OREGON PROGRAMMING LANGUAGES SUMMER SCHOOL

Introduction to Type Theories — Anja Petković Komel

Lecture 2 - *June 24, 2025*

We continue from last lecture, introducing the rules of type theories. As these lectures are designed around Egbert Rijke's book [1], these notes won't try to replicate the exact content from the book, but instead act as a companion. Furthermore, they will include specificities from the lecture that potentially do not appear in the book.

As a reminder, we have four kinds of judgments, namely for well-formed type, judgementally equal types, well-formed term, and judgementally equal terms.

$$\Gamma \vdash A \text{ type} \qquad \Gamma \vdash A \doteq B \text{ type} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash a \doteq b : A$$

and, for each type construct, we will need five kinds of rules:

- Formation rules, which define how we can form the type
- Introduction rules, which define how we can create values of that type
- Elimination rules, which define how we can use values of that type and extract information
- Computation rules, which define how the introduction and elimination rules interact
- Congruence rules (sometimes omitted, in which case they are left implicit), which define that all introduced terms are well-defined with respect to judgmental equality

1 Dependent functions

We will write $\prod_{(x:A)} B(x)$ for the type of functions which take an argument $x : A$ and return a value of type $B(x)$. It is also called the *dependent product*. You can refer to Rijke for more detailed explanations of the rules. To give an overview roughly, they are

- Formation rule: $\prod_{(x:A)} B(x)$ is a type if $B(x)$ is a type with $x : A$ in the context.

$$\frac{\Gamma, x : A \vdash B(x) \text{ type}}{\Gamma \vdash \prod_{(x:A)} B(x) \text{ type}}\, \Pi$$

A concrete example of this would be:

$$\frac{x : \mathbb{N} \vdash \mathrm{Vec}\, \mathbb{N}\, x \text{ type}}{\vdash \prod_{(x:\mathbb{N})} \mathrm{Vec}\, \mathbb{N}\, x \text{ type}}$$

- Congruence Rule: omitted for it's simple to understand.

$$\frac{\Gamma \vdash A \doteq A' \text{ type} \qquad \Gamma, x : A \vdash B(x) \doteq B'(x) \text{ type}}{\Gamma \vdash \prod_{(x:A)} B(x) \doteq \prod_{(x:A')} B'(x) \text{ type}}\, \Pi\text{-eq}$$

- Introduction rule: If the term $b(x)$ has type $B(x)$ with $x : A$ in the context, then the lambda term $\lambda x.b(x)$ has type $\prod_{(x:A)} B(x)$. This is how we can introduce a term with a dependent function type.

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

A concrete example:

$$\frac{x : \mathbb{N} \vdash (0, 0, \ldots, 0) : \mathrm{Vec}\, \mathbb{N}\, x}{\vdash \lambda x.(0, 0, \ldots, 0) : \prod_{(x:\mathbb{N})} \mathrm{Vec}\, \mathbb{N}\, x}$$

- Elimination (evaluation) rule: If the function, f has type $\prod_{(x:A)} B(x)$, then a $x : A$ can be introduced into the context and f applied to x has type $B(x)$. This is function application.

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B(x)}{\Gamma, x : A \vdash f(x) : B(x)}\, ev$$

Here is a simpler concrete example of this:

$$\frac{\vdash \lambda x.(0, 0, \ldots, 0) : \prod_{(x:\mathbb{N})} \mathrm{Vec}\, \mathbb{N}\, x}{x : \mathbb{N} \vdash (0, 0, \ldots, 0) : \mathrm{Vec}\, \mathbb{N}\, x}\, ev$$

Here is another example to illustrate why this is related to evaluation, once paired with substitution.

$$\frac{\vdash \lambda x.(0,0,\ldots,0) : \prod_{(x:\mathbb{N})} \text{Vec}\ \mathbb{N}\, x}{x : \mathbb{N} \vdash (0,0,\ldots,0) : \text{Vec}\ \mathbb{N}\, x}\ ev$$

- Computation rules: β and η

  β

  $$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma, x : A \vdash (\lambda y.b(y))(x) \doteq b(x) : B(x)}\ \beta$$

  - This rule shows local soundness. It says that a function may be constructed (as $\lambda y.b(y)$), then immediately eliminated by applying it to an argument (x), and the original type ($B(x)$) is preserved.
  - *A note about notation.* In the β rule

    $$\frac{\Gamma, x : A \vdash b(x) : B(x)}{\Gamma, x : A \vdash (\lambda y.b(y))(x) \doteq b(x) : B(x),}\beta$$

    the notation $-(-)$ is overloaded to mean two different things: it is both the syntax for evaluation (*i.e.*, function application), and also the syntax for a term with a free variable / substituting a free variable (*i.e.* dependent on). Using ev for the first, and more traditional substitution notation for the second, you might write this rule as

    $$\frac{\Gamma, x : A \vdash b : B}{\Gamma, x : A \vdash \mathsf{ev}(\lambda y.b, x) \doteq b[x/y] : B[x/y],}\beta$$

    where x may be free in both b and B.

  η

  $$\frac{\Gamma \vdash f : \prod_{(x:A)} B(x)}{\Gamma \vdash \lambda x.f(x) \doteq f : \prod_{(x:A)} B(x)}\ \eta$$

  - Function elimination ($f(x)$) then construction ($\lambda x.f(x)$) (*i.e.* application then abstraction) perseveres the original type ($\prod_{(x:A)} B(x)$).
  - *A note about function extensionality.* We have the η rule

    $$\frac{\Gamma \vdash f : \displaystyle\prod_{(x:A)} B(x)}{\Gamma \vdash f \doteq \lambda x.f(x) : \displaystyle\prod_{(x:A)} B(x).}\eta$$

An alternative rule which the η rule can be derived from (sometimes called *extensionality*, although this term is very heavily overloaded) is

$$\frac{\Gamma, x : A \vdash f(x) \doteq g(x) : B(x)}{\Gamma \vdash f \doteq g : \prod_{(x:A)} B(x).}\eta'$$

While η is a more traditional way to write this rule, η' is more along the lines of how this rule is actually implemented in a type checker: to check that two functions f and g are equal (below the line), it suffices to check that $f(x) \doteq g(x)$ in a context extended by a fresh variable $x : A$ (above the line).

- Congruence rules: omitted

1.1 Ordinary Function Types

A special case of (dependent) function type arises when both A and B are existing types within the context Γ. In this case, the codomain has no "real" dependency.

The following definitions of functions and arrow types are directly quoted from Rijke[1].

A term $f : \prod_{(x:A)} B$ is a function that takes an argument $x : A$ and returns $f(x) : B$. In other words, terms of type $\prod_{(x:A)} B$ are indeed ordinary functions from A to B. Therefore, we define the type $A \to B$ of **(ordinary) functions** from A to B by $A \to B := \prod_{(x:A)} B$.

If $f : A \to B$ is a function, then the type of A is also called the *domain* of f, and type of B is also called the *codomain* of f.

- If A and B are well-formed types without depending on the function argument, but could be looked up in the existing context Γ, then it is an "arrow" type.
- This rule (also see page 14 of [1]) is also an example of how we can introduce new notation and symbols that extends the type theory.

To state these formally, we define as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma, x : A \vdash B \text{ type}} W}{\Gamma \vdash \prod_{(x:A)} B \text{ type}} \Pi}{\Gamma \vdash A \to B := \prod_{(x:A)} B \text{ type}}$$

1.2 Derivations

The exercise of providing derivations for the identity function and function composition was to show that it is quite painful. A proof assistant will do much of this for us. The full derivations may be found on pages 15 (identity) and 16 (composition). The proof trees are as follows:

Identity:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash A \text{ type}}{\Gamma, x : A \vdash x : A}}{\Gamma \vdash \lambda x.x : A \to A}}{\Gamma \vdash \mathrm{id}_A := \lambda x.x : A \to A}$$

Composition:

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\Gamma \vdash A \text{ type} \qquad \Gamma \vdash B \text{ type}}{\Gamma, f : B^A, x : A \vdash f(x) : B}\,(\mathrm{a}) \qquad \cfrac{\Gamma \vdash B \text{ type} \qquad \Gamma \vdash C \text{ type}}{\Gamma, g : C^B, y : B \vdash g(y) : C}\,(\mathrm{b})}{\Gamma, g : C^B, f : B^A, x : A \vdash f(x) : B \quad \Gamma, g : C^B, f : B^A, x : A, y : B \vdash g(y) : C}}{\Gamma, g : C^B, f : B^A, x : A \vdash g(f(x)) : C}}{\Gamma, g : C^B, f : B^A \vdash \lambda x.g(f(x)) : C^A}}{\Gamma, g : B \to C \vdash \lambda f.\lambda x.g(f(x)) : B^A \to C^A}}{\Gamma \vdash \lambda g.\lambda f.\lambda x.g(f(x)) : C^B \to (B^A \to C^A)}}{\Gamma \vdash \mathrm{comp} := \lambda g.\lambda f.\lambda x.g(f(x)) : C^B \to (B^A \to C^A)}$$

Note that, B^A is just an alternative form of writing $A \to B$.

2 Natural numbers

We introduce a type $\mathbb{N}$ of natural numbers. You can refer to Rijke and the slides for the rules. Collectively, they are:

- Formation rule: $\mathbb{N}$ is a type

$$\cfrac{}{\vdash \mathbb{N} \text{ type}}\ \mathbb{N}\text{-form}$$

- Introduction rules: 0, successors $\mathsf{succ}(n)$

$$\frac{}{\vdash 0_{\mathbb{N}} : \mathbb{N}}$$

$$\frac{}{\vdash \mathrm{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}$$

- Elimination rule: the induction principle for natural numbers

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) \text{ type} \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_s : \prod_{(n:\mathbb{N})} P(n) \to P(\mathrm{succ}_{\mathbb{N}}(n))}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_s) : \prod_{(n:\mathbb{N})} P(n)}\ \mathbb{N}\text{-ind}$$

  - Notice that in this rule, propositions are types. We are proving a proposition, so it must have a type.
  - Furthermore, remember that a proof is a well-formed derivable judgment that a term has a type.

- Computation rules: how recursion evaluates when given 0 or $\mathsf{succ}(n)$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) \text{ type} \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_s : \prod_{(n:\mathbb{N})} P(n) \to P(\mathrm{succ}_{\mathbb{N}}(n))}{\Gamma \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_s, 0_{\mathbb{N}}) \doteq p_0 : P(0_{\mathbb{N}})}$$

$$\frac{\Gamma, n : \mathbb{N} \vdash P(n) \text{ type} \qquad \Gamma \vdash p_0 : P(0_{\mathbb{N}}) \qquad \Gamma \vdash p_s : \prod_{(n:\mathbb{N})} P(n) \to P(\mathrm{succ}_{\mathbb{N}}(n))}{\Gamma, n : \mathbb{N} \vdash \mathrm{ind}_{\mathbb{N}}(p_0, p_s, \mathrm{succ}_{\mathbb{N}}(n)) \doteq p_s(n, \mathrm{ind}_{\mathbb{N}}(p_0, p_s, n)) : P(\mathrm{succ}_{\mathbb{N}}(n))}$$

- Congruence rules: omitted

Example (addition). The term $\mathsf{ind}_{\mathbb{N}}(-,-)$ lets us do recursion on natural numbers. We'd like to define $\mathsf{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, satisfying the (pattern matching) equations

$$\begin{aligned} \mathsf{add}\ m\ 0 &= m \\ \mathsf{add}\ m\ \mathsf{succ}(n) &= \mathsf{succ}(\mathsf{add}\ m\ n) \end{aligned}$$

We can do this using the term

$$\mathsf{add} \coloneqq \lambda m.\mathsf{ind}(m, \lambda n.\lambda a.\mathsf{succ}(a))$$

which satisfies the desired pattern matching equations.

It's a lot easier to implement functions with pattern matching, but it is actually equivalent[1]! Proof assistants like Lean implement pattern matching by automatically translating it to a suitable use of ind, both for $\mathbb{N}$ and for any other inductively-defined type.

Other inductive types. Some other inductive types, whose rules follow a similar pattern to those for $\mathbb{N}$, include:

[1] Or at least, can be equivalent, depending on the exact rules for pattern matching and for ind

- The unit type (1)
- The empty type ($\varnothing$), with inductive principle $\text{ind}_{\emptyset} : \prod_{(x:\emptyset)} P(x)$, and its non-dependent version **ex-falso** $:= \text{ind}_{\emptyset} : \emptyset \to A$
- The dependent sum / coproduct $\Sigma_{(x:A)} B(x)$
- Propositional Equality ($=$)

3 Dependent pairs

Given a type A and a type family $x : A \vdash B(x)$ type, we define a type whose elements are pairs (a, b), with $a : A$ and $b : B(a)$. This type is written $\sum_{(x:A)} B(x)$, and sometimes also called the *dependent sum* type. It is equipped with a **pairing function**

$$\text{pair} : \prod_{(x:A)} \left(B(x) \to \sum_{(y:A)} B(y) \right).$$

A note on confusing terminology The non-dependent pair type, $A \times B$, is often called the product of A and B. However, we use "dependent product" to mean the type of dependent *functions*, and "dependent sum" to mean the type of dependent pairs.

To give a concrete example of this, imagine a vector of length 2, we will have:

$$\text{pair } 2\ (0, 0) : \Sigma_{(n:\mathbb{N})} \text{Vec } \mathbb{N}\ n$$

We can define the *projections* on pairs, and such projections would be the *elimination rules* of dependent sums. Consider a type A and a type family B over A.

- The first projection map
$$\text{pr}_1 : \left(\sum_{(x:A)} B(x) \right) \to A$$
is defined as $\text{pr}_1(x, y) := x$.
- The second projection map
$$\text{pr}_2 : \Pi_{(p:\sum_{(x:A)} B(x))} B(\text{pr}_1(p))$$
is defined as $\text{pr}_2(x, y) := y$.

Note that, from the induction principle given in the textbook, such definitions can also be postulated with induction. See page 14 of the textbook for further deductions.[1]

References

[1] Egbert Rijke. *Introduction to Homotopy Type Theory*. 2022. arXiv: 2212.11082 [math.LO]. URL: https://arxiv.org/abs/2212.11082.