Modern Type Theory

An introduction to modern dependent type theory

Ridan Vandenbergh

KU Leuven

1980-01-01

1.1. Overview

Why study types?

- to introduce **rigorous correctness** to programming languages
- to understand the core theory in **proof assistants** like Agda, Coq, ...
- to formalize a **new set of foundations** for mathematics[1]
- to draw commutative diagrams you can be proud of

[1] As an alternative to ZFC set theory

1.1. Overview

What we'll discuss in this course:

- The design of a good type theory
- Logic as an **emergent property**
- Martin-Löf (extensional) type theory
- Concepts from category theory presented in type theory

1.2. Set theory vs type theory

As we present type theory as a suitable replacement for set theory as a foundation of mathematics, it's worth nothing the differences:

Set theory	Type theory
Comprised of two layers: First-order logic with axioms for the theory on top	Is its own deductive system: no dependency on logic
Two basic notions: sets and propositions	One basic notion: *types*
One deductive outcome[1]: A has a proof.	Several deductive outcomes: introducing types, variables, ...

[1] We'll start calling these *judgements* soon

2.1.1. Judgements

One word we'll come across often is **judgement**:

Judgement

A **judgement** is the *assertion* or *validation* of some **proposition** P, hence the claim of a **proof** of P.

To distinguish a proposition from a judgement, the turnstile symbol is used:

$$\vdash P \qquad \textit{I know } P \textit{ is true.}$$

If our "proof" of P is based on assumptions, we write those in front of the $\vdash$:

$$A \vdash P \qquad \textit{From } A\textit{, I know that } P.$$

2.1. Deductive system

Judgements form a core building block of our type theory.

As seen in the introduction, type theory is its own deductive system. Let's define that:

Deductive system

A **deductive system** (or, sometimes, **inference system**) is specified by

1. A collection of allowed **judgements**, and
2. A collection of *steps*, each of which has a (typically finite) list of judgments as hypotheses and a single judgment as conclusion. A step is usually written as

$$\frac{J_1 \qquad \dots \qquad J_n}{J}$$

If n is 0, the step is often called an **axiom**.

2.1. Deductive system

The steps of a deductive system are generated by **inference rules**, which are schematic ways of describing collections of steps, generally involving metavariables.

For example, consider the following inference rule, describing a simple congruence:

$$\frac{x = y}{S(x) = S(y)}\,\text{CONG}$$

Here, x and y are variables. Substituting particular terms for them will yield a *step* which is an instance of this inference rule.

2.1. Deductive system

In type theory, every element *by its very nature* is an element of some type.
Introducing a variable, therefore, requires specifying its type.

We will be using the following judgements:

- A type — A is a (well-formed) type.
- Γ cx — Γ is a (well-formed) context.
- $a : A$ — a is an object of type A.
- $a \equiv b : A$ — a and b are definitionally equal objects of type A.

Not every presentation of type theory will brandish the same set of judgements.
In particular, the canonical version of homotopy type theory as presented by *the* HoTT book, only includes 2 judgements (the third and fourth)!

2.2. What's in a type?

Types may *look like* sets, but they are not!

In type theory, a type:

- is a first-class mathematical object
- types may or may not have **inhabitants** (values of that type)
- types are not iterable, quantifiable over or have cardinality

We'll formalize how to define a set of **base types**,
and how we **construct** types from other types,
forming a simple process to create types by **induction**.

2.2. What's in a type?

There is a general pattern for introduction of a new kind of type. We specify:

- **formation rules** for the type: for example, we can form the function type $A \to B$ when A is a type and when B is a type.
- **constructors** for its elements: for example, a function type has one constructor, λ-abstraction.
- **eliminators**: How to use the type's elements, for example function application.
- a **computation rule**[1], which expresses how an eliminator acts on a constructor.

[1] also referred to as β-reduction or β-equivalence

2.2. What's in a type?

- an optional **uniqueness principle**[1], which expresses uniqueness of maps into or out of that type.

[1] also referred to as η-expansion. When the uniqueness principle is not taken as a rule of judgemental equality, it is often nevertheless provable as a *propositional* equality.

2.2. What's in a type?

Type

A **type** is a first-class mathematical object defined by a set of **formation rules**, **constructors**, **eliminators**, **computation rules** and an optional **uniqueness principle**.

A type[1] is nothing more and nothing less than the behaviour as described by its rules.

[1] In type theory. Types in STLC, for example, behave differently.

3.1. Terms in STLC

The theory of functional programming is built on extensions of a core language known as the *simply-typed lambda calculus* (henceforth abbreviated STLC).

STLC is composed of two kinds of *sorts*: **types** and **terms**.

We can define types in this calculus as expressions generated by the grammar

$$\text{Types } A, B := \boldsymbol{b} \mid A \times B \mid A \to B.$$

Note the included $\boldsymbol{b}$ type: without it the grammar would have no terminal symbols.

Equivalently, the A type judgement may be derived from a number of inference rules corresponding to the production rules of Types:

$$\frac{}{b \text{ type}} \qquad \frac{A \text{ type} \qquad B \text{ type}}{A \times B \text{ type}} \qquad \frac{A \text{ type} \qquad B \text{ type}}{A \to B \text{ type}}$$

See how each rule derives **a new judgement** from a number of other judgements.

Remember that an inference rule is a form of a generic step, with metavariables we must fill in.

By combining these rules into a tree of steps whose leaves have no premises, we can produce *derivations* of judgements.

The tree below is a proof that $(b \times b) \to b$ is a type:

$$\dfrac{\dfrac{\dfrac{}{b\ \mathsf{type}} \qquad \dfrac{}{b\ \mathsf{type}}}{b \times b\ \mathsf{type}} \qquad \dfrac{}{b\ \mathsf{type}}}{(b \times b) \to b\ \mathsf{type}}$$

It is imperative to remember and learn to identify the metavariables in inference rules, as we'll be using them very often.

Terms are however, considerably more difficult to define due to two issues:

- There are infinitely many sorts of terms (one for each type)
- **The body $\underline{b}$ of a function $\lambda x.\underline{b} : A \to B$ is a term of type B of our grammar *extended* by a new constant $x : A$!**

To account for these extensions, we will introduce the concept of a **context** first.

3.1.1. Context

Context

The judgement $\vdash \Gamma\ \mathsf{cx}$ ("Γ is a **context**") expresses that Γ is a list of pairs of term variables (x, y, ...) with types (A, B, ...).

We write $\mathbf{1}$ for the empty context and $\Gamma, x : A$ for the extension of Γ by a variable x with type A.

The $\vdash \Gamma\ \mathsf{cx}$ judgement may be derived from the following inference rules:

$$\frac{}{\vdash \mathbf{1}\ \mathsf{cx}} \qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad A\ \mathsf{type}}{\vdash \Gamma, x : A\ \mathsf{cx}}\ \textsc{cx-ext}$$

A context Γ expresses the "parameter space" types have access to, enabling us to specify the variables that a dependent type depends on.

Now, defining terms becomes relatively straightforward, encompassing a number of rules:

$$\frac{(x : A) \in \Gamma \qquad \Gamma \text{ cx}}{\Gamma \vdash x : A} \qquad \frac{\text{a base term } \mathsf{c} \qquad \Gamma \text{ cx}}{\Gamma \vdash \mathsf{c} : \boldsymbol{b}}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash (a, b) : A \times B} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathbf{fst}(p) : A} \qquad \frac{\Gamma \vdash p : A \times B}{\Gamma \vdash \mathbf{snd}(p) : B}$$

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash f : A \to B}{\Gamma \vdash f\ a : B}$$

$\Gamma \vdash x : A$ is read "x has type A in context Γ."

Note that these contain the construction and elimination rules for all types in STLC.

Finally, we must not forget the computation rules, which introduce a notion of sameness to our theory:

$$\frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathbf{fst}((a, b)) \equiv a : A} \qquad \frac{\Gamma \vdash a : A \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathbf{snd}((a, b)) \equiv b : B}$$

$$\frac{\Gamma \vdash p : A \times B}{\Gamma \vdash p \equiv (\mathbf{fst}(p), \mathbf{snd}(p)) : A \times B} \qquad \frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b) \equiv b[a/x] : B}$$

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \equiv \lambda x.(f\ x) : A \to B}$$

The substitution operator $\Phi[x/y]$ means replacing every occurrence of y with x in Φ. For a definition, refer to the book.

4.1. Types and contexts

Just as we did with the judgement for terms, we will continue by extending the type judgement using contexts:

A type becomes $\Gamma \vdash A$ type read "A is a type in context Γ."

This has many downstream implications: we must now take into account *in what context* a type is well-formed. A handful of rules must be updated, starting with the cx judgement:

$$\frac{\vdash \Gamma \text{ cx} \qquad \Gamma \vdash A \text{ type}}{\vdash \Gamma, x : A \text{ cx}}\,\textsc{cx-ext},$$

For the rest, you are invited to update those in your head.

4.1. Types and contexts

Armed with term variable context added to the type judgment, we can now explain when $(x : A) \to B$ is a well-formed type:

$$\frac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma, x : A \vdash B \ \mathsf{type}}{\Gamma \vdash (x : A) \to B \ \mathsf{type}}$$

This is an example of a **formation rule**.

4.2. Type and term dependency

We can categorize type systems into three ‘tiers’ of dependency:

- It may be **not dependent**, disallowing a type to be parameterized by another type. For example, C’s type system is not dependent.
- It may have **uniform dependency**: types and terms may be parameterized by *other types*. Generic types in Java are an example of uniformly parameterized types.
- Or, ultimately, a type system exhibits **full-spectrum dependency** when types are indexed by terms.

4.2.1. Full-spectrum dependency

To achieve full-spectrum dependency, a type system must allow types to depend on other types and values, such as the following type family indexed by Nat:

```
nary : Set → Nat → Set
nary A 0 = A
nary A (suc n) = A → nary A n
```

`nary` takes a type A, some number n : Nat and produces a function of the shape:

$$\underbrace{A \to \ldots}_{n \text{ times}} \to A$$

4.2. Type and term dependency

Evidently, the *structure* of the resulting type is based on the input n.
This is a feat only a type system with full-spectrum dependency supports.

4.3. Substitution calculus

Let us turn our attention to what is called the **variable rule**:

$$\frac{}{\Gamma, x : A \vdash x : A}\,\text{VAR}$$

Although this looks sensible, this rule's assumptions do not hold:

- Assumption 1: $\vdash (\Gamma, x : A)\ \mathsf{cx}$
  - We could add $\vdash \Gamma\ \mathsf{cx}$ and $\Gamma \vdash A\ \mathsf{type}$ to the premise for this to hold, but ..
- Assumption 2: $\Gamma, x : A \vdash A\ \mathsf{type}$
  - .. even then, $\Gamma \vdash A\ \mathsf{type}$ does not imply $\Gamma, x : A \vdash A\ \mathsf{type}$!
    This would require us to prove a *weakening lemma* for types, which is not trivial.

[1] $\Gamma, x : A \vdash B\ \mathsf{type}$ when $\Gamma \vdash B\ \mathsf{type}$, but this introduces ambiguity into our rules.

4.3. Substitution calculus

Instead of that approach, or adding a silent weakening rule[1], we will opt to introduce **explicit weakening**.

4.3. Substitution calculus

The **explicit weakening rule** asserts the existence of an operation sending types and terms in context Γ to types and terms in context $\Gamma, x : A$, both written $-[\mathbf{p}]$[1].

$$\frac{\Gamma \vdash B \ \mathsf{type} \qquad \Gamma \vdash A \ \mathsf{type}}{\Gamma, x : A \vdash B[\mathbf{p}] \ \mathsf{type}} \qquad \frac{\Gamma \vdash b : B \qquad \Gamma \vdash A \ \mathsf{type}}{\Gamma, x : A \vdash b[\mathbf{p}] : B[\mathbf{p}]}$$

Using these rules, we can fix the variable rule from before:

$$\frac{\vdash \Gamma \ \mathsf{cx} \qquad \Gamma \vdash A \ \mathsf{type}}{\Gamma, x : A \vdash x : A[\mathbf{p}]}\,\textsc{var}$$

Bringing us to a safe and unambiguous variable rule.

[1] The p stands for projection. We'll see why later.

4.3. Substitution calculus

To use variables that occur earlier in the context, we can apply weakening repeatedly until they are the last variable.

Suppose we want to use x in the context $(x : A, y : B)$:

We know that $x : A \vdash x : A[\mathbf{p}]$ *Apply variable rule*

And so $x : A, y : B \vdash x[\mathbf{p}] : A[\mathbf{p}][\mathbf{p}].$ *Apply weakening*

In general, we can derive the following principle:

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, x : A \vdash B_1\ \mathsf{type} \qquad \dots \qquad \Gamma, x : A, y_1 : B_1, \dots \vdash B_n\ \mathsf{type}}{\Gamma, x : A, y_1 : B_1, \dots, y_n : B_n \vdash x \underbrace{[\mathbf{p}]\dots[\mathbf{p}]}_{n\text{ times}} : A \underbrace{[\mathbf{p}]\dots[\mathbf{p}]}_{n+1\text{ times}}}$$

4.3. Substitution calculus

$$\frac{\Gamma \vdash A \text{ type} \qquad \Gamma, x : A \vdash B_1 \text{ type} \qquad \dots \qquad \Gamma, x : A, y_1 : B_1, \dots \vdash B_n \text{ type}}{\Gamma, x : A, y_1 : B_1, \dots, y_n : B_n \vdash x \underbrace{[\mathbf{p}] \dots [\mathbf{p}]}_{n \text{ times}} : A \underbrace{[\mathbf{p}] \dots [\mathbf{p}]}_{n+1 \text{ times}}}$$

As "happy accident" of this approach we find is that the term $x[\mathbf{p}]^n$ encodes in two ways which variable it refers to:

- by the name x,
- but also by the number of weakenings n (called the variable's **de Bruijn index**),

so we might as well drop variable names altogether!

From now on, we can present contexts as lists of types $A.B.C$ with no variable names, and adopt a single notation for "the last variable in the

context".[1]

[1] Freeing us from the torment of explaining variable binding.

Simultaneous substitution

A **simultaneous substitution** (henceforth, just **substitution**) is an arbitrary composition of

- zero or more **weakenings**
- zero or more **term substitutions**

A substitution can turn any context Γ into any other context Δ.

Rather than axiomatizing *single* term substitutions and weakenings, we will axiomatize the concept of a simultaneous substitution instead.

4.3. Substitution calculus

For this, we will add one final basic judgement to our theory:

$$\Delta \vdash \gamma : \Gamma \qquad \text{“}\gamma \text{ is a substitution from } \Delta \text{ to } \Gamma\text{”}$$

corresponding to operations that send types/terms from context Γ to context Δ[1].

Notation We write

- Cx for the set of contexts,
- $\mathsf{Sb}(\Delta, \Gamma)$ for the set of substitutions from Δ to Γ,
- $\mathsf{Ty}(\Gamma)$ for the set of types in context Γ,
- and $\mathsf{Tm}(\Gamma, A)$ for the set of terms of type A in context Γ.

[1] If this notation seems backward to you, you must learn to live with it.

Now we *finally* have everything we need to discuss the dependent type theory as described by Martin-Löf:

The rules for contexts are as previous, but without variable names:

$$\frac{}{\vdash \mathbf{1}\ \mathsf{cx}} \qquad \frac{\vdash \Gamma\ \mathsf{cx} \qquad \Gamma \vdash A\ \mathsf{type}}{\vdash \Gamma.A\ \mathsf{cx}}\,\textsc{cx-ext}$$

The purpose of a substitution $\Delta \vdash \gamma : \Gamma$ is to shift types and terms from context Γ to context Δ:

$$\frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash A\ \mathsf{type}}{\Delta \vdash A[\gamma]\ \mathsf{type}} \qquad \frac{\Delta \vdash \gamma : \Gamma \qquad \Gamma \vdash \alpha : A}{\Delta \vdash \alpha[\gamma] : A[\gamma]}$$

The simplest interesting substitution is weakening, written $\mathbf{p}$:

$$\frac{\Gamma \vdash A\ \mathsf{type}}{\Gamma.A \vdash \mathbf{p} : \Gamma}$$

In concert with the substitution rules and $\mathbf{p}$, we can recover the weakening rule seen previously. Further, we have rules that close substitutions under nullary and binary composition:

$$\frac{\vdash \Gamma\ \mathsf{cx}}{\Gamma \vdash \mathrm{id} : \Gamma} \qquad \frac{\Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_2 \vdash \gamma_0 \circ \gamma_1 : \Gamma_0}$$

And these operations are unital and associative, as one might expect:

$$\frac{\Delta \vdash \gamma : \Gamma}{\Delta \vdash \gamma \circ \mathrm{id} = \mathrm{id} \circ \gamma = \gamma : \Gamma} \qquad \frac{\Gamma_3 \vdash \gamma_2 : \Gamma_2 \qquad \Gamma_2 \vdash \gamma_1 : \Gamma_1 \qquad \Gamma_1 \vdash \gamma_0 : \Gamma_0}{\Gamma_3 \vdash \gamma_0 \circ (\gamma_1 \circ \gamma_2) = (\gamma_0 \circ \gamma_1) \circ \gamma_2 : \Gamma_0}\ {}^{1}$$

TODO: put a[id]=a and substitution of composition is composition of substitution here?

[1] Yes, composition is reversed just like the $\Delta \vdash \gamma : \Gamma$ syntax.

As we've done away with variable names, we update the variable rule as follows:

$$\frac{\Gamma \vdash A \ \mathsf{type}}{\Gamma.A \vdash \mathbf{q} : A[\mathbf{p}]}\text{VAR}$$

We will use $\mathbf{q}$ to unambiguously refer to the last variable in the context. A variable in our system is a term of the form $\mathbf{q}[\mathbf{p}^n]$, where n is its de Bruijn index.

TODO: terminal substitutions and substitution extension