Predicate logic & dependent types

We have a complete correspondence between the simply-typed λ-calculus and the logic of propositions ($\Rightarrow, \vee, \wedge, \neg, \bot, \ldots$)

If we want to do mathematics though, propositional logic is a bit limited. We need predicates (propositions which are allowed to quantify over individuals using "for all" and "there exists")

In predicate logic / first order logic, we have two layers:
- the universe of objects (integers in Peano Arithmetic, sets in ZFC set theory) which has constants and functions
- the logical layer (which has all the connectives seen so far, plus $\forall, \exists, =$)

Example : $\forall x \, \forall y, (x \times y = 0) \Rightarrow (x = 0) \vee (y = 0)$

To account for this richer language, we extend our calculus from the STLC to dependent type theory (the foundation of Rocq, Lean, etc.)

We noted that the types of STLC could play the role of either datatypes or propositions. In dependent type theory, types will play both roles simultaneously

$\mathbb{N}$	type of integers
$\mathbb{N} \to \mathbb{N}$	type of integer functions
$x =_{\mathbb{N}} y$	type of proofs that x is equal to y
$x =_{\mathbb{N}} y \to y =_{\mathbb{N}} z$	type of proofs that $x = y$ implies $y = z$
$x =_{\mathbb{N}} y \to \mathbb{N}$	} mixing types-as-datatypes and types-as-propositions
$\mathbb{N} \to x =_{\mathbb{N}} y$	

And most importantly, types may depend on terms (hence the name of dependent type theory)

$x : \mathbb{N}, y : \mathbb{N} \vdash \underbrace{x =_{\mathbb{N}} y}_{\text{type}}$ — terms: x, y

$x : \mathbb{N} \vdash \underbrace{\text{Vector } x}_{\text{type of } x\text{-tuples of integers}}$ — term: x

Universes

Another way of understanding dependent types is via a universal type. We will write Type for the "type of types"

$$\mathbb{N} : \text{Type} \qquad \mathbb{N} \times \mathbb{N} : \text{Type} \qquad A : \text{Type}, B : \text{Type} \vdash A \to B : \text{Type}$$

Dependent types happen when we can write functions from a "regular" type ($\mathbb{N}$, $\mathbb{N} \to \mathbb{N}$, ...) into Type

$n : \mathbb{N} \vdash \text{Vector } n : \text{Type}$ and thus $\text{Vector} : \mathbb{N} \to \text{Type}$

Likewise, the equality of integers is a function from $\mathbb{N} \times \mathbb{N}$ to Type or, in curried form:

$$eq_{\mathbb{N}} : \mathbb{N} \to \mathbb{N} \to \text{Type}$$

we tend to write $x =_{\mathbb{N}} y$ instead of $eq_{\mathbb{N}}\ x\ y$

Size issues

Type is the type of types, yet we do not have Type : Type. Indeed, this would lead to a contradiction similar to Russell's paradox. Instead, Type is member of a larger universe of types called Type_1

Type is the type of "small" types ($\mathbb{N}$, $\mathbb{N} \to \mathbb{N}$, $\mathbb{N} \times \mathbb{N}$ ...)
Type_1 is the type of "1-large" types (Type, $\mathbb{N} \to$ Type, Type $\times$ Type ...)
Type_2 is the type of "2-large" types
etc.

Compare with: - V_α where α is an inaccessible cardinal
- Grothendieck universes in category theory

Quantifiers

How should quantifiers work in dependent type theory? Well, a proof of $\forall x \in \mathbb{N},\ x + x \geq x$ should tell us that for any given $x : \mathbb{N}$, we can find a proof of $x + x \geq x$

In DTT, if A is a type and $B(x)$ a type which depends on $x : A$, we can form the type $\Pi\,(x : A).\ B(x)$, called a dependent product type. Its elements are functions whose input type is A, but whose output type depends on the input

For instance, assume that $f : \Pi\,(n : \mathbb{N}).\ \text{Vector } n$
Given any $t : \mathbb{N}$, we can apply f to t and obtain $f\,t : \text{Vector } t$
$f\,3 : \text{Vector } 3$
$f\,(2 \times 2) : \text{Vector } 2 \times 2$
$f\,(x + y) : \text{Vector } (x + y)$

Thus the rules for dependent products are

this judgment means that B is a well-formed type assuming $\Gamma, x : A$

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Pi\,(x : A).\ B : \text{Type}}$$

the conclusion is that the type $\Pi\,(x : A).\ B$ is well-formed

rule for forming Π-types

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda\,(x : A).\ t : \Pi\,(x : A).\ B}$$

rule for constructing elements of Π-types

$$\frac{\Gamma \vdash t : \Pi\,(x : A).\ B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B[u/x]}$$

rule for using elements of Π-type

... along with the β-rule $(\lambda x.\ t)\,u \equiv t[u/x]$
and the η-rule $\lambda x.\ (t\,x) \equiv t$

Existential quantifier

A proof of $\exists\, x \in \mathbb{N},\ x + x \geq x$ should tell us how to construct some element $n \in \mathbb{N}$, and a proof that $n + n \geq n$ (for that specific n that we constructed)

In DTT, if A is a type and $B(x)$ a type which depends on $x : A$, we can form the type $\Sigma\,(x : A).\,B(x)$ called a dependent sum type. Its elements are pairs $\langle t ; u \rangle$ of some $t : A$ and some $u : B(t)$ (the type of the second member depends on the type of the first member)

For instance, consider the type $\Sigma\,(n : \mathbb{N}).\ \text{Vector}\ n$
Its elements are pairs of an integer n and a n-uplet
$\langle 3, [4,3,1] \rangle : \Sigma\,(n : \mathbb{N}).\ \text{Vector}\ n$

Rules for Σ-types

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma, x : A \vdash B : \text{Type}}{\Gamma \vdash \Sigma\,(x : A).\,B : \text{Type}}$$ rule for formation

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[t/x]}{\Gamma \vdash \langle t ; u \rangle : \Sigma\,(x : A).\,B}$$ rule for constructing elements of a Σ-type

$$\frac{\Gamma \vdash t : \Sigma\,(x : A).\,B}{\Gamma \vdash \pi_1\, t : A} \qquad \frac{\Gamma \vdash t : \Sigma\,(x : A).\,B}{\Gamma \vdash \pi_2\, t : B[\pi_1 t / x]}$$ rules for using elements of Σ-types

$\pi_1 \langle t ; u \rangle \equiv t$ $\qquad$ $\pi_2 \langle t ; u \rangle \equiv u$ $\qquad$ β rules

$\langle \pi_1 t ; \pi_2 t \rangle \equiv t$ $\qquad$ η rule

Nondependent vs dependent types

Observe that in case the type B does not depend on A,
- $\Pi\,(x : A).\,B$ has the same rules as $A \to B$
- $\Sigma\,(x : A).\,B$ has the same rules as $A \times B$

In dependent type theory, it is customary to define only Π and Σ, and to have products and arrows as special cases of those.

In Rocq and Lean, $A \to B$ is an abbreviation for $\Pi\,(x : A).\,B$

The natural numbers and induction

So far, our dependent type theory includes
- Type universes
- Π and Σ types (which may be used to simulate $\to$ and $\times$)

In the STLC, we also mentioned
- base datatypes such as $\mathbb{N}$ or Bool
- disjoint sums $A + B$
- the empty type $\bot$

These three types have somewhat similar properties, if you squint your eyes a bit. They fit in the broad family of inductive types. Let us see how natural numbers work.

Mathematically, there is a nice way to define the integers: it is the smallest set which contains 0 and which is closed under successor. In type theory, we define $\mathbb{N}$ as the inductive type generated by
- $0 : \mathbb{N}$
- $\mathrm{suc} : \mathbb{N} \to \mathbb{N}$

Thus we get two introduction rules:

$$\frac{\Gamma \text{ is well-formed}}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathrm{suc}(n) : \mathbb{N}}$$

we can use these rules to construct all natural numbers as $\mathrm{suc}(\mathrm{suc}(\mathrm{suc}(\mathrm{suc} \ldots (\mathrm{suc}(0)) \ldots)))$

The elimination rule(s) should tell us how to define a function out of the natural numbers to an arbitrary type X. Defining a function $f : \mathbb{N} \to X$ amounts to saying
- what $f(0)$ is
- what $f(\mathrm{suc}(n))$ is, knowing what $f(n)$ is.

For instance, the following should be a valid definition of the factorial function:

$$\begin{aligned} &\mathrm{fact} : \mathbb{N} \to \mathbb{N} \\ &\mathrm{fact}\ 0 \equiv 1 \\ &\mathrm{fact}\ (\mathrm{suc}\ n) \equiv (\mathrm{suc}\ n) \times (\mathrm{fact}\ n) \end{aligned}$$

To formalise this idea as a typing rule, we introduce the recursion operator $\mathbb{N}\text{-rec}$. If we want to build a function of type $\mathbb{N} \to X$, and we know

- $x_z : X$ — x_z is the image of 0
- $x_s : \mathbb{N} \to (X \to X)$ — x_s computes $f(n+1)$ if we give it the integer n and the result of $f(n)$

Then $\mathbb{N}\text{-rec}\,(X, x_z, x_s, n)$ computes $f(n)$ from that. Here is the corresponding typing rule:

$f(0)$ — how to compute $f(\text{suc } n)$ given $f(n)$ — an arbitrary number

$$\frac{X : \text{Type} \qquad x_z : X \qquad x_s : \mathbb{N} \to X \to X \qquad n : \mathbb{N}}{\mathbb{N}\text{-rec}\,(X, x_z, x_s, n) : X}$$

the result of $f(n)$

We also get some equations:

$$\mathbb{N}\text{-rec}\,(X, x_0, x_s, 0) \equiv x_0$$

$$\mathbb{N}\text{-rec}\,(X, x_0, x_s, \text{suc } n) \equiv x_s\ n\ (\mathbb{N}\text{-rec}\,(X, x_0, x_s, n))$$

Our earlier definition of factorial can be encoded with this operator as $\text{fact} \equiv \lambda n.\ \mathbb{N}\text{-rec}\,(\mathbb{N}, 1, \lambda n'.\lambda m.\ \text{suc}(n') \times m)$
The reader may easily check that the computation rules entail $\text{fact}(3) \equiv 1$, $\text{fact}(5) \equiv 120$ etc.

However, this rule only work to construct nondependent functions $\mathbb{N} \to X$. In dependent type theory, we also want to construct dependent functions of type $\Pi(n : \mathbb{N}).\ P$.
Thus, we upgrade our recursion operator to the induction operator:

property — true at 0 — preserved by suc — arbitrary element

$$\frac{P : \mathbb{N} \to \text{Type} \qquad p_0 : P\,0 \qquad p_s : \Pi(n : \mathbb{N}).\ P\,n \to P\,(\text{suc } n) \qquad n : \mathbb{N}}{\mathbb{N}\text{-elim}\,(P, p_0, p_s, n) : P\,n}$$

Remark that this is strikingly similar to Peano's induction on natural numbers! Remark also that $\mathbb{N}\text{-rec}$ is just the special case of $\mathbb{N}\text{-elim}$ when P does not use its argument.

We also get the same equations:

$\mathbb{N}\text{-elim}\ (X, x_0, x_S, 0) \equiv x_0$

$\mathbb{N}\text{-elim}\ (X, x_0, x_S, \text{suc}\ n) \equiv x_S\ n\ (\mathbb{N}\text{-rec}\ (X, x_0, x_S, n))$

Of course $\mathbb{N}$-rec and $\mathbb{N}$-elim are not very readable. In practice, we write these definitions using equations, e.g.

zeros : $\Pi(n : \mathbb{N})$. Vector n
zeros 0 ≡ []
zeros (n+1) ≡ 0 :: (zeros n)

Instead of

$\lambda n.\ \mathbb{N}\text{-elim}\ (\text{Vector}, [], \lambda n'.\ \lambda v.\ 0 :: v, n)$

Disjoint sums

We defined the natural numbers as the inductive type (smallest type) generated by $0 : \mathbb{N}$ and $\text{suc} : \mathbb{N} \to \mathbb{N}$. Likewise, we will define $A+B$ as the inductive type generated by $\text{in}_1 : A \to A+B$
$\text{in}_2 : B \to A+B$

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash B : \text{Type}}{\Gamma \vdash A+B : \text{Type}}$$ formation rule

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \text{in}_1(t) : A+B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \text{in}_2(t) : A+B}$$ introduction rules

$$\frac{\Gamma \vdash P : A+B \to \text{Type} \quad \Gamma, x : A \vdash t_1 : P(\text{in}_1 x) \quad \Gamma, x : B \vdash t_2 : P(\text{in}_2 x) \quad \Gamma \vdash u : A+B}{\Gamma \vdash \text{sum-elim}\ (P, t_1, t_2, u) : P\ u}$$ elimination rule

$\text{sum-elim}\ (P, t_1, t_2, \text{in}_1(a)) \equiv t_1[a/x]$
$\text{sum-elim}\ (P, t_1, t_2, \text{in}_2(b)) \equiv t_2[b/x]$ computation rules

Empty type

The empty type is the inductive type generated by nothing at all.

$$\frac{\Gamma \text{ is well-formed}}{\Gamma \vdash \bot : \text{Type}}$$ formation rule

no introduction rules

$$\frac{\Gamma \vdash P : \bot \to \text{Type} \qquad \Gamma \vdash t : \bot}{\Gamma \vdash \text{empty-elim}(P, t) : P\ t}$$ elimination rule

no computation rules

Equality

To formulate non-trivial mathematical statements, we should have equality in our language. In classical set theory equality is part of the very language of logic, it's not really a mathematical object. In type theory however, equality is just another type.

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t =_A u : \text{Type}}$$ formation rule

sometimes one writes $I(A, t, u)$ to emphasise that it is a type

As such, proofs of equality between t and u correspond to terms of type $t =_A u$. This means that proofs of equality are mathematical objects in their own right, and we can even talk about equalities between equality proofs!

In type theory, equality is also an inductive type, just like $\mathbb{N}$ or $A + B$: it is the smallest relation on A that is reflexive. As a consequence, we get a single introduction rule, corresponding to a proof of reflexivity

$$\frac{\Gamma \vdash A : \text{Type} \qquad \Gamma \vdash t : A}{\Gamma \vdash \text{refl}(t) : t =_A t}$$ introduction rule

As an inductive type, the elimination rule for equality should follow the same kind of pattern as the eliminator for $\mathbb{N}$

Assume some dependent type $P : A \to \text{Type}$, and assume that we have some term $t : P\,x$. Then, if we want to show $P\,y$ for any term $y : A$ with $e : x =_A y$, it suffices to show it when $y \equiv x$ and $e \equiv \text{refl}(x)$

$$\frac{x : A \quad P : A \to \text{Type} \quad t : P\,x \quad y : A \quad e : x =_A y}{\text{transp}(A, x, P, t, y, e) : P\,y}$$

elimination rule (transport)

$$\text{transp}(A, x, P, t, x, \text{refl}(x)) \equiv t$$

computation rule

This rule seems a bit daunting at first, but it is not too difficult to build an intuition about it : it says that if we know that $x =_A y$, then we can substitute x with y in any dependent type (we can go from $P(x)$ to $P(y)$)

This rule is already sufficient to prove that equality is symmetric and transitive. For transitivity, assume that we have $e_1 : x =_A y$ and $e_2 : y =_A z$.
Then by using transport along e_2, we can substitute y with z in any type. If we apply it to e_1, we get that
$\text{transp}(\lambda y'.\, x =_A y', y, e_1, z, e_2) : x =_A z$

Thus equality is an equivalence relation, which is reassuring.

We are almost done with the rules of dependent type theory, but we are still missing a detail. Equality proofs are mathematical objects in their own right, so we can have types that depend on equality proofs. To handle these, we should strengthen the elimination rule of equality a little bit to obtain the "J rule"

$$\frac{x : A \quad P : \Pi(y : A).\, x =_A y \to \text{Type} \quad t : P\,x\,\text{refl}(x) \quad y : A \quad e : x =_A y}{J(A, x, P, t, y, e) : P\,y\,e}$$

$$J(A, x, P, t, x, \text{refl}(x)) \equiv t$$

Concluding remarks

The dependent type theory that we presented here is very close to the theory implemented by Rocq. This type theory is neutral, in that it leaves many questions open:

- Is reasoning by contradiction allowed?

By default, dependent type theory is constructive, meaning that the only way to prove an existential statement is to supply a concrete witness. If we want to allow proofs by contradiction, we should postulate classical axioms such as the law of excluded middle

- Can equality proof have computational content?

In this minimal type theory, we cannot prove the principle of Uniqueness of Identity Proofs (UIP), which is formulated as $\Pi\,(A : \mathrm{Type}).\ \Pi\,(x\ y : A).\ \Pi\,(e_1\ e_2 : x =_A y).\ e_1 =_{x =_A y} e_2$
Even in constructive set theory, this principle goes without saying, but it may be interesting to work without, most notably in homotopy type theory

- Are functions extensional?

In plain type theory, we cannot even show the principle of function extensionality, which says that two functions are equal if and only if they send equal inputs to equal outputs. Models that negate function extensionality are quite rare, but may be useful if one is interested in algorithmic complexity.

→ Rocq stays neutral and lets the user postulate what they need
→ Lean brings type theory closer to set theory, by postulating classical logic, UIP and function extensionality
→ Homotopy type theory postulates univalence to allow for higher, homotopy-coherent mathematics.