22

MARTIN-LÖF TYPE THEORY

Type theory can be seen as an intuitionistic foundation of mathematics.

- **Martin-Löf Type Theory** (MLTT) can be used as a constructive and computational foundation of mathematics. It is not directly based on first-order predicate logic, but only interpreted in it via the Curry-Howard isomorphism. It comes in two flavours, intensional and extensional.
- **Intensional type theory** (ITT) is an intuitionistic type theory serving as the core to many other type theories.
- **Extensional type theory** (ETT) extends Intensional Type Theory with equality of reflection and uniqueness of identity proofs. Types behave like sets when we interpret them in this setting, and therefore, it can be seen as an intuitionistic theory of sets.

The principal difference between types and sets is that In Type theory, membership of an element to a type, as in $a : A$, is a *judgement* of the theory instead of a *proposition*. In practice, this means that that we can write things like $\forall a.a \in A \to (a, a) \in A \times A$, as $a \in A$ **is** a proposition; but we cannot write $\prod_{a:A}(a, a) : A \times A$. A type declaration is not part of the language.

This section follows [Course on Hott, Harper] and [Hott].

We introduce ITT with Naturals, Sigma, Pi, Identity types and Universes. [Martin-Löf 73]

22.1 CONTEXTS AND JUDGMENTS

As we did when we described simply typed λ-calculus; we will present a type theory using contexts, substitutions and some inference rules. We consider three different kinds of judgments for this formal system, namely,

- Γ `ctx`, expressing that Γ is a context;
- $\Gamma \vdash a : A$, expressing that, from the context Γ, it follows that a is of type A; and

- $\Gamma \vdash a \equiv a' : A$, expressing that, from the context Γ, it follows that a is *definitionally equal* to a'. It is important not to confuse this notion of equality with the notion of *propositional equality* we will describe later.

Contexts, in particular, will be given by a (possibly empty) list of type declarations

$$x_1{:}A_1,\ x_2{:}A_2,\ \ldots,\ x_n{:}A_n,$$

and we will consider as valid the judgement saying that the empty context is a context ($\cdot$ `ctx`), and the judgement saying that, given a context $x_1{:}A_1,\ x_2{:}A_2,\ \ldots, x_{n-1}{:}A_{n-1}$ and any type A_n, we can take a new unused variable to form a new context

$$x_1{:}A_1,\ x_2{:}A_2,\ \ldots, x_{n-1}{:}A_{n-1}, x_n{:}A_n\ \texttt{ctx}.$$

22.2 TYPE UNIVERSES

If we want to blur the difference between types and terms, we will need type declarations for types. Types whose elements are types are called ***universes***. We would like, then, to define a single universe of all types $\mathcal{U}$ containing even itself as an element, $\mathcal{U} : \mathcal{U}$; but this leads to paradoxes. In particular, we can encode a particular version of Russell's paradox.

To avoid having to deal with paradoxes, we will postulate a ***hierarchy of cumulative universes***; that is, we will consider a list of inclusions between countably many type universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \ldots$$

and such that $a : \mathcal{U}_i$ implies $a : \mathcal{U}_{i+1}$. The relevant inference rules are

$$\frac{}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}$$

for each i in the natural numbers.

Universe indices, however, are usually implicitly assumed when writing ITT; as we will mostly work with types from some fixed universe of the hierarchy $\mathcal{U}$ and only refer to higher universes when necessary. This should not lead to confusion: a derivation will be only valid if we can assign indices to the universes consistently, even if we do not explicitly write them.

22.3 DEFINING TYPES

Types will be defined by the formation and elimination rules we can apply to them. We will follow a general pattern for specifying the typing rules of ITT. They will be classified into

- **formation rules**, indicating how to create new types of a particular kind;
- **introduction rules**, indicating how to create new terms of a particular type;
- **elimination rules**, indicating how to use elements of a particular type;
- β**-reductions**, indicating how elimination acts on terms;
- η**-reductions**, defining some kind uniqueness principle.

Depending in how the uniqueness principle is defined, we classify types into negative and positive types.

- Uniqueness of **negative types** states that every element of the type can be reconstructed by applying eliminators to it and then applying a constructor. Products are an example of negative types.
- Uniqueness of **positive types** states that every funtion from the type is determined by some data. Coproducts are an example of positive types.

22.4 DEPENDENT FUNCTION TYPES

Dependent function types, or Π**-types**, are the generalized version of the function types in simply-typed lambda calculus. The elements of $\prod_{x:A} B(x)$ are functions with the type A as domain and a changing codomain $B(x)$, depending on the specific element to which the function is applied. This type is often written also as $\Pi(x : A), B(x)$ to resemble the universal quantifier; under the *propositions as types* interpretation, it would correspond to the proof that a proposition B holds for any $x : A$, that is, $\forall(x : A), B(x)$.

Definition 112 (Dependent function type)**.** The following rules apply for the dependent function type:

- its formation rule sinthetizes the fact that the codomain type can depend on the argument to which the function is applied,
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : U_i}{\Gamma \vdash \prod_{x:A} B : \mathcal{U}_i}\ \Pi\text{-FORM}$$
- its introduction rule its a generalized version of λ-abstraction,
$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : \prod_{x:A} B}\ \Pi\text{-INTRO}$$
- and its elimination rule generalizes function application,
$$\frac{\Gamma \vdash f : \prod_{x:A} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]}\ \Pi\text{-ELIM}$$
- its β-rule is similar to simply typed lambda calculus' β-rule, with the difference that it has to specify a substitution on the type of the codomain
$$\frac{\Gamma, x : A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda x.b)\ a \equiv b[a/x] : B[a/x]}\ \Pi\text{-COMP}$$

- and its η-rule is also similar to simply typed lambda calculus' one

$$\frac{\Gamma \vdash f : \prod_{x:A} B}{\Gamma \vdash f \equiv \lambda x.f(x) : \prod_{x:A} B}\ \Pi\text{-UNIQ}$$

In the particular case in which x does not appear in B so that B is constant and does not depend on A, we obtain our usual **function type**. In ITT, we define the function type as a particular case of the dependent function type, $A \to B :\equiv \prod_{x:A} B$.

Polymorphic functions can be defined in terms of dependent function types and universes. We can see a polymorphic term as a dependent function taking a type as its first argument. For example, the identity function $\mathrm{id} : \prod_{A:\mathcal{U}} A \to A$ can be defined as $\mathrm{id} :\equiv \lambda(A : \mathcal{U}).\lambda(x : A).x$. When applied, the type should be passed as an argument, but it is a common convention to omit arguments when they are obvious from the context and only refer to them when typechecking. A more complex example of polymorphic function is *function composition*, which is also an example of higher-order function. Its type signature takes three types as arguments and two functions between these types,

$$\circ : \prod_{A:\mathcal{U}_i} \prod_{B:\mathcal{U}_i} \prod_{C:\mathcal{U}_i} (A \to B) \to (B \to C) \to A \to C$$

and it is defined as

$$\circ :\equiv \lambda(A : \mathcal{U}_i).\lambda(B : \mathcal{U}_i).\lambda(C : \mathcal{U}_i).\ \lambda g.\lambda f.\lambda x.\ g(f(x)).$$

However, we face a notational nuisance here: to explicitly write down all the types each time we want to notate a function would be very wordy. Instead, we will allow arguments to be determined implicitely. Implicit arguments must be inferred by the context and its supression is not formalized as part of our type theory. At the same time, we will use infix notation whenever it is easier to read.

Thus, we will write $f \circ g$ instead of $\circ(A, B, C, f, g)$.

22.5 DEPENDENT PAIR TYPES

Dependent pair types, or **Σ-types**, can be seen as a generalized version of the product type, but they can be also particularized in the union type. The elements of $\sum_{x:A} B(x)$ are pairs where the first element is of type A and the second element is of type $B(x)$, where x is the first one; that is, the type of the second component depends on the first component. This type is often written as $\Sigma(x : A), B(x)$ and it corresponds to the intuitionistic existential quantifier under the *propositions as types* interpretation. That is, the proof of $\exists(x : A), B(x)$ must be seen as a pair given by an element x and a proof of $B(x)$.

Definition 113 (Dependent pair type)**.** The following rules apply for the dependent pair type:

- its formation rule is similar to that of the product

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x : A \vdash B : \mathcal{U}_i}{\Gamma \vdash \sum_{x:A} B : \mathcal{U}_i}\ \Sigma\text{-FORM}$$

- a pair can be constructed by its two components

$$\frac{\Gamma, x : A \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \sum_{x:A} B}\ \Sigma\text{-INTRO}$$

- its elimination rule tell us that we can use a pair using its two elements, x, y, to create a new one

$$\frac{\Gamma, z : \sum_{x:A} B \vdash C : \mathcal{U}_i \qquad \Gamma, x : A, y : B \vdash g : C[(x,y)/z] \qquad \Gamma \vdash p : \sum_{x:A} B}{\Gamma \vdash \mathrm{ind}_{\sum}([z].C, [x].[y].g, p) : C[p/z]}\ \Sigma\text{-ELIM}$$

- and its β-rule substitutes the two elements of the pair onto an element

$$\frac{\Gamma, z : \sum_{x:A} B \vdash C : \mathcal{U}_i \qquad \Gamma, x : A, y : B \vdash g : C[(x,y)/z] \qquad \begin{array}{c}\Gamma \vdash a : A \\ \Gamma \vdash b : B[a/x]\end{array}}{\Gamma \vdash \mathrm{ind}_{\sum}([z].C, [x].[y].g, (a,b)) \equiv g[a/x][b/y] : C[(a,b)/z]}\ \Sigma\text{-COMP}$$

Note that we do not postulate an η-rule for the dependent pair type. We will show later that it follows from these rules.

The elimination rule says that we can define a function from the dependent pair type $\prod_{x:A} B$ to an arbitrary type C by assuming some $x : A$ and $y : B(x)$ and constructing a term of type C with them.

For instance, we can declare of the first projection to be $\mathtt{pr}_1 : (\sum_{x:A} B(x)) \to A$ and define it as $\mathtt{pr}_1((a,b)) :\equiv a$. The second projection is slightly more complex, as it must be a dependent function

$$\mathtt{pr}_2 : \prod_{p:\sum_{x:A} B(x)} B(\mathtt{pr}_1(p))$$

whose type depends on the first projection. It can be again defined as $\mathtt{pr}_2((a,b)) :\equiv b$.

An interesting example arises when we consider a function that builds a function from a term R that can be regarded as a proof-relevant binary relation, in that any term of type $R(x,y)$ is a witness of the relation between some particular x and y.

$$\mathtt{ac} : \left(\prod_{x:A} \sum_{y:B} R(x,y)\right) \to \left(\sum_{f:A\to B} \prod_{x:A} R(x, f(x))\right)$$

Under the propositions as types interpretation, this type represents the **axiom of choice**; that is, if for all $x \in A$ there exists a $y \in B$ such that $R(x,y)$, then there exists a function $f : A \to B$ such that $R(x, f(x))$ for all $x \in A$.

The actual **axiom of choice** can be shown to be independent of Zermelo-Fraenkel set theory. However, maybe surprisingly, this type theoretic version can be directly proved from the axioms of our system. The function $\mathtt{ac}$ can be constructed as

$$\mathtt{ac} :\equiv \lambda g.\, (\mathtt{pr}_1 \circ g, \mathtt{pr}_2 \circ g)\,,$$

and we can check that, in fact, if $g : \prod_{x:A} \sum_{y:B} R(x,y)$, then

- $\mathtt{pr}_1 \circ g$ is of type $A \to B$, and
- $\mathtt{pr}_2 \circ g$ is of type $\prod_{x:A} R(x, \mathtt{pr}_1(g(x)))$;

effectively proving that $\mathtt{ac}(g)$ has type $\sum_{f:A\to B} \prod_{x:A} R(x, f(x))$.

Why the type-theoretic version of the axiom of choice is, instead, a **theorem of choice**? The crucial notion here is that no choice is actually involved. The dependent pair Σ differs from the classical existential quantification $\exists$ in that it is constructive, that is, any witness of $\sum_{a:A} B$ has to actually provide an element of A such that $B(a)$; while a proof of $\exists a \in A, B$ does not need to point to any particular element of A.

Algebraic structures can be also described as dependent pairs. For instance, we can define a **magma** to be a type A endowed with a binary operation $A \to A \to A$. This can be encoded in a type of magmas as

$$\mathtt{Magma} :\equiv \sum_{A:\mathcal{U}} (A \to A \to A).$$

After we introduce identity types, we will be able to add constraints to our structures that will allow us to define more complex algebraic structures such as groups or categories.

22.6 UNIT, EMPTY, COPRODUCT AND BOOLEAN TYPES

The **empty** type, $0 : \mathcal{U}$, has no inhabitants and a function $f : 0 \to C$ is defined without having to give any equation.

Definition 114 (Empty type)**.** The following rules apply for the empty type:

- it can be formed without any assumption,
$$\frac{}{\Gamma \vdash 0 : \mathcal{U}_i}\ 0\text{-FORM}$$
- and it can be used without any restriction,
$$\frac{\Gamma, x : 0 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash a : 0}{\Gamma \vdash \mathrm{ind}_0([x].C, a) : C[a/x]}\ 0\text{-ELIM}$$

Note that the empty type has no introduction and β-rules. We cannot compute with the empty type and it should not be possible to create an element of that type.

The **unit** type, $1 : \mathcal{U}$ has only one inhabitant. To define a function $f : 1 \to C$ amounts to choose on element of type C.

Definition 115 (Unit type)**.** The following rules apply for the unit type:

- it can be formed without any assumption
$$\frac{}{\Gamma \vdash 1 : \mathcal{U}_i}\ 1\text{-FORM}$$
- and its only instance can be also introduced without any assumption,
$$\frac{}{\Gamma \vdash \star : 1}\ 1\text{-INTRO}$$

- it can be used to choose an instance of any type

$$\frac{\Gamma, x : 1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\star/x] \qquad \Gamma \vdash a : 1}{\Gamma \vdash \mathrm{ind}_1([x].C, c, a) : C[a/x]}\ 1\text{-ELIM}$$

- and its beta rule computes the element we had chosen

$$\frac{\Gamma, x : 1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\star/x]}{\Gamma \vdash \mathrm{ind}_1([x].C, c, \star) \equiv c : C[\star/x]}\ 1\text{-COMP}$$

Uniqueness need not to be postulated, it can be proved using these axioms.

Definition 116 (Coproduct type)**.** The following rules apply for coproduct types:

- there is a coproduct type for any two types,

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash A + B : \mathcal{U}_i}\ +\text{-FORM}$$

- a term can be introduced in two different ways

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathtt{inl}(a) : A + B}\ +\text{-INTRO1}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathtt{inr}(b) : A + B}\ +\text{-INTRO2}$$

- and its elimination rule has to consider both cases

$$\frac{\begin{array}{ccc} & \Gamma, x{:}A \vdash c : C[\mathtt{inl}(x)/z] & \\ \Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i & \Gamma, y{:}B \vdash d : C[\mathtt{inr}(y)/z] & \Gamma \vdash e : A + B \end{array}}{\Gamma \vdash \mathtt{ind}_{A+B}([z].C, [x].c, [y].d, e) : C[e/z]}\ +\text{-ELIM}$$

- in particular, we have two β-rules, each one for each different way in which a term can be introduced

$$\frac{\begin{array}{ccc} & \Gamma, x{:}A \vdash c : C[\mathtt{inl}(x)/z] & \\ \Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i & \Gamma, y{:}B \vdash d : C[\mathtt{inr}(y)/z] & \Gamma \vdash a : A \end{array}}{\Gamma \vdash \mathtt{ind}_{A+B}([z].C, [x].c, [y].d, \mathtt{inl}(a)) \equiv c[a/x] : C[\mathtt{inl}(a)/z]}\ +\text{-COMP1}$$

$$\frac{\begin{array}{ccc} & \Gamma, x{:}A \vdash c : C[\mathtt{inl}(x)/z] & \\ \Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i & \Gamma, y{:}B \vdash d : C[\mathtt{inr}(y)/z] & \Gamma \vdash b : B \end{array}}{\Gamma \vdash \mathtt{ind}_{A+B}([z].C, [x].c, [y].d, \mathtt{inr}(b)) \equiv d[b/x] : C[\mathtt{inr}(b)/z]}\ +\text{-COMP2}$$

22.7 NATURAL NUMBERS

Although we will see later that it can be seen as a particular case of inductive datatypes, a **natural numbers** type can be directly defined in the core of our type theory. Its introduction rules will match Peano's axioms and its elimination and β-rules will provide us with the notion of induction.

Definition 117 (Natural numbers)**.** The following rules apply for natural numbers:

- the natural numbers type can be formed without any assumption,
$$\frac{}{\Gamma \vdash \mathbb{N} : \mathcal{U}_i}\ \mathbb{N}\text{-FORM}$$
- a natural number can be introduced in two ways, as zero or as the successor of a natural number
$$\frac{}{\Gamma \vdash 0 : \mathbb{N}}\ \mathbb{N}\text{-INTRO1} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{succ}(n) : \mathbb{N}}\ \mathbb{N}\text{-INTRO2}$$
- we can apply induction on natural numbers
$$\frac{\begin{array}{c}\Gamma \vdash c_0 : C[0/x] \\ \Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathtt{succ}(x)/x] \qquad \Gamma \vdash n : \mathbb{N}\end{array}}{\Gamma \vdash \mathtt{ind}_{\mathbb{N}}([x].C, c_0, [x].[y].c_s, n) : C[n/x]}\ \mathbb{N}\text{-ELIM}$$
- and it must be interpreted recursively for zero or a successor with the following two β-rules
$$\frac{\begin{array}{c}\Gamma \vdash c_0 : C[0/x] \\ \Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathtt{succ}(x)/x]\end{array}}{\Gamma \vdash \mathtt{ind}_{\mathbb{N}}([x].C, c_0, [x].[y].c_s, 0) \equiv c_0 : C[0/x]}\ \mathbb{N}\text{-COMP1}$$
$$\frac{\begin{array}{c}\Gamma \vdash c_0 : C[0/x] \\ \Gamma, x{:}\mathbb{N} \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}\mathbb{N}, y{:}C \vdash c_s : C[\mathtt{succ}(x)/x]\end{array}}{\begin{array}{c}\Gamma \vdash \mathtt{ind}_{\mathbb{N}}([x].C, c_0, [x].[y].c_s, \mathtt{succ}\ n) \equiv \\ c_s[n/x][\mathtt{ind}_{\mathbb{N}}([x].C, c_0, [x].[y].c_s, n)/y] : C[\mathtt{succ}(n)/x]\end{array}}\ \mathbb{N}\text{-COMP2}$$

Any function on natural numbers can be defined in two equivalent ways, either using the induction principle or its defining equations.

- If we use the induction principle, a function from the naturals to the type C must be defined as
$$f :\equiv \mathtt{ind}_{\mathbb{N}}(C, c_0, c_s)$$
where c_s can depend on an element of type C. Two defining equations capturing the definition could be written in this case as
$$\begin{aligned} f(0) &:\equiv c_0, \\ f(\mathtt{succ}(n)) &:\equiv c_s, \end{aligned}$$
where c_s would depend on an element of type C that can be interpreted as being $f(n)$, that is, a recursive call to the function.
- Conversely, we could define a function by its two defining equations. If we had
$$\begin{aligned} f(0) &:\equiv \Phi_0, \\ f(\mathtt{succ}(n)) &:\equiv \Phi_s, \end{aligned}$$
where $\Phi_s : C$ depends on $f(n)$, we could substitute $f(n)$ by a variable r and equivalently write the definition as
$$f :\equiv \mathtt{ind}_{\mathbb{N}}(C, \Phi_0, \lambda n.\lambda r.\Phi_s).$$

This can be done in general with types whose induction principle can be split in multiple cases. We will call **pattern matching** to this style of defining functions and it will be used from now on because of its simplicity. However, we must be careful when doing this, as it could create the false impression that arbitrary recursion is allowed in our definitions. An equation such as $f(\mathtt{succ}(n)) :\equiv f(\mathtt{succ}(\mathtt{succ}(n)))$, for example, is not valid.

Our first example will be the definition of addition on natural numbers. It will be an infix function $+ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ given by

$$\begin{aligned} 0 + m &:\equiv m, \\ (\mathtt{succ}\ n) + m &:\equiv \mathtt{succ}(n + m), \end{aligned}$$

where we are applying the induction principle to the first argument. In other words,

$$+ :\equiv \lambda(n{:}\mathbb{N}).\lambda(m{:}\mathbb{N}).\mathtt{ind}_{\mathbb{N}}(\mathbb{N}, m, [s].[r].\mathtt{succ}\ r, n).$$

Under this definition it can be shown that, for example, $2 + 2 :\equiv 4$. However, if we wanted to prove the fact that addition is commutative, we would have no way of expressing this in terms of an extensional equality. We would need a notion of propositional equality, that is, an equality $=$ that could be used as a type in an expression, in order to write something like

$$\mathtt{comm}_{+} : \prod_{n:\mathbb{N}} \prod_{m:\mathbb{N}} (n + m = m + n).$$

22.8 IDENTITY TYPES

Under a *propositions as types* interpretation, the proposition that two elements of a type are equal, should correspond to a type; and therefore, equality should correspond to a family of types,

$$= : \prod_{A:\mathcal{U}} A \to A \to \mathcal{U}.$$

We will write the equality type between $a, b : A$ interchangeably as $a =_A b$ or $\mathrm{Id}_A(a, b)$.

Definition 118 (Identity types)**.** The following rules apply for identity types:

- an identity type is defined between any two elements of the same type
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : A}{\Gamma \vdash a =_A b : \mathcal{U}_i} =\text{-FORM}$$
- the only way to introduce an identity is using the principle of **reflexivity**, that is, there exists an element $\mathtt{refl}$ of type $a =_A a$ for each $a : A$,
$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathrm{refl} : a =_A a} =\text{-INTRO}$$

- its elimination rule of $p : x =_A y$ allows us to create an instance of a particular type, or prove a proposition, simply assuming that x and y are in fact the same element and $p = \text{refl}_z$,

$$\frac{\begin{array}{ll}\Gamma, x{:}A, y{:}A, p{:}(x =_A y) \vdash C : \mathcal{U}_i & \Gamma \vdash a : A \\ \Gamma, z{:}A \vdash c : C[z/x][z/y][\text{refl}/p] & \Gamma \vdash b : A \qquad \Gamma \vdash q : a =_A b\end{array}}{\Gamma \vdash \mathtt{ind}_{=_A}([x].[y].[p].C, [z].c, a, b, q) : C[a, b, q/x, y, p]} =\text{-ELIM}$$

- and its β-rule simply substitutes two equal instances by the same one

$$\frac{\begin{array}{c}\Gamma, x{:}A, y{:}A, p{:}x =_A y \vdash C : \mathcal{U}_i \\ \Gamma, z{:}A \vdash c : C[z, z, \mathtt{refl}_z/x, y, p] \qquad \Gamma \vdash a : A\end{array}}{\Gamma \vdash \mathtt{ind}_{=_A}([x].[y].[p].C, [z].c, a, a, \mathtt{refl}_a) \equiv c[a/z] : C[a/x][a/y][\mathtt{refl}_a/p]} =\text{-COMP}$$

Note that the introduction rule on types is a dependent function called **reflexivity** that provides us with a basic way of constructing an element of type $(a =_A b) : \mathcal{U}$,

$$\mathtt{refl} : \prod_{a:A}(a =_A a).$$

In particular, any two definitionally equal terms, such as $2 + 2 :\equiv 4$, are also propositionally equal. In fact, $\mathtt{refl}_4 : 2 + 2 = 4$ is a well-typed statement, precisely because of the fact that $2 + 2$ and 4 are definitionally equal.

The elimination principle and the β-rule for identity types are usually called the **path induction** principle. It states that, given a family of types parametrized over the identity type,

$$C : \prod_{x,y:A}(x =_A y) \to \mathcal{U},$$

every function defined over the `refl` element, $c : \prod_{z:A} C(z, z, \mathtt{refl}_z)$, can be extended to any equality term as

$$f : \prod_{x:A}\prod_{y:A}\prod_{p:x=y} C(x, y, p),$$

and it follows from the β-rule that $f(x, x, \mathtt{refl}_x) :\equiv c(x)$, that is, that it is in fact an extension.

This **path induction** principle could be wrapped into a function

$$\mathtt{ind}_= : \prod_{\left(C:\prod_{x:A}\prod_{y:A}(x=y)\to\mathcal{U}\right)} \left(\prod_{x:A} C(x, x, \mathtt{refl}_x) \to \prod_{x:A}\prod_{y:A}\prod_{p:x=y} C(x, y, p)\right)$$

and the β-rule could be replaced by $\mathtt{ind}_=(C, c, x, x, \mathtt{refl}_x) :\equiv c(x)$. We will use path induction to prove a the principle of **indiscernability of identicals**, that is, the fact that, for every family of types $C : A \to \mathcal{U}$, there is a function between the type associated to propositionally equal elements, that is,

$$f : \prod_{x:A}\prod_{y:A}\prod_{p:x=y} C(x) \to C(y),$$

such that, when applied to reflexivity, it outputs the identity function, $f(x, x, \mathtt{refl}_x) :\equiv \mathtt{id}_{C(x)}$.

We can prove this principle for any $C : A \to \mathcal{U}$ by defining

$$f :\equiv \mathtt{ind}_{=}\left(\lambda x.\lambda y.\lambda p.C(x) \to C(y), \mathtt{id}_{C(x)}\right),$$

and the fact that $f(x, x, \mathtt{refl}_x) :\equiv \mathtt{id}_{C(x)}$ follows from the β-rule.

22.9 INDUCTIVE TYPES

Inductive types provide the intuitive notion of a free type generated by a finite set of constructors. That is, the only elements of an inductive type should be those that can be obtained by applying a fixed finite set of constructors. Thus, we can define functions over inductive types using certain induction principles that define a specific action for each one of the possible constructors.

The **W-type** (or *Brower ordinal*) $\mathsf{W}_{a:A} B(a)$, also written as $\mathsf{W}(a : A), B(a)$, can be thought as an inductive type whose set of constructors is indexed by the type A, such that the constructor indexed by the particular element a has arguments indexed by $B(a)$. In other words, it is a well-founded tree where every node is labeled by an element of a and has a set of child nodes indexed by the type $B(a)$.

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : A \vdash B : \mathcal{U}}{\Gamma \vdash \mathsf{W}_{x:A} B : \mathcal{U}}\ \text{W-FORM}$$

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x{:}B(a) \vdash \mathsf{W}_{x:A} B}{\Gamma \vdash \mathtt{sup}[a]([x].w) : \mathsf{W}_{x:A} B}\ \text{W-INTRO}$$

$$\frac{\begin{array}{c}\Gamma, z : \mathsf{W}_{x:A} B \vdash P : \mathcal{U} \\ \Gamma, a{:}A, p : B(a) \to \mathsf{W}_{x:A} B, h : \prod_{b:B(a)} P(p(b)) \vdash M : P(\mathtt{sup}(p))\end{array}}{\Gamma, z{:}\mathsf{W}_{x:A} B \vdash \mathtt{ind}_{\mathsf{W}}(a, p, [h].M) : P(z)}\ \text{W-ELIM}$$

The relevant endofunctor must be polynomial.

Not all endofunctors have initial algebras, but all polynomial functors do have initial algebras.

Each W-type determines a functor on types given by

$$F(X) = \sum_{a:A} B(a) \to X.$$

Functors of this form are called **polynomial functors**, as they can be written as $\sum_{a:A} X^{B(a)}$ and thought as a generalized polynomial function.

In fact, this functor is an F-algebra with the function

$$\lambda(a, w).\mathrm{sup}[a](w) : F(X) \to X.$$