Typed Lambda-calculus

Type theory started as a language for defining mathematical functions
For those who know : parallel with functional programming

In set theory, a function is technically a set of pairs, but in practice most functions are defined using expressions

example:
$f : \mathbb{N} \to \mathbb{N}$
$f : n \mapsto n \times n + 3$
and then $f(3)$ is obtained by subst n with 3

Type theory modifies this scheme in 2 ways :

- functions do not need to have a name
  anonymous functions are noted as $\lambda n . \underbrace{n \times n + 3}_{\text{body}}$ (binder: λn)
- the "typing" judgment is extended to any type built out of base types ($\mathbb{N}, \mathbb{R}, \ldots$), arrows ($A \to B$) and products ($A \times B$)

examples :
$f : \mathbb{N} \to \mathbb{N}$; $f \equiv \lambda n . n \times n + 3$
$n : \mathbb{N}$; $n \equiv f(4)$
$diag : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$; $diag \equiv \lambda n . \langle n, n \rangle$

$$comp : (\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$$
$$comp \equiv \underbrace{\lambda g}_{1} . (\underbrace{\lambda x}_{2} . \underbrace{\pi_1(g)}_{3} (\underbrace{\pi_2(g)}_{4} (\underbrace{x}_{5})))$$

1 – take $g : (\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \to \mathbb{N})$. The composition is defined as follows:
2 – take $x : \mathbb{N}$
3,4 – compute the components $\pi_1(g)$ and $\pi_2(g)$
5 – apply them to x in order

This λ-expression is a bit messy ! To make it somewhat more digestible, type theorists write parentheses only when necessary : $\lambda g . \lambda x . \pi_1 g (\pi_2 g x)$

We can also expand binders on cartesian products into components
→ $\lambda \langle g_1, g_2 \rangle . \lambda x . g_1 (g_2 x)$

Lastly, we may give type annotations in binders when useful
→ $\lambda \langle g_1, g_2 \rangle . \lambda (x : \mathbb{N}) . g_1 (g_2 x)$

Simple type theory

A type theory is a set of rules to produce such definitions. These rules should let us write any reasonable definition, and forbid nonsensical stuff such as

$$\begin{aligned} &\text{wrong} : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \\ &\text{wrong} \equiv \lambda f.\ f\ f \end{aligned}$$

← type mismatch : f expects an argument of type $\mathbb{N}$, but we are giving $f : \mathbb{N} \to \mathbb{N}$

In fact, there is no way to give a meaningful type to self application, which is why type theory was initially developped by Russell to avoid the liar paradox !

Contexts

Since mathematics often work with assumptions ("let x be an integer...") we generalise typing judgments to include a context of assumptions

For instance : $\underbrace{\underbrace{f : \mathbb{N} \to \mathbb{N}}_{\text{assumption 1}},\ \underbrace{x : \mathbb{N}}_{\text{assumption 2}}}_{\text{context}} \vdash \underbrace{x + 2 \times f(5) : \mathbb{N}}_{\text{judgment}}$ } sequent

"Let f be a function of type $\mathbb{N} \to \mathbb{N}$. Let x be an integer. Then the term $x + 2 \times f(5)$ has type $\mathbb{N}$"

Inference rules

The rules of a type theory define which sequents hold, and which sequents do not hold. They have the following general shape :

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$ } premises } inference rule

} conclusion

"If t has type $A \to B$ in context Γ and u has type A in context Γ, then $t\ u$ has type B in context Γ"

Here are all the inference rules of simple type theory

structural rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

rules for functions

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}$$

rules for products

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1(t) : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2(t) : B}$$

Rules for the base types

$$\frac{}{\Gamma \vdash 0 : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N}}{\Gamma \vdash \mathrm{suc}(t) : \mathbb{N}} \qquad \frac{\Gamma \vdash t : \mathbb{N} \quad \Gamma \vdash u : \mathbb{N}}{\Gamma \vdash t + u : \mathbb{N}}$$

We can use these rules to derive judgments by chaining them
It is often convenient to start from the conclusion & work backwards

derivation tree

$$\dfrac{\dfrac{\dfrac{\dfrac{}{n : \mathbb{N} \vdash n : \mathbb{N}} \quad \dfrac{}{n : \mathbb{N} \vdash n : \mathbb{N}}}{n : \mathbb{N} \vdash n + n : \mathbb{N}} \quad \dfrac{\dfrac{\dfrac{}{n : \mathbb{N} \vdash 0 : \mathbb{N}}}{n : \mathbb{N} \vdash 1 : \mathbb{N}}}{n : \mathbb{N} \vdash 2 : \mathbb{N}}}{n : \mathbb{N} \vdash (n + n) + 2 : \mathbb{N}}}{\cdot \vdash \lambda n.\,(n + n) + 2 : \mathbb{N} \to \mathbb{N}}$$

$1 \equiv \mathrm{suc}(0)$
$2 \equiv \mathrm{suc}(\mathrm{suc}(0))$

Since all the branches are closed, the conclusion is a valid sequent
Well-typed terms are inductively generated by the inference rules — they are an "initial algebra" for a "generalised algebraic theory"

Second example =

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{g : [\cdots], x : A \vdash g : (B \to C) \times (A \to B)}}{g : [\cdots], x : A \vdash \pi_1 g : B \to C} \quad \dfrac{\dfrac{\dfrac{}{g : [\cdots], x : A \vdash g : [\cdots]}}{g : [\cdots], x : A \vdash \pi_2 g : A \to B} \quad \dfrac{}{g : [\cdots], x : A \vdash x : A}}{g : [\cdots], x : A \vdash \pi_2 g\, x : B}}{g : (B \to C) \times (A \to C), x : A \vdash \pi_1 g\,(\pi_2 g\, x) : C}}{g : (B \to C) \times (A \to C) \vdash \lambda x.\,\pi_1 g\,(\pi_2 g\, x) : A \to C}}{\cdot \vdash \lambda g.\,\lambda x.\,\pi_1 g\,(\pi_2 g\, x) : (B \to C) \times (A \to B) \to (A \to C)}$$

Of course, no one wants to do this by hand : it's the proof assistant's job!

Equations

When we define a function as an expression such as $f : n \mapsto n \times n + 3$, we understand that $f(5)$ is defined as the expression $5 \times 5 + 3$ (where we replaced every occurence of n with 5)

We want the same thing to work with anonymous functions:

$$(\lambda n.\ n \times n + 3)\ \boxed{\text{something}} \equiv \boxed{\text{something}} \times \boxed{\text{something}} + 3$$

The general rule may be written as $(\lambda x.\ t)\ u \equiv t[u/x]$, where $t[u/x]$ means "replace every occurence of x with u"

NB: when computing $t[u/x]$, we may want to rename some bound variables to avoid name clashes. Consider for instance

$$(\lambda x.\ \lambda y.\ x\ y)(\lambda y.\ y) \xrightarrow{\text{naive}} \lambda y.\ (\lambda y.\ y)\ y$$
$$(\lambda x.\ \lambda y.\ x\ y)(\lambda y.\ y) \xrightarrow{\text{better}} \lambda y.\ (\lambda z.\ z)\ y$$

Since we also have products in our system, we want to add a hanful of computation rules for them, most notably

$$\pi_1 \langle t, u \rangle \equiv t \qquad \text{and} \qquad \pi_2 \langle t, u \rangle \equiv u$$

These rules are respectively called the β-rule for functions and the β-rules for products

η-rules

For completeness, we want to add that

- breaking down a pair and putting it back together does nothing

$$\langle \pi_1 t, \pi_2 t \rangle \equiv t$$

- abstracting and applying immediately after does nothing

$$\lambda x.\ (t\ x) \equiv t$$

These are called η-rules

NB: we might also want to add equations for our base types, such as $0 + t \equiv t$ $\quad \text{suc}(t) + u \equiv \text{suc}(t + u)$

Currification

NB : maybe for exercise sessions

Writing functions of several arguments using pairs is not super convenient... Recall our definition of composition :

$$\begin{aligned} \text{comp} &: (\mathbb{N} \to \mathbb{N}) \times (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\ \text{comp} &\equiv \lambda g . (\lambda x . \pi_1(g)(\pi_2(g)(x))) \end{aligned}$$

Type theorists like replacing these with higher order functions, that is, functions which output another function.

For instance, consider an arbitrary function $f : A \times B \to C$
we can transform it into a function of type $A \to (B \to C)$
by using a partial application : $f' \equiv \lambda a . \lambda b . f\langle a, b\rangle$
(informally, $f'(a) \equiv f(a, -)$)
f' is the curried form of f

Conversely, consider a function $g : A \to (B \to C)$
we can transform it into a function of type $A \times B \to C$
by defining $g' x \equiv (g(\pi_1 x))(\pi_2 x)$
g' is the uncurried form of g

Thus we have two transformations

also written as $A \to B \to C$
arrows are right-associative

$$A \times B \to C \underset{\text{uncurry}}{\overset{\text{curry}}{\rightleftarrows}} A \to (B \to C)$$

One readily checks that they are inverse to each other
NB : this is an instance of a very general phenomenon.
see : cartesian closed categories, Hom-tensor adjunction

The curried form of comp is more legible :

$$\begin{aligned} \text{comp}' &: (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N}) \\ \text{comp}' &= \lambda f . \lambda g . \lambda x . f(g\,x) \end{aligned}$$

For this reason and many other, type theorists always privilege curried functions

The Curry-Howard correspondence

Take the rules of the STLC

[Remember to do a side-to-side table]

structural rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

rules for functions

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\,u : B}$$

rules for products

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash u : B}{\Gamma \vdash \langle t, u \rangle : A \times B} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1(t) : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2(t) : B}$$

And modify them slightly : - erase the terms
- replace $\times$ with $\wedge$ and replace $\to$ with $\Rightarrow$

$$\frac{A \in \Gamma}{\Gamma \vdash A}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$ modus ponens

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

We obtain a set of rules which look a lot like the rules of natural deduction, a system for writing formal proofs from the 30s !

This is called the Curry-Howard correspondence : STLC may be understood either as a system for defining functions, or as a system for writing proofs in natural deduction

Recall for instance our composition function

$$\text{comp} : (B \to C) \times (A \to B) \to (A \to C)$$
$$\text{comp} \equiv \lambda g . (\lambda x . \pi_1(g)\,(\pi_2(g)\,(x)))$$

In the "logical" notation, it becomes $(B \Rightarrow C) \wedge (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$ i.e. a proof that implication is transitive.
More generally, every function in the STLC proves a theorem in natural deduction.

Extending the STLC

Note that we are missing some logical connectives: what about disjunction and negation? The rules of disjunction in natural deduction are the following:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

This suggests adding a new connective to the STLC for disjoint sums $A + B$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathrm{inl}(t) : A + B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathrm{inr}(t) : A + B}$$

$$\frac{\Gamma \vdash t : A + B \quad \Gamma, x : A \vdash u : C \quad \Gamma, x : B \vdash v : C}{\Gamma \vdash \begin{array}{l}\text{match } t \text{ with} \\ \mid \mathrm{inl}(x) \to u \\ \mid \mathrm{inr}(x) \to v\end{array} : C}$$

See also:
- coproducts in category theory
- algebraic datatypes in programming

Now what about negation? Logicians like to encode negation as $\neg A \equiv A \Rightarrow \bot$, where $\bot$ is the false proposition. The false proposition has only one rule

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

if we can prove a contradiction, then we can prove any other proposition!

In the STLC, we add a new empty type $\bot$ with the following rule:

$$\frac{\Gamma \vdash t : \bot}{\Gamma \vdash \mathrm{raise}(t) : A}$$

There is no way to build a term of type $\bot$, however if we somehow have one, we can use it to build anything

Compare to:
- exceptions in programming
- initial object in category theory

With this, we have a complete correspondence natural deduction $\Leftrightarrow$ STLC$^+$