The Lambda Cube and You

Jonathan Sun

March 2015

1 Introduction

One of the most difficult aspects of understanding type theory and programming languages research is making sense of the precise relationships between the abstractions in logic that are usually expressed via lambda calculus and their applications that are ultimately implemented (or purposefully unimplemented) in real-world programming languages. Features of higher-order type theories are often simply expressed by logical rules, but the power and consequence of adding such rules are often difficult to express in terms of more common programming paradigms, or do not yield many simple yet meaningful examples.

This is especially evident in the study of dependent type systems. Because dependent type theories are a relatively new object of study in the realm of logic (though concepts have existed in literature prior), the applications of dependently typed functions have been largely unrealized outside of the PL research community. Furthermore, the concept of dependent typing is more than a single idea; as illustrated by the *Lambda Cube*, a fully dependent type system is actually a combination of simpler concepts in type theory that each uniquely extend the simply-typed lambda calculus. The theory behind the Lambda Cube is well understood, but most existing literature fails to relate the features of the Lambda Cube to dependent type systems in practice. Furthermore, existing dependently typed programming languages usually implement the full cube of features without making the distinction clear between the comprising components. This paper aims to clarify the Lambda Cube's properties from a programmer's perspective by relating the corners and axes of the cube with theory and real-world programming concepts.

2 Dependent Types

Intuitively, a dependent type system allows types that depend on the values of other types. That is, unlike a *simple type* like Int which has a single set of values, a dependent type like Vector[n] actually defines a family of Vector types indexed by the parameter n (in this case, the length). From a logical perspective, dependent types allow quantification over types since predicates can now depend on the types of their free variables. From a practical perspective, dependent types yield more powerful static program checking mechanisms than simple types alone. For example, dependent type checkers can, at compile-time, prove that a program does not run out of memory

bounds. This is achieved by encoding the memory usage as part of an object's type (e.g. the size of a List). At compile-time, the type checker can then verify that all List access operations do not violate the object's type signature, thus *proving* an aspect of the program's correctness.

2.1 Π-Types

The fundamental object in the theory of dependent types is the Π type constructor, pronounced "pi-type":

$$\Pi_{a:A}B(a) = \left\{ f : A \to \bigcup_{a:A} B(a) \mid \forall_{a:A} f a : B(a) \right\}$$

The name Π-type comes directly from the fact that dependent types can be thought of as Cartesian *products* between the types in question, a relationship initially identified by Martin Löf in his type theory published in 1984. From the definition it is clear that for each value in type A, the function f produces a corresponding type $B(a)$. Thus, the whole type $\Pi_{a:A}B(a)$ is the product of all such types $B(a)$. In practice, the simplest examples of Π types include the common examples of simple types parameterised by some scalar value–for example, vectors parameterised by length. Their distinction from ordinary types is clear, since simple type systems do not allow B to depend on a; in essence, they only admit Π-types where B is constant.

2.2 Σ-Types

A less prominently studied but equally important extension of Π-types is the Σ type constructor:

$$\Sigma_{a:A}B(a) = \left\{ f : A \to \bigcup_{a:A} a \times B(a) \mid \forall_{a:A} f a : a \times B(a) \right\}$$

Σ-types express the notion of a dependent *pair*, where elements of the whole type look like $(a, B(a))$ and the type of the second term is allowed to depend on the *value* of the first. Where Π-types correspond to the Cartesian product between types, Σ-types correspond to the *disjoint union*. This is because for each a, f generates a set of pairs $(a, B(A))$, and the whole type $\Sigma_{a:A}B(a)$ is the disjoint union of the family of sets indexed by A. Don't be confused by the appearance of the Cartesian product again–the *elements* of a Σ-type are indeed products (pairs) of the respective types, but the *whole* type as a set is a disjoint union of the family of types indexed by A.

2.3 The Curry Howard Correspondence

The product/disjoint union relationship between Π-types and Σ-types is of theoretical importance, as they correspond directly to universal quantification and existential quantification under the Curry-Howard correspondence. In fact, the Π-type can also be written as:

$$\forall(a : A).B(a)$$

The logical interpretation of this type is a set of functions that, *for all* values of a produce a corresponding type $B(a)$. Likewise, the Σ-type can also be written as:

$$\exists(a : A).B(a)$$

The logical interpretation here is a set of pairs $(a : A, b : B(a))$ that each acts as a *proof* of proposition B, in the sense that *there exists* some element a which is a witness to having the property $B(a)$.

Contemporary implementations of dependent type systems rarely resemble the notation originally put forth by Löf. These abstract constructions also do little to reveal the richness nor the restrictiveness of the language of dependent types. The most prominent unanswered question is the range of values for A and B. Does it matter whether A is restricted to simple types, or can A in fact be a set of dependent types parameterised by a third set C? The Lambda Cube helps clear up the confusion by providing a fine-grained breakdown of the possibilities and their relationship to each other.

3 The Lambda Cube

Introduced by Henk Barendregt in 1991, the Lambda Cube is a way to visualize the different aspects of dependent type theory and identify an inclusive hierarchy of features that dependent type systems have. While not essential to understanding or using dependent types in practice, the Lambda Cube provides a good way to understand the relationship between features without the confluence of notation or implementation. Originally the Lambda Cube was based on the *Calculus of Constructions*, an alternative type theory developed by Thierry Coquand in 1988 that, like Löf's, is dependently typed. Though formulated later, the Calculus of Constructions became more widely used in practice with the development of the Coq programming language. However, the theoretical aspects differ only in notation and are important to study in generality; properties of the Lambda Cube apply to contemporary dependently typed programming systems as well.

3.1 Axes of Abstraction

The main purpose of the Lambda Cube is to systematically categorize a wide variety of type systems according to a few high level features and identify inclusive relationships between them. In the above figure, each of the eight corners of the cube represents an abstract class of type systems, and each of the arrows denotes inclusion, i.e. each type system has strictly more expressive power than the type system preceding it in the arrow hierarchy. At the bottom left corner of the cube, denoted $\lambda{\to}$, is the *simply-typed lambda calculus*, the simplest form of lambda calculus with the ability to express types. At the top right corner of the cube, denoted $\lambda P\omega$, is the full Calculus of Constructions, a far more expressive system. Each of the arrows travel along one of three independent axes (hence forming a three-dimensional cube), and each of the axes represents one particular refinement of type "dependency". These refinements can be intuitively described in four ways: *terms depending on terms*, *terms depending on types*, *types depending on types*, and *types depending on terms*.

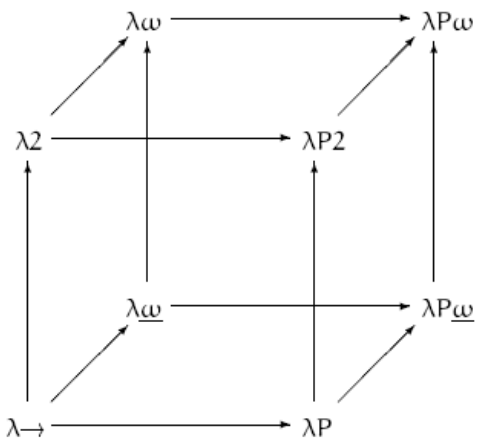

Figure 1: Barendregt's Lambda Cube

3.1.1 Terms Depending on Terms

The simplest form of dependency in type theory, terms depending on terms, is so fundamental that there is no axis for it on the Lambda Cube. Every function can be thought of as a dependency between terms, so any type theory that supports propositional logic (virtually every studied one) supports terms depending on terms. Simply having this property, however, restricts the type system to the expressive power of the simply-typed lambda calculus. While academically useful to the study of logic, the simply-typed lambda calculus cannot express basic concepts like a universal identity function (every type in a simply-typed system must have its own identity function for elements). The other three refinements each extend the simply-typed lambda calculus in a perpendicular manner, and their combinations yield the entire cube as shown above.

3.1.2 Terms Depending on Types

The type dependency most commonly encountered in real-world programming is represented by the vertical axis of the Lambda Cube, where terms depend on types. More commonly known as *polymorphism*, this class of type system supports functions whose result depends on the types of its parameters. The most fundamental example of such a function is the universal identity function for typed lambda calculi:

$$I := \lambda x : A.x$$

Note that the type of the output depends on (i.e. is identical to) the input type, which is not part of the simply-typed lambda calculus. Polymorphism alone is powerful enough to warrant implementation in the most common functional *and* imperative programming languages. For example, in Java, polymorphism generally refers to function behavior when *subclassing*:

```
class Vector extends Container
```

```
class LinkedList extends Container
Container c = ...
c.size()
```

Here the *size* function's behavior depends on whether the input parameter is actually of type *Vector* or *LinkedList* (or some other subclass of *Container*). However, another aspect of polymorphism actually appears in Java in the form of *function overloading*:

```
Boolean equal(String a, String b)
Boolean equal(LinkedList a, LinkedList b)
```

Most imperative programmers simply think of this as two distinct functions that happen to share a name, but in the functional programming sense they can be thought of as one function whose result depends on different types of input. In Haskell, a functional programming language, polymorphism is fundamental and can be achieved more elegantly:

```
length :: [x] -> Integer
length [] = 0
length (first:rest) = 1 + length rest
```

In this example, the *length* function can apply to lists of any type. On the Lambda Cube, adding polymorphism to the simply-typed lambda calculus results a type system capable of expressing second-order *propositional* logic, since polymorphic functions correspond directly to the logical notion of universal quantification over propositions (simple types). Thus, the new type system is denoted $\lambda 2$ on the Lambda Cube. Note that this is not capable of expressing second-order *predicate* logic, which requires quantification over *terms*.

3.1.3 Types Depending on Types

The "diagonal" axis shown on the Lambda Cube corresponds to the addition of types depending on types, also described in terms of *type operators*. Intuitively, type systems with this feature enable functions to produce types depending on other types. Note that this is distinct from the previously discussed notion of subclassing, since the subclasses themselves are not *dependent* on any external type. The most common form of type operator is the generic type constructor, with an example from the dependently-typed functional programming language Agda:

```
data _X_ (A B : Set) : Set where
<_,_> : A -> B -> A X B
```

Here, "X" denotes the Cartesian product and the declaration defines a Pair data type that can generically apply to *any* arbitrary types A and B (at least, any A and B belonging to *Set*). In later versions of Java (1.5+), a restricted version of this feature was added in the form of generic classes:

```
class Set<T>
class Map<K, V>
```

The Set class is now parameterised to only contain elements of a specific type T. Similar to the Pair example in Agda, the Map class in Java depends on two types for its keys and values, which can themselves be generic classes (e.g. Maps between Sets). The most important benefit of the added syntax is that the compiler will be able to type check all of a Map's operations to ensure that key and value insertions are actually of the correct type. Without generic programming, this would not be possible since type checking could not be done separately for an infinite class of typed Maps, but a non-generic Map could still be implemented (with all the risks of no type checking).

Although type operators can be applied to the simply-typed lambda calculus alone, in practice such type systems (denoted $\lambda\underline{\omega}$ on the Lambda Cube) are rarely studied. $\lambda\omega$, which combines both polymorphism and type operators, is far more commonly used, as together the two axes are sufficient to describe higher order propositional logic. This is also notably evident in real-world programming languages, which usually support either polymorphism alone or polymorphism with generic functions. Programming languages with generic functions only are rarely seen. The most important observation before moving to the last axis is that all four type systems represented on the left side of the cube model *propositional* logic systems–without quantification over terms, *predicate* logics like the familiar first-order logic are not yet expressible.

3.1.4 Types Depending on Terms

The final direction of the Lambda Cube is the most important distinction for dependent typing and forms the lateral axis in the diagram. Intuitively the idea of types depending on terms is similar to the idea of generic types, but now the parameter can be a value of a type instead of just a type. The canonical example is a vector type signature that depends on its length:

```
data Vector (A : Set) : Nat -> Set where
[] : Vector A zero
_::_ : {n : Nat} -> A -> Vector A n -> Vector A (succ n)
```

In the above Agda example, the Vector type is parameterized both by a type A (which identifies the type of elements allowed in the Vector) and a *value* of type Nat (which identifies the length of the vector). The type signature plays an important role in the constructors, as now an empty Vector has a type signature with length zero, while a *cons* operation on a Vector of length n produces a Vector of length $n + 1$. This kind of construction is notably absent in imperative programming languages like Java, where a declaration would look like:

```
class Vector<T><n>
```

In this case, problems arise in the imperative programming paradigm because modifiers to the Vector class would inherently change its type signature, which partially explains why they are rarely seen. Functional programming languages do not suffer from this limitation since their functions are generally characterized by their return value, not their side effects, and thus see dependent type support more often.

Though not widely used in practice yet, even simple dependent types have important potential applications in real-world programming. When implemented with dependent types, access operations on Vector can be type checked at compile-time to ensure they do not access an out-of-bounds index. Parameterising on a scalar value–in this case, an integral length–is the most common form of types depending on terms because it is the most natural extension of generic types to an application for which there is a strong demand in program verification. After all, any kind of container class (Set, List, Map, etc) can be made more secure by statically checking for access bounds. However, more sophisticated examples demonstrate the expressive power of dependent types well beyond scalar values:

```
data Image _∋_ {A B : Set}(f : A -> B) : B -> Set where
im : (x : A) -> Image f∋fx
```

This Agda example defines a data type Image that depends on two type parameters A and B and a value parameter f which is of type $A \to B$, i.e. a function between the two types. The Image data type has one constructor which takes a value of A and applies f to it (note that f is now part of the type signature). As originally seen with Π-types, Image produces *for all* types A and B in Set a family of types indexed by the set of all functions between A and B.

On the Lambda Cube, the addition of dependent types to the simply-typed lambda calculus results in a type system called λP, which logically corresponds to first-order predicate logic. The logical hierarchy from the left side of the Lambda Cube can then be extended to the right side, in the sense that adding polymorphism (the vertical axis) to λP results in second-order predicate logic, while adding generic types (the diagonal axis) on top of that results $\lambda P\omega$, the full Calculus of Constructions. Interestingly, any type system that allows all four dependencies between types and terms yields an alternative interpretation of type theory in which types and terms are the same thing, since terms can be used to type other terms and types can be used as values within terms. This finally provides a clearer picture of Π-types, where A and B can range over all terms (including other Π types).

Although the early dependent type systems did not distinguish between the different flavors of dependency, the term "dependent types" now usually refers to type systems with types depending on terms, while type systems with the other axes are called *parameterised types* to distinguish them. In practice most modern programming languages support parameterised types in some form, but rarely do languages implement fully dependent types in this sense. From a logical perspective, dependent types allow universal and existential quantification over terms, which yields a dramatic shift between the propositional logics on the left side of the Lambda Cube and the predicate logics on the right.

4 Conclusion

This paper started by introducing the intuition of dependent types and defining the fundamental constructors, Π and Σ, by which dependent types can be constructed in lambda calculi. While simple to write down, the Π and Σ constructors are far too general to give a sense of what is going

on structurally with dependent type system. Furthermore, the abstract nature of their parameters makes it difficult to identify features of dependent typing in real-world programming systems, where types and terms often do have important distinctions. The introduction of the Lambda Cube, while not an immediate characterization of dependent types, greatly aids in understanding the components of a dependently typed system and how they are constructed from the most basic lambda calculi. The cubic structure is important–following the inclusion arrows yields a hierarchy of type systems that is non-linear; rather, a wide variety of type systems can be formed by choosing what flavors of dependency are allowed between types and terms. Logically, the most important takeaway is the fact that the left side of the Lambda Cube represents a hierarchy of propositional logic systems, while the right side represents each corresponding analogue in predicate logic.

As a final note, while the Lambda Cube is a very useful tool for classifying and understanding the relationship between the basic dependent type systems, the theory of dependent types does not simply end at the cube boundaries. In terms of generality, all the type systems in the Lambda Cube fall into the general category of *pure type systems*, which extend the notion of dependent types to *sorts* and higher order abstractions. The study of pure type systems is quite abstract and outside the scope of this paper. In terms of granularity, the Lambda Cube highlights the basic forms of dependency between terms and types (the most important concepts in regards to classifying dependent type systems) but does not distinguish very specific language features like *subtyping*, which is a specialized form of polymorphism. Lastly, it is important to remember that from a programming perspective, the most important application of these type systems is to provide language features in the form of static type-checking and correctness guarantees that correspond to each axis of the Lambda Cube.

References

[1] Peter Dybjer, Ana Bove, *"Dependent Types at Work." Language Engineering and Rigorous Software Development*. Piriapolis, Uruguay, 2008.

[2] Simon Thompson, *"Type Theory & Functional Programming"*. Computing Laboratory, University of Kent, 1999.

[3] Henk Barendregt, *"Introduction to Generalized Type Systems"*. Journal of Functional Programming 1 (2): 125-154 Catholic University Nijmegen, The Netherlands, April 1991.

[4] Maria João Frade, *"Calculus of Inductive Constructions: Software Formal Verification"*. [slides], Universidade do Minho, 2009.

[5] Martin Löf, *"Intuitionistic Type Theory"*. Padua, Italy, 1980.

[6] Ulf Norell, *"Dependently Typed Programming in Agda"*. Lecture Notes from the Summer School in Advanced Functional Programming, Chalmers University, Gothenburg, 2008.