OPLSS 22 Introduction to Type Theory (1)

Cosmo Viola, Nabil Hassein, Daniel Zackon, Jiawei Chen

June 2022

This course will focus on intuition behind type theory instead of the formalism of type theory.

1 What is Type Theory?

- An alternative foundation to Set Theory
- A logic
- A programming language

2 How are Set Theory and Type Theory different?

- **Elementhood is a proposition in set theory but a judgement in type theory.** Elements are not independent of their type, unlike in set theory. For example, $\forall x, x \in A \implies x \in B$ makes sense in in set theory as a definition of subset but not in type theory.
- **Set theory doesn't rule out strange questions about the encoding of elements.** For example, in the encoding of natural numbers in set theory, we can ask if $2 \subseteq 3$, which is valid but terrible. Type theory makes representation invariance easier.
- **The logic of type theory is intuitionistic.** It arises naturally out of the propositions as types explanation.
- **Functions are a primitive of type theory.** In set theory, a function is a special type of relation $f \subseteq A \times B$ such that $\forall x \in A, \exists! y \in B, (x, y) \in f$. (Note that $\exists! y$ means "there exists a unique y".) In type theory, functions are used to define relations: a relation is a function $R : A \to B \to Prop$.

3 What is a function?

A function is a black box that takes in an input and produces an output. Two functions are equal if the functions produces the same output on every input; this is *extensional* equality. We can define a function like

Figure 1: Block-diagram representation of the higher-order function h, which takes a function as an argument

$$f : \mathbb{N} \to \mathbb{N}, f\ n = n + 2$$

We can evaluate f on an element of $\mathbb{N}$. For example,

$$\begin{aligned} & f\ 3 \\ & = (n+2)[n := 3] \text{ (This is } \beta \text{ reduction.)} \\ & = 3 + 2 \\ & = 5 \end{aligned}$$

We can define functions anonymously. For example, $\lambda n \to n + 2$ is the same function as the above but without a name. Then, let $f' = \lambda n \to n + 2$, and we can evaluate $f'\ 3$ similarly to before.

$$\begin{aligned} & f'\ 3 \\ & = (\lambda n \to n + 2)\ 3 \\ & = (n+2)[n := 3] \\ & = 3 + 2 \\ & = 5 \end{aligned}$$

Function types are right associative: $A \to B \to C = A \to (B \to C)$. Function application is left associative: $g\ x\ y = (g\ x)\ y$. However, in general $A \to B \to C \neq (A \to B) \to C$ and $g\ x\ y \neq g\ (x\ y)$.

Functions can be higher order; they can take other functions as inputs and produce functions as outputs. Consider h of type $(\mathbb{N} \to \mathbb{N}) \to \mathbb{N}$, which takes a function of type $\mathbb{N} \to \mathbb{N}$ as input (Fig. 1).

Let $g : \mathbb{N} \to \mathbb{N} \to \mathbb{N},\ g\ x\ y = y + x$ (Fig. 2). Suppose we have $y : \mathbb{N}$, and consider $g\ y$. If we simply substitute y for x in g, we will have two different ys with two different meanings. We use α-conversion to solve this:

$$\begin{aligned} g\ y & = (\lambda x \to \lambda y \to y + x)\ y \\ & = (\lambda y \to y + x)[x := y] \\ & = (\lambda z \to z + x)[x := y] \text{ (}\alpha\text{-conversion)} \\ & = \lambda z \to z + y \end{aligned}$$

Finally, consider a function $hh : \mathbb{N} \to \mathbb{N}, hh = \lambda n \to f'n$. There is an η-rule which says that $hh = f'$ definitionally, or judgementally; this is despite the fact that the functions are defined differently.

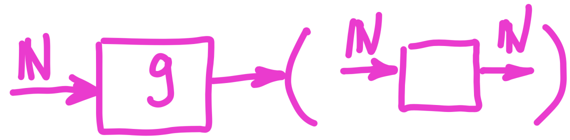

Figure 2: Block-diagram representation of g

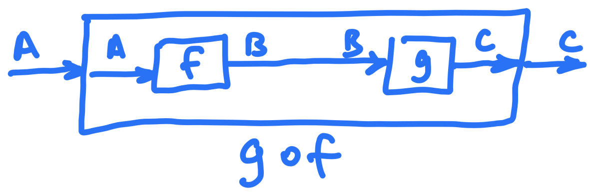

Figure 3: Block-diagram representation of $g \circ f$

4 Combinators

Combinators are functions that only use pure lambda calculus. One combinator, the identity, is

$$\begin{aligned} &id : A \to A \\ &id\ x = x. \end{aligned}$$

Another combinator, composition (Fig. 3), is

$$\begin{aligned} &\_ \circ \_ : (B \to C) \to (A \to B) \to (A \to C) \\ &(g \circ f)\ x = g\ (f\ x). \end{aligned}$$

Here is a third combinator, the K combinator:

$$\begin{aligned} &K : A \to B \to A \\ &K\ a\ b = a \end{aligned}$$

Here is a fourth combinator, the S combinator:

$$\begin{aligned} &S : (A \to B \to C) \to (A \to B) \to (A \to C) \\ &S\ f\ g\ a = f\ a\ (g\ a) \end{aligned}$$

S takes as arguments functions f and g (Fig. 4).

K and S are sufficient to define all other combinators. The motivation behind this was

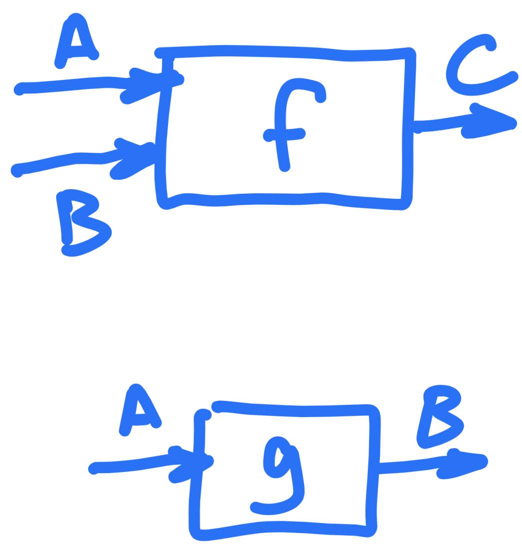

Figure 4: Block-diagram representation of the argument functions f and g

to eliminate the use of variables when defining new functions. We can define the identity in terms of S and K:

$$\begin{aligned}&I : A \to A\\&I = S\ K\ K\end{aligned}$$

Recall the $\circ$ combinator, but written as prefix instead of infix:

$$\begin{aligned}&CC : (B \to C) \to (A \to B) \to (A \to C)\\&CC = \lambda g\ f\ x \to g\ (f\ x)\end{aligned}$$

We will give equations which can be used to transform general lambda terms into lambda terms only using the S and K combinators. Note that $\lambda x \to x = I$, $\lambda x \to y = K\ y$, and $\lambda x \to M\ N = S\ (\lambda x \to M)\ (\lambda x \to N)$. This last equation holds because

$$\begin{aligned}&(\lambda x \to M\ N)\ L\\&= (M\ N)[x := L]\\&= M[x := L]\ N[x := L],\end{aligned}$$

while

$$\begin{aligned}&S\ (\lambda x \to M)\ (\lambda x \to N)\ L\\&= (\lambda x \to M)\ L\ ((\lambda x \to N)\ L)\\&= M[x := L]\ N[x := L]\end{aligned}$$

If x does not occur in M, then $\lambda x \to M = K\ M$ and $\lambda x \to M\ x = M$. Then, here is the same combinator CC, defined in terms of S and K using the equations given above:

$$
\begin{aligned}
&CC - sk : (B \to C) \to (A \to B) \to (A \to C)\\
&\lambda f\ g\ a \to f\ (g\ a)\\
&= \lambda f \to \lambda g \to \lambda a \to f\ (g\ a)\\
&= \lambda f \to \lambda g \to S\ (\lambda a \to f)\ (\lambda a \to g\ a)\\
&= \lambda f \to \lambda g \to S\ (K\ f)\ g\\
&= \lambda f \to \lambda g \to (S\ (K\ f))\ g\\
&= \lambda f \to S\ (K\ f)\\
&= S\ (\lambda f \to S)\ (\lambda f \to K\ f)\\
&= S\ (K\ S)\ K
\end{aligned}
$$