Dependent Types and Martin-Löf Type Theory: An Introduction

Carl Factora

April 9, 2016

Abstract

Who knows.

1 Introduction

Introduce the contents of the paper, notation, talk about Agda.

2 Dependent Types

In this section, we introduce the concept of dependent types and their role in type theoretic systems. At their core, dependent types can be thought of simply as an *interplay of types and terms*. To understand this relationship, it helps to think of the ways in which types and terms can *depend* on each other. Altogether, there are four different kinds of dependencies:

1. terms depending on terms
2. terms depending on types
3. types depending on types
4. types depending on terms

These dependecies, however, should not be confused with an actual dependent type. In fact, only (2) and (4) correspond to expressions that are typed (necessarily) with a dependent type (we explain later why (1) and (3) are not typed with dependent types).

What follows is a brief description of each form of dependency and the corresponding λ-calculus that is associated with it. With this, we introduce the (fully-dependent) system of the *Calculus of Constructions* (λC). Here, the word *fully* signifies that λC features all four forms of dependency mentioned above.

Simply-Typed (STLC)

Role: Base System

The STLC is not a dependently typed system. In fact, the term *dependent type* is drawn in constrast with the term *simple type*, where the STLC derives its namesake. A simple type is either a *primitive type*, which is the set of all types "pre-defined" within the system, or a *function type* between other simple types. Thus, simple types have the following grammar:

$$\texttt{t} \mid\mid \texttt{t} \to \texttt{t}$$

The terms of the STLC are simply the terms of the *untyped λ-calculus*, with the exception of terms that use *self-application* (i.e., Y combinator, Omega, etc.). This has the benefit of removing all *infinite* calculations from the system but also removing all recursive computations. Thus, the STLC is not *Turing complete* and is thus the appropriate foundation for deriving a type theoretic system. It must first, however, be extended with more expressive types, as the only form of dependency present in the STLC is (1), terms depending on terms (i.e. *term level* λ-abstraction).

The key difference between the STLC and λC are the places in which terms and types are allowed to be. In the STLC, terms and types are entirely seperate entities in that terms only exist on *term level* and types on *type level*. Thus, as we mentioned earlier, this puts a constraint on the system to only allow λ-abstraction to happen at term level (viz. definition of simple type). For example, the *constant function* over types $\mathbb{N}$ and $\mathbb{B}$ in the STLC looks like:

$$\begin{aligned}&\texttt{const} : \mathbb{N} \to \mathbb{B} \to \mathbb{N}\\ &\texttt{const} = \lambda\ \texttt{n}\ \texttt{b} \to \texttt{n}\end{aligned}$$

Given that `const` is of type $\mathbb{N} \to \mathbb{B} \to \mathbb{N}$, the term that inhabits this type is λ `n b` $\to$ `n`. In this example, we can see that terms and types are treated differently and are only related in terms of *inhabitance*. This relation is reason for the limited expressivity of the STLC and is also the method in which the STLC can be extended into λC.

Second-Order ($\lambda 2$)

Role: Addition of polymorphic functions

Another name for $\lambda 2$ is System F. For our purposes, we treat $\lambda 2$ as simply an extension of the STLC. This system is adds *polymorphic* terms to the STLC (viz. Dependency (2)). For example, let's take the definition of `const` above and change it to define `polyConst`:

```
polyConst : (A B : Set) → A → B → A
polyConst A B = λ a b → a
-- Works for any type A and B
```

Unlike `const`, `polyConst` takes four arguments, where the first two are types. These two types (viz `A` and `B`) are the corresponding types of the third and fourth arguments of `polyConst`. To clarify, we can instantiate `polyConst` with $\mathbb{N}$ and $\mathbb{B}$ resulting in a function of type $\mathbb{N} \to \mathbb{B} \to \mathbb{N}$, giving us

$$\texttt{const} = \texttt{polyConst}(\mathbb{N}, \mathbb{B}).$$

Thus, adding polymorphism to the STLC has the effect of extending the notion of type and term. With $\lambda 2$, we are now allowed to pass types as arguments at term level (viz. we pass the actual types $\mathbb{N}$ and $\mathbb{B}$). It is, however, necessary to constrain this expressivity with a more expressive type (e.g. `(x : A)` $\to$ `...`). Types that appear in this fashion are called Π-types. This leads us to a new grammar for types (`t`):

```
t || t → t || ∀ (x : Set) → t
```

In general, the $\forall$ symbol is used to express Π-types

Briefly, Π-types are a form of dependent type that are inhabited by polymorphic functions. In cases like `polyConst`, Π-types are necessary to signify the treament of a polymorphic function and to prevent the appearance of free variables within type expressions. For example, if we had typed `polyConst`

as `A` $\rightarrow$ `B` $\rightarrow$ `A`, then `A` and `B` are free variables in that they, by themselves, have no *meaning*. This is why `polyConst` is first passed two types to instantiate the meanings of `A` and `B`. In a sense, a Π-type binds values for types just as λ-abstraction binds values for terms.

Π-types are actually significantly more expressive than the ones in $\lambda 2$. In $\lambda 2$, they are constrained to be only parameterized over other types (e.g. `t` in $\forall$ `(x : Set)` $\rightarrow$ `t`). In a later system, we allow Π-types to be parameterized over terms (e.g. $\forall$ `(x : t)` $\rightarrow$ `t`).

So far, we've seen how terms can depend on other terms (STLC) and how terms can depend on types (term level polymoprhism). In the following system, we introduce a small, yet important dependency: types depending on types.

Higher-order ($\lambda\underline{\omega}$)

Role: Addition of type constructors

To clarify, $\lambda\underline{\omega}$ is generally not a standalone system, instead, it is generally treated as an "add-on" system. Later, we model the relationship of each system to λC. For the sake of time, we treat $\lambda\underline{\omega}$ as an extension of $\lambda 2$, giving us the following transition:

$$\text{STLC} \rightarrow \lambda 2 \rightarrow \lambda\underline{\omega} \rightarrow \lambda\omega$$

Thus, the system we are introducing is $\lambda\omega$, which, unlike $\lambda\underline{\omega}$, features a more diverse grammar of types (i.e., $\lambda\underline{\omega}$ does not have Π-types). Given its $\lambda 2$ component, another name for this system is System Fω. Before we introduce the new grammar for types, we first introduce the concept of *levels*.

We are already familiar with the first two levels, terms and types, which, from the STLC, are still related in terms of inhabitance. To meet the constraints of the STLC when introducing type constructors to our system, we must also provide the type of types. For our purposes, we assume an infinite universe of types, as is common practice in deriving type theoretic systems to avoid certain paradoxes. For example, let's first assume that natural numbers are primitive within our system. We then have the following for the natural number 42:

$$\texttt{42} : \mathbb{N} : \texttt{Set} : \texttt{Set}_1 \ \ldots$$

Another name for `Set` is a *kind*. For our puroses, we use the more familiar set-theoretic equivalent reading. Reading the above example is rather straightforward: 42 is of type $\mathbb{N}$ which is of type `Set` which of type $\texttt{Set}_1$ and so on. There is, however, an important distinction for the first two levels: 42 is NOT of type `Set`, while $\mathbb{N}$ is both of type `Set` and $\texttt{Set}_1$. Terms are therefore treated uniquely, since they only exist as elements of types. While important, this topic is a bit out of the scope of an introductory paper, however, we briefly discuss this in a later section.

We can now revise the definition of types (`t`) to include all instances of `Set` in our infinite universe. We notate this as `Set` below:

```
t || t → t || ∀ (x : Set) → t || Set
```

The curious effect of doing this allows us to construct expressions of type `t → t`, where either `t` can have subexpressions containing `Set`. Let's take a familiar definition of the list data structure:

```
data List (A : Set) : Set where
  Nil  : List A
  _::_ : A → List A → List A
```

We discuss `List` more in depth in a later section. For now, we simply ask *what is the type of* `List`*?* To answer this question, we can simply look at the first line of the definition above (viz. `List (A : Set) : Set`). From this line, we can see that `List` is a type parameterized over any type `A`. Another way to think about this statement is to think of `List` as a form of polymorphic function of type `Set → Set`, as is similar to other *parameterized types*, that can be applied at type level (thus the name *type constructor*). This is to insure that we cannot have create type expressions that fail to instantiate `List` (i.e., `List → A`).

From this we can see that the simple addition of `Set` into our grammar of types has great effect on the expressivity of our system. There is only one other extension necessary to consider our system fully-dependently typed, which, similar to our treatment of $\lambda\underline{\omega}$ is also deceptively simple.

Predicates (λP)

Role: Encoding of Propositional logic

To review, if we compile all the previously mentioned systems above, we have the system $\lambda\omega$ (System Fω). Our final addition, the system of λP,

allows us to construct *type families* which are meant to directly correspond to propositions in higher-order logic.

Let's take a simple example. Let's define a proposition `isEven` that takes a natural number. Another way to say this is that `isEven` is a predicate over natural numbers. When writing propositions, we write them as the equivalent type definition (similar to `List`). Consequently, the proposition `isEven` is an example of a dependent type, since its *meaning* depends on the natural number passed to it.

```
data _isEven : ℕ → Set where
  ZEven : 0 isEven
  NEven : ∀ n → n isEven → (Suc (Suc n)) isEven
```

This is the last form of dependency of the four we mentioned earlier. Here, we've defined two ways in which `n isEven` for all natural numbers (`n`) can be proven "true", which is done via its two type constructors `ZEven` and `NEven`, corresponding to the two constructors of ℕ. `ZEven` corresponds to the proof that 0 is an even number. `NEven` then defines the property that if a given natural number (`n`) is even (viz. `n isEven`), then `n + 2` is even (viz. `(Suc (Suc n)) isEven`). To clarify how this type can be used, let's prove that 4 is even. We start by providing the appropriate type definition and also providing a name for our proof (viz. `4isEven`):

```
4isEven : 4 isEven
```

We must then show that 4 is indeed even using a combination of `ZEven` and `NEven`. From the definition of `isEven`, we know that to prove 4 is even, we must know that 2 is even. Consequently, to prove 2 is even, we must know that 0 is even. Fortunately, we have a proof that 0 is even: `ZEven`. Thus, the proof that 2 is even is `NEven 0 ZEven`, and finally the proof that 4 is even is:

```
4isEven : 4 isEven
4isEven = NEven 2 (NEven 0 ZEven)
```

This method of proof is adopted from *Intuitionistic Logic*. In this form of logic, "truth" corresponds to the idea of *type inhabitability*. This makes sense since we use define types to correspond to logical propositions (e.g. `isEven`). "Falsity" then naturally corresponds to un-inhabitability of a given type. This can be seen if we try to prove that 5 is even using `isEven`. The proof seems to get "stuck."

While it might seem intuitive and more effective to instead try to prove that 5 is *not* even, this, however, also doesn't work. This is due to the treatment of negation in Intuitionistic Logic. To prove that 5 is not even is to prove that $\neg(5\ \texttt{isEven})$, which is equivalent to constructing a function of type:

$$\texttt{5 isEven} \to \bot$$

That is, given a proof that `5 isEven`, construct $\bot$, the empty type. This is not possible since we cannot ever construct a proof that 5 is even. We discuss the empty type later in Section 3.

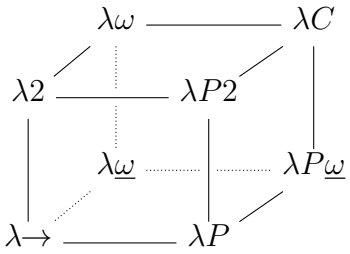
Calculus of Constructions and Remarks

To conclude our discussion over dependent types, we present the final grammar for types after extending $\lambda\omega$ with λP:

$$\texttt{t || t} \to \texttt{t || } \forall\ \texttt{(x : t)} \to \texttt{t || Set}$$

It should come with no surprise that the change is minor. We simply have lifted the restriction on Π-types and allowed them to be parameterized over both terms and types.

In the process of introducing dependent types, we have shown the necessary changes and extensions on the STLC to achieve the expressivity necessary to function as a foundation for mathematics. To clarify this process and to show the relationship between each of the systems we have previously mentioned, we include a diagram of the *Lambda Cube*:



Reading the cube requires us to start on the edge marked $\lambda\to$, which corresponds to the STLC, the base system. From here, we can either move inwards, upwards, or rightwards, which corresponds to the extension to the

other systems marked on the respective edges. Using only these three movements, we eventually reach λC, leaving it in the *most powerful* position on the cube.

3 Martin-Löf Type Theory

In this section, we introduce the foundational concepts of Martin-Löf Type Theory. We discuss recursion, induction, and as well several primitive type encodings (e.g. from propositional logic). We recommend the reader to have a running proof assistant while reading this section to help fortify (better word) several concepts.

Principles

Recursion and induction are one of the powerful tools available in a fully-dependently typed system. Using recursion and induction principles to define functions and proofs, we are guaranteed termination and totality. This is because, in general, recursion and induction principles are defined under certain known intuitions on the given type that they are meant to act on and thus also follow a certain pattern.

```
recT : (C : Set) → C₀ ... → T → C
indT : (C : T → Set) → (C₀ : ...) ... → (t : T) → C t
```

The recursion and induction principles for some type `T`.

The first argument given to these principles is sometimes called the *motive*. This simply means that `C` is the type we are going to inhabit. For recursion principles, this means the given output of the function we are defining, and for induction principles, this can be read as the proposition we are trying to prove. The terms that follow (viz. $\texttt{C}_0$) correspond to the constructors of the given type, which is how these principles guarantee totality. Finally, both principles take a term of the type that are acting on (viz. `T` and `(t : T)`). The only difference is that the induction principle introduces a type family, as opposed to non-parameterized type in the recursion principle. We discuss this in Section 4.

In this section, we discuss the recursion and induction principles, `rec` and `ind`, for each *primitive type*: $\bot$, $\top$, $+$, $\times$, and Σ. The symbols $+$ and $\times$ signify disjunction and conjunction and, thus, should not be confused with addition and multiplication. For the latter, we use the names `plus` and `times`.

Primitive Type Encodings

4 Example Types and Sample Proofs

Describe the section.

Booleans

Natural Numbers

List/Orderings

Identity Types

5 Discussion

Say a few things about the paper.

Extension to Homotopy Type Theory