Type Theory Crash Course based on Thorsten Altenkirch's notes

lectures/slides by William DeMeo
<williamdemeo@gmail.com>

UH MFC Bootcamp, 29–31 March 2017

Part 0: What is Type Theory?

References

The material we cover here is based on the following:

- Altenkirch (2016) Naive Type Theory short course
  www.cs.nott.ac.uk/~psztxa/ntt/
- Capretta (2002) Abstraction and Computation, PhD thesis
  www.cs.nott.ac.uk/~vxc/publications/Abstraction_Computation.pdf
- Harper (2013) CMU course on HoTT
  www.cs.cmu.edu/~rwh/courses/hott/
- Pfenning (2009) Lecture notes on natural deduction
  www.cs.cmu.edu/~fp/courses/15317-f09/lectures/02-natded.pdf

Two Iterpretations

- **Type Theory** (TT) (with caps) is an alternative foundation for Mathematics—an alternative to Set Theory (ST)
- pioneered by Swedish mathematician **Per Martin-Löf**
- **type theory** (tt) (w/out caps) is the theory of types in programming languages
- TT and tt are related but different subjects

Type Theory: the basic idea

- organize mathematical objects into **Types** instead of **Sets**
  eg, the **Type** $\mathbb{N}$ of natural numbers, the **Type** $\mathbb{R}$ of reals, etc
- to say that π is real, write $\pi : \mathbb{R}$
- *Wait a minute!* Type Theory is merely Set Theory with the word Set replaced by Type and the symbol $\in$ replaced by $:$??

  WTF?!
- Of course not. In Type Theory we can only make objects of a certain type—*the type comes first*—and then we can construct elements of that type.
- In Set Theory all objects are there already and we can organize them into different sets; we might have an object x and ask wether this object is a **nat** ($x \in \mathbb{N}$) or a **real** ($x \in \mathbb{R}$).

Type Theory vs. Set Theory

- In Type Theory we think of $x : \mathbb{N}$ as meaning that x is a natural number "by birth" and we can ask whether x is a real number.
- We say $x : \mathbb{N}$ is a **judgement** while $x \in \mathbb{N}$ is a **proposition**
- We will revisit these ideas again and again, and they will become clearer once we gain some experience with Type Theory.

End of Part 0

Part 1: Constructive Math

- What is a good language for writing proofs?
- What kind of math do we want to do?
- In principle all math can be formalized in ZFC.
- Usually a much weaker theory is sufficient
  (PA suffices for much of Number Theory)
  (Analysis can be formalized in PA2)
- In fact, we don’t need to commit, as long as our proofs use standard techniques that we believe are formalizable in *some* system.

- But to do math on a computer
  we must make a choice!
- A computer program must be based on *some* formal system
- **ZFC** is not the obvious choice
- **constructive type theory**
  can be justified on both philosophical and practical grounds

Question: Why do math on a computer?

- Because computers can check whether proofs are correct? No, the peer review process works.
- Because computers can prove many things humans can't? No, at least not anytime soon.
- Because computers are really good at computing? Yes!!

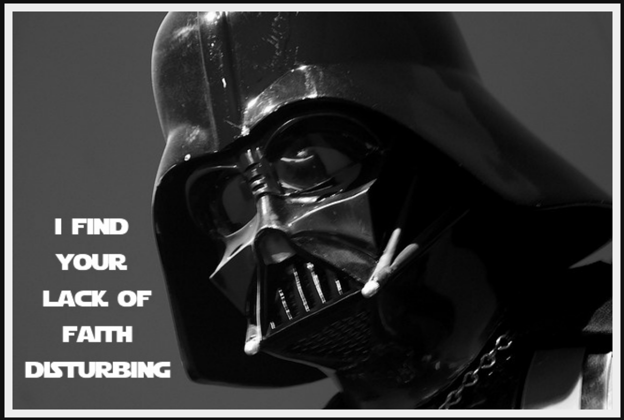
Nobody would question the utility of computer programs on the grounds that we can write those programs on a piece of paper faster and more easily in pseudo-code. This would be silly, since *programs written on paper cannot be executed*

The objection that formalizing math on computer is pointless because we can more easily write it down on a piece of paper can be disputed on similar grounds. But…

proofs of math theorems cannot be executed

…or can they?

- *Classical* proofs cannot always be executed, but *constructive* proofs can, in a sense.
- Constructive proofs give algorithms to compute all objects claimed to exist and decide all properties claimed decidable.
- It may seem strange to think of a proof as a program, even stranger that there can be different proofs of the same result that differ in "efficiency."



A Change of Tack

- Instead of discussing ways to formalize math, let’s consider ways to extend programming languages, e.g. richer data types, new paradigms/techniques.
- We will consider a high level functional language and see how it makes programming easier; some classical algorithms become easy or obvious; previously inconceivable programs are possible.
- We don’t mention logic and math at first.

Curry-Howard Correspondence

Eventually, we see *programs as proofs* of theorems and **constructive math** as a subsystem of the programming language.

The most important advantage:
programs are guaranteed correct
by virtue of the their inherent logical content!

End of Part 1

(time for a break)

Part 2: Type Theory vs Set Theory

Sets vs Types

- In Set Theory, $3 \in \mathbb{N}$ means
  “3 is an element of the set of natural numbers”
- In Type Theory, $3 : \mathbb{N}$ means
  “3 is an element of the type of natural numbers”
- Seems trivial… but here’s the significance…

Sets vs Types

- While $3 \in \mathbb{N}$ is a proposition, $3 : \mathbb{N}$ is a *judgment*; ie a piece of static information.
- In Type Theory every object and every expression has a (unique) type which is statically determined.
- Hence it doesn't make sense to use $a : A$ as a proposition.
- In Set Theory we define $P \subseteq Q$ as $\forall x.\, x \in P \to x \in Q$.
  This doesn't work in Type Theory since $x \in P$ is not a proposition.
- Set theoretic operations like $\cup$ or $\cap$ are not operations on types
  …but they can be defined as operations on predicates (subsets) of a given type. $\subseteq$ can be defined as a predicate on such subsets.

- Type Theory is **extensional** in the sense that we can't talk about details of encodings.
- In Set Theory we can ask whether $\mathbb{N} \cap \mathsf{Bool} = \emptyset$ Or whether $2 \in 3$. The answer to these questions depends on the choice of representation of the objects and sets involved.
- In addition to the judgment $a : A$, we introduce the judgment $a \equiv_A b$ which means a and b are **definitionally equal**.
- Definitional equality is a *static* property, hence it doesn't make sense as a proposition. (Later we introduce **propositional equality** $a =_A b$ which can be used in propositions)
- We write definitions using $:\equiv$, eg $n :\equiv 3$ defines $n : \mathbb{N}$ to be 3
- Type Theory is more restrictive than Set Theory… but this has some benefits…

Univalence Axiom

Since we can't talk about intensional aspects (implementation details), we can identify objects which have the same extensional behavior. This is reflected in the univalence axiom, which identifies **extensionally equivalent** types.

Truth Vs. Evidence

Another important difference between Set Theory and Type Theory is the way propositions are treated: Set Theory is formulated using predicate logic which relies on the notion of **truth**. Type Theory is self-contained and doesn’t refer to **truth**, but rather **evidence**.

Curry-Howard Correspondence

Using the propositions-as-types translation we can assign to any proposition P the type of its evidence $[[P]]$ as follows:

$$
\begin{aligned}
[[P \Rightarrow Q]] &\equiv [[P]] \rightarrow [[Q]] \\
[[P \wedge Q]] &\equiv [[P]] \times [[Q]] \\
[[\mathsf{True}]] &\equiv 1 \\
[[P \vee Q]] &\equiv [[P]] + [[Q]] \\
[[\mathsf{False}]] &\equiv 0 \\
[[\forall x : A.\, P]] &\equiv \Pi x : A.\, [[P]] \\
[[\exists x : A.\, P]] &\equiv \Sigma x : A.\, [[P]]
\end{aligned}
$$

0 is the empty type, 1 is the type with exactly one element

disjoint union +, product ×, and → (function) types are familiar

Π and Σ may be less familiar; we look at them later.

End of Part 2

Part 3: Non-dependent types

Universes

- To get started we have to say what a *type* is. We could introduce another judgement, but instead we'll use **universes.**
- A **universe** is a type of types. For example, to say that $\mathbb{N}$ is a type, we write $\mathbb{N} : \mathsf{Type}$, where Type is a universe.
- But what is the type of Type? Do we have $\mathsf{Type} : \mathsf{Type}$?
- This doesn't work in Set Theory due to **Russell's paradox** (consider the set of all sets that don't contain themselves)
- In Type Theory $a : A$ is not a Prop, hence it's not immediately clear wether the paradox still occurs.

- It turns out that a Type Theory with $\mathsf{Type} : \mathsf{Type}$ does exhibit **Russell's paradox**.
- Construct the tree $T : \mathsf{Tree}$ of all trees that don't have themselves as immediate subtrees. Then T is a subtree of itself iff it isn't.
- To avoid this, we introduce a hierarchy of universes
$$\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \cdots$$
and we decree that any $A : \mathsf{Type}_i$ can be *lifted* to $A^+ : \mathsf{Type}_{i+1}$
- Being explicit about universe levels can be quite annoying.
In notation we ignore the levels, but take care to avoid using universes in a cyclic way.
- That is we write Type as a metavariable for Type_i and assume that all levels act the same unless stated otherwise.

Functions

- In Set Theory **function** is a derived concept (a subset of the cartesian product with certain properties)
- In Type Theory **function** is a primitive concept.
- The basic idea is the same as in functional programming: a function is a **black box**; you feed it elements from its domain and out come elements of its codomain.
- Hence given $A, B : \mathsf{Type}$ we introduce the type of functions
$$A \to B : \mathsf{Type}$$
- We can define a function $f : \mathbb{N} \to \mathbb{N}$ explicitly, eg, $f(x) :\equiv x + 3$.
- We can now apply, $f(2) : \mathbb{N}$, and evaluate this application by replacing all x's in the body with 2; hence $f(2) \equiv 2 + 3$
- If we know how to calculate $2 + 3$ we can conclude $f(2) \equiv 5$

A word about syntax

- In Type Theory, as in functional programming, we usually try to save parentheses and write $fx :\equiv x + 3$ and $f2$
- The explicit definition of a function requires a name but we want **anonymous functions** as well—this is the justification for the λ-notation
- We write $\lambda x.\, x + 3 : \mathbb{N} \to \mathbb{N}$ to avoid naming the function.
- We can apply: $(\lambda x.\, x + 3)(2)$
- The equivalence $(\lambda x.\, x + 3)(2) \equiv 2 + 3$ is called β-reduction
- The explicit definition $fx \equiv x + 3$ can now be understood as a shorthand for $f \equiv \lambda x.\, x + 3$.

Products and sums

- Given $A, B : \mathsf{Type}$ we can form
  - their product $A \times B : \mathsf{Type}$
  - their sum $A + B : \mathsf{Type}$
- The elements of a product are tuples, that is $(a, b) : A \times B$ if $a : A$ and $b : B$
- The elements of a sum are injections, that is $\mathsf{inl}\ a : A + B$ if $a : A$ and $\mathsf{inr}\ b : A + B$, if $b : B$

- To define a function from a product or a sum it suffices to say what the function returns for the constructors (tuples for products; injections for sums)

- As an example we derive the tautology
$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R)$$
using the propositions as types translation.

- Assuming $P, Q, R : \mathsf{Type}$, we must construct an element of the following type
$$\begin{aligned}((P \times (Q + R) &\to (P \times Q) + (P \times R)) \\ \times((P \times Q) + (P \times R) &\to P \times (Q + R))\end{aligned}$$

Solution

Define $f : P \times (Q + R) \to (P \times Q) + (P \times R)$ as follows:

$$f(p, \mathsf{inl}\ q) :\equiv \mathsf{inl}\ (p, q)$$

$$f(p, \mathsf{inr}\ r) :\equiv \mathsf{inr}\ (p, r)$$

Define $g : (P \times Q) + (P \times R) \to P \times (Q + R)$ as follows:

$$g(\mathsf{inl}\ (p, q)) :\equiv (p, l\mathsf{inl}\ q)$$

$$g(\mathsf{inr}(p, r))) :\equiv (p, \mathsf{inr}\ r)$$

The tuple (f, g) is an element of the desired type!

Exercise 1

Using the propositions as types translation, try to prove the following tautologies (where P, Q, R : Type are propositions represented as types)

1. $(P \wedge Q \Rightarrow R) \Leftrightarrow (P \Rightarrow Q \Rightarrow R)$
2. $((P \vee Q) \Rightarrow R) \Leftrightarrow (P \Rightarrow R) \wedge (Q \Rightarrow R)$
3. $\neg(P \vee Q) \Leftrightarrow \neg P \wedge \neg Q$
4. $\neg(P \wedge Q) \Leftrightarrow \neg P \vee \neg Q$
5. $\neg(P \Leftrightarrow \neg P)$

Exercise 2

Law of Excluded Middle $(\forall P)(P \vee \neg P)$ is not provable in TT

However, we can prove its double negation (ie “LEM is not refutable”)

Using the **propositions-as-types** translation, prove

$$(\forall P)\neg\neg(P \vee \neg P)$$

If for a particular proposition P we can establish $P \vee \neg P$ then we can also derive the principle of indirect proof $\neg\neg P \Rightarrow P$ for the same proposition.

Show $(P \vee \neg P) \Rightarrow (\neg\neg P \Rightarrow P)$

The converse does not hold **locally** (Counterexample?)

…but it holds **globally**. Show that the two principles are equivalent. That is, prove:

$$(\forall P)(P \vee \neg P) \Longrightarrow (\forall P)(\neg\neg P \Rightarrow P)$$

Functions out of products and sums can be reduced to using a fixed set of **combinators** called *non-dependent eliminators* or *recursors* (even though there is no recursion going on).

$$R^{\times} : (A \to B \to C) \to A \times B \to C$$

$$R^{\times} f\ (a, b) :\equiv f a b$$

$$R^{+} : (A \to C) \to (B \to C) \to A + B \to C$$

$$R^{+} f g\ (\mathsf{inl}\ a) :\equiv f a$$

$$R^{+} f g\ (\mathsf{inr}\ b) :\equiv g b$$

- The **recursor** $R^{\times}$ for products maps a **curried** function $f : A \to B \to C$ to its **uncurried** form, taking tuples as arguments.
- The **recursor** R^{+} basically implements the case function performing case analysis over elements of $A + B$.

Exercise 3

Show that using the **recursor** $R^{\times}$ we can define the projections:

fst : A × B → A

fst (a, b) :≡ a

snd : A × B → B

snd (a, b) :≡ b

Vice versa: can the recursor be defined using only the projections?

The unit and empty types

- Denote by $\mathbf{1}$ the empty product, called the *unit type*
- Denote by $\mathbf{0}$ the empty sum, called the *empty type*
- $() : \mathbf{1}$ is the only inhabitant of the unit type
- Nothing inhabits $\mathbf{0}$ (it's the *empty* type!)
- We introduce the corresponding recursors:
  $R^{\mathbf{1}} : C \to (\mathbf{1} \to C)$ is defined by $R^{\mathbf{1}} c() :\equiv c$
  $R^{\mathbf{0}} : \mathbf{0} \to C$ (no defining eqn since it won't be applied)
- The recursor for $\mathbf{1}$ is pretty useless. It just defines a constant function.
- The recursor for the empty type implements the logical principle *ex falso quod libet*

Exercise 4

Construct solutions to exercises 1 and 2 using only the eliminators.

- The use of arithmetical symbols for operators on types is justified because they act like the corresponding operations on finite types.
- Let us identify the number n with a **type** inhabited by the following elements: $0_n, 1_n, \ldots, (n-1)_n \; : \; \underline{n}$
- Then we observe that

$$
\begin{aligned}
\underline{0} &= 0 \\
\underline{m+n} &= \underline{m} + \underline{n} \\
\underline{1} &= 1 \\
\underline{m \times n} &= \underline{m} \times \underline{n}
\end{aligned}
$$

- Read $=$ here as “has the same number of elements”
  This use of equality will be justified later when we introduce the **univalence principle**

Function Types are Exponentials

The arithmetic interpretation of types also extends to the function type, which corresponds to exponentiation. Indeed, in Mathematics the function type $A \to B$ is often written as B^A, and indeed we have: $\underline{m^n} = \underline{n} \to \underline{m}$.

End of Part 3

Part 4: Dependent Types

What are Dependent Types?

You may be familiar with polymorphic types (aka generics)
These are types that are indexed by other types

Example

```
Array<Integer>      // Java

List[(String, Int)] // Scala
```

A *dependent type* is indexed by an *element* of another type

Examples

A^n : Type, the *type of n-tuples* whose inhabitants are $(a_0, a_1, \ldots, a_{n-1}) : A^n$ where $a_i : A$

$\underline{n}$: Type, the *finite type* whose inhabitants are $0_n, 1_n, \ldots, (n-1)_n : \underline{n}$

What are Dependent Types?

The $\boldsymbol{n}$**-tuple type** A^n : Type is indexed by parameters

$$\underline{n} : \mathsf{Type} \qquad \text{and} \qquad A : \mathsf{Type}$$

In general, a *dependent type* is obtained by applying a function with codomain Type.

Example 1

$$\mathsf{Vec} : \mathsf{Type} \to \mathbb{N} \to \mathsf{Type}$$

$$\mathsf{Vec}\ A\ n :\equiv A^n$$

Example 2

$$\mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$$

$$\mathsf{Fin}\ n :\equiv \underline{n}$$

Curry-Howard again

In the *propositions-as-types* view, **dependent types** are used to encode predicates.

Example

$\mathsf{Prime} : \mathbb{N} \to \mathsf{Type}$

This takes $n : \mathbb{N}$ as input and outputs $\mathsf{Prime}\ n : \mathsf{Type}$, the type representing *evidence that n is a prime number*.

If φ is a proof that $n : \mathbb{N}$ is prime, then $\varphi : \mathsf{Prime}\ n$.

Of course $\mathsf{Prime}\ n$ could be uninhabited, eg $\mathsf{Prime}\ 4$.

Codifying Relations

By currying we can also use dependent types to represent **relations**

Example $\leq \, : \mathbb{N} \to \mathbb{N} \to \mathsf{Type}$
$m \leq n : \mathsf{Type}$, the type of *evidence that* m *is less or equal to* n

If φ is a proof of $m \leq n$, then $\varphi : m \leq n$

Example Let $\mathbf{A}$ be an algebra and $\theta \in \mathrm{Con}\,\mathbf{A}$ a congruence

$\theta : A \to A \to \mathsf{Type}$ takes inputs $a, b : A$ and outputs $a \; \theta \; b : \mathsf{Type}$
, the type of *evidence for* $(a, b) \in \theta$

If φ is a proof of $a \; \theta \; b$, then $\varphi : a \; \theta \; b$

Martin-Lof intensional type theory

- Intensional type theory is the brand of type theory used in systems like Agda and Coq
- NuPrl is based on extensional type theory
- This is an important distinction and it centers around different notions of **equality**

- In the original formulation by Martin-Lof, there is a judgement called *definitional equality*, which is asserted when two terms denote the same value.
- Today, this is most often replaced by a reduction relation. Two terms are called *convertible* when they can be reduced to a common decendant using the reduction rules. If we reduce a term as much as possible, we always obtain after a finite number of steps, a unique *normal form* (that cannot be simplified further). Convertible terms are interchangeable.
- *extensional* versions of type theory, like NuPrl, have a stronger notion of **definitional equality**
for example, two functions can be identified if their graphs are the same
- However, the price to pay is *undecidability of type checking*

Extensional Type Theory

Intensional Extensional

- ETT does not distinguish between **definitional equality** (computational) and **propositional equality** (requires proof)
- Type checking is **undecidable** in ETT
  programs in the theory might not terminate
- **Example** In ETT we can give a type to the **Y-combinator**
- This does not prevent ETT from being a basis for a practical tool, as NuPRL demonstrates.
- From a practical standpoint, there's no difference between a program which doesn't terminate and a program which takes a million years to terminate

Intensional Type Theory

Intensional Extensional

- ITT has decidable type checking, but the representation of standard math concepts can be more cumbersome.
- In ITT **extensional reasoning** requires using setoids or similar constructions.
- There are many common math objects that are hard to work with and/or can't be represented without this.
- **Examples:** Integers and rational numbers can be represented without setoids, but the representations are not easy to work with; Reals cannot be represented without setoids or something similar.

Homotopy type theory

- HoTT works on resolving these problems
- HoTT allows one to define higher inductive types that not only define **first-order constructors** (values or points), but **higher-order constructors** (equalities between elements–paths), equalities between equalities (homotopies), ad infinitum.