. Church (1940).

2. $\lambda 2$: system F, the second-order typed lambda calculus. Girard (1972), Reynolds (1974).
3. λP: AUT-QE, LF. de Bruijn (1970), Harper et. al. (1987), Longo and Moggi (1988).
4. $\lambda P2$: no related given. Longo and Moggi (1988).
5. $\lambda \underline{\omega}$: POLYREC. Renardel de Lavalette (1991).
6. $\lambda \omega$: Fω. Girard (1972).
7. $\lambda P\omega = \lambda C$: CC, the calculus of constructions. Coquand and Huet (1988).

Barendregt does not include $\lambda P\underline{\omega}$, which is evidently sufficiently unusual that it was not studied (generally the weak ω ones seem pretty strange). Barendregt was the first to observe that they are taxonomized by this nice lattice-looking inclusion structure, though the pure type system framework is able to encode this as well and preceded it by a couple of years.

IX. DEPENDENTLY TYPED PROGRAMMING AND A CODE EXAMPLE FROM CAYENNE

Dependent types present some challenges for practical programming languages. In particular the types are now pretty strong; too strong to infer.

We did not manage to cover this during the presentation, but here's an example from Cayenne. In this language, printf gets to have a type (despite its heterogeneity: in other languages, it can be difficult to properly typecheck the inputs to printf along with its format string).

```
printf fmt = pr fmt "" where
  pr ""           res = res
  pr ('%':'d':cs) res = i -> pr cs (res ++ show (i::Int))
  pr ('%':'s':cs) res = s -> pr cs (res ++ s)
  pr ('%': c :cs) res = pr cs (res ++ [c])
  pr (c:cs)       res = pr cs (res ++ [c])
```

Some code analysis. This uses `pr` to generate a function with a type that depends on the format string. `pr` destructs the format string, adding arguments to a function it's building up when it finds things marked by the %-notation. The output of this function concatenates the argument onto an intermediate, `res`.

Commentary: Cayenne has a bottom $\perp$ which inhabits every type, so it's not consistent as a proof language (though they make mention that they could plausibly check for this kind of thing). They are still concerned about having many sorts (which they refer to using #). In fact there is a reason for this; without the stratification, it's impossible to tell whether an expression corresponds to a type or an actual value at runtime, and the compiler could not strip out the types. (Evidently however one can get some benefits from maintaining types at runtime, like switching on a type.) By the way, if one identifies $*$ and $\Box$, one gets a bad system; this is called Girard's paradox. Coq and Cayenne both make a tower of these things.

Cayenne has other work introducing records and syntactic sugar to make for a practical programming language.