Surveying the Lambda Cube

Bryce Hickle

15 December 2020

Introduction

The Lambda Cube, as described in, "Lambda Calculi with Types" by Henk Barendregt, is an essential starting point for any novice computational logician. It encompasses the work of many of the pioneers of modern type theory. By performing a literature survey and gaining a clear understanding of this model, one will be well-equipped for more advanced topics in theoretical computer science.

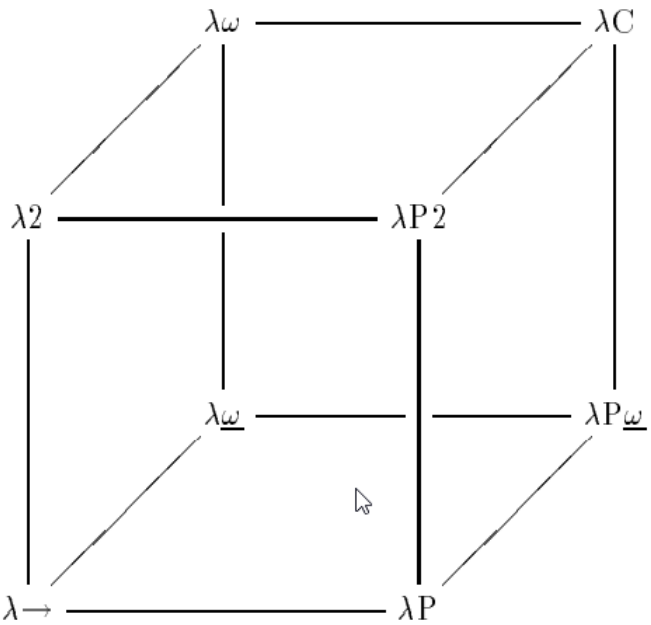

Figure 1: The Lambda Cube

Above is a visual representation of the Lambda Cube from Henk Barendregt's textbook [1]. Figure 1 summarizes the features of the eight type systems and their relations to each other [1]. The trade off to such a concise summary is the barrier to entry to understanding. To be more clear, one might wonder which type systems are being compared exactly, why is the cube a three-dimensional directed graph, what is the significance of the arrows, and why is there a source and sink node to the model. Each node will be analyzed individually, and the dimensions will be discussed later as well. However, now may be a good time to evaluate the arrows.

The arrows are strict inclusion relations [1]. That is, the node that is receives the arrow has all the type features of the source of the arrow but with additional type features built on top [1]. That is why there is a source

node, it has the most basic typing. This also explains the sink node it has all of the type features addressed by Dr. Barendregt in section 5.1 of "Lambda Caluli with Types" [1]. Next, let us survey the origin point for the Lambda Cube and modern type theory.

Dimensions and Dependencies

The Simply Typed Lambda Calculus (Lambda Right Arrow)

The type system that is The Simply Typed Lambda Calculus is as good a place as any to start studying type systems. It has clear use in programming languages (most programming languages have some notion of variable and function typing which is all simple types consist of) [4]. It is the point of the Lambda Cube where no arrows enter [1]. It is our origin. Below are some examples of terms typed using simple types.

```
id : Nat → Nat = λ x. x.
```

Figure 2: The Natural Identity Function

```
notId : ( Nat → Nat )  → ( Nat → Nat ) = λ f. λ x. f ( f ( f x ) ) .
```

Figure 3: Simply-Typed Expression

```
notId2 : ( Bool → Bool )  → ( Bool → Bool ) = λ f. λ x. f ( f ( f x ) ).
```

Figure 4: Another Simply-Typed Expression

We can see, from the above examples, easy to read types for each function. A problem does arise when looking at notId and notID2 though. The programs of each are identical! The general structure of the types for both notID and notID2 are identical. Only the specific base types used differ between the two. Simple typing creates redundant code that is in need of abstraction which can be handled in more advanced type-systems that will be discussed.

On other note regarding Simply Typed Lambda Calculus, we already have a form of dependency referenced in, “Lambda Calculi with Types” in the form of terms depending on terms [1]. This kind of dependency does not get its own dimension, but the other dependencies discussed will [1].

Polymorphic Typed Lambda Calculus (Lambda 2)

These interesting issues of redundant code that is only simply typed can be resolved with polymorphic typing. This kind of typing allows for terms to depend on types [1]. Barendregt formalizes this with a the use of the

universal quantifiers at the type level and the capital lambda at the term level which can be seen in the Cedille examples below (figures 6 - 9).

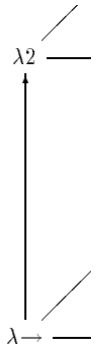

Figure 5: Polymorphic Type Dimension

```
id : Nat → Nat = λ x. x.
```

Figure 6: The Natural Identity Function

```
notId : ( Nat → Nat ) → ( Nat → Nat ) = λ f. λ x. f ( f ( f x ) ) .
```

Figure 7: Simply-Typed Expression

```
notId2 : ( Bool → Bool ) → ( Bool → Bool ) = λ f. λ x. f ( f ( f x ) ).
```

Figure 8: Another Simply-Typed Expression

```
notIdPolymorphic : ∀ X : ★. ( X → X ) → ( X → X ) = Λ X. λ f. λ x. f ( f ( f x ) ).
```

Figure 9: A Polymorphically-Typed Expression

The notIdPolymorphic example clearly shows the utility of this type feature, solving the issue of its two predecessors. This node of the Lambda Cube is the System F node and forms the first dimension discussed so far [1]. Now, onward to the type system famously used for mathematical theorems!

Dependently Typed Lambda Calculus (Lambda P)

The node Lambda P is the typing that has both simple types and dependent types. This type system allows types to depend on terms (the inverse of polymorphic typing) which is another dimension of the Lambda Cube [1]. This kind of type system was famously used as part of the Automath languages to proof check mathematical theorems through the use of capital Pi at the type level for quantifies over sets of terms [1, 2]. The examples below showcase how these types clarify the exact proposition being proven.

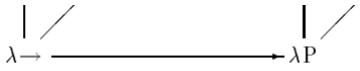

Figure 10: Dependently Typed Dimension

```
notId : ( Nat → Nat )  → ( Nat → Nat ) = λ f. λ x. f ( f ( f x ) ) .
```

Figure 11: Simply-Typed Expression

```
id3 : Π n : Nat. { id n ≃ n  } = λ n. β.
```

Figure 12: Dependently-Typed Expression

Cedille is helping quite a bit with its curly bracket notation for propositions. None-the-less, id3 clearly shows that the identity function applied to any natural number is the same as the original natural number (in this case n). Of course, under the Curry-Howard Isomorphism, every type mentioned including simple types correspond to a proposition; and its respective program, a proof of such a proposition [4]. However, including terms in the type helps make the proposition more understandable to the working mathematician or logician [1, 4]. Interestingly, Bruijn considered Automath useful only from a purely mathematical standpoint stating, "It is not a programming language, although it has several features in common with existing programming languages," [2]. The final dimension of the Lambda Cube is ahead.

Type Operators for Lambda Calculus (Lambda Omega Weak)

The third dimension of the cube is the dependency of types on types [1]. This type feature is often called type operators [1]. This allows for typing functors. I made a slightly none-traditional List functor to demonstrate below.

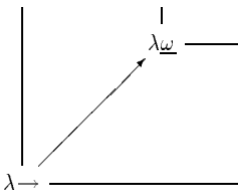

Figure 13: Type Operator Dimension

```
notId : ( Nat → Nat )  → ( Nat → Nat ) = λ f. λ x. f ( f ( f x ) ) .
```

Figure 14: Simply-Typed Expression

```
ListF : ★ → ★ → ★ = λ A: ★. λ R: ★. Sum ·Unit ·(Pair ·A ·R).
```

Figure 15: Type Operator Example

Before continuing, a mention of kinding is in order. In the Lambda Cube, types are of kind * or recursively defined with the arrow constructor, similarily to constructing simple types [1, 4]. Types of kind that have at least one arrow can take a type as input such as functors. Kinds themselves belong to [], the second sort besides kinds (which are axiomatic objects used to build up the type-system) [1]. The functor example above is unique because it has kind * -> * -> *. One could also make a list functor with kind *->* which is the normal appoach. The use of this setup is that ListF can take other functors as input to output another type. An additional way to phrase this feature in more common terms is that kinding allows for a programming language to have type classes[1].

Combinations of Type Features

Now that we have the dimensions and some examples examined, the rest of the nodes are much easier to comprehend. They are just type-systems that have some combination of the four dependencies mentioned previously [1]. The remaining nodes on the top half of the cube have polymorphic typing [1]. The four nodes on the back face of the Lambda Cube have type operators [1]. The four nodes on the right face of the Lambda Cube have dependent types[1].

The most important of the remaining nodes is Lambda C. The type system of Calculus of Constructions [1, 3]. This type system is famously powerful and used in programming languages such as the proof checker Coq [1, 3]. Importantly, Coquand and Hue prove that the Calculus of Constructions is strongly normalizing in the famouse paper, "Calculus of Constructions" [3].

Discussion

The use of the Lambda Cube is far more reaching than just examining typings that can be imposed on lambda calculus terms. Any programming language could be placed on the cube based on the type features the language has. For example, Robert Widmann placed Swift as being between System F and Lambda Omega based on the lack of dependent types and the restrictions on type classes for the language swift [5]. The language Haskell would have a Lambda omega typing because of its polymorphic typing and type operators but lack of dependent types. I would strongly encourage using the Lambda Cube as a model for very quickly comparing languages based on the properties of their type system.

Conclusion

The Lambda Cube is an elegant model of modern type systems and makes for an excellent entry point into computational logic and type theory. Personally, this survey has given me a fresh perspective on the programming languages respective to their type functionality, and I think that the wide range of topics summarized by the Lambda Cube will serve me well in future studies. Hopefully, the related presentation to this survey presented in lecture will serve others in their studying as well.

References

[1] Henk Barendregt. *Lambda Calculi with Types.* Oxford University Press.

[2] N. G. de Bruijn. Automath, a language for mathematics. 1983.

[3] Coquand Hierry and Gerard Hue. Calculus of constructions. 1988.

[4] Aaron Stump. *Introduction to Lambda Calculus.* University of Iowa, 2020.

[5] Robert Widmann. A type system from scratch, 2017.