Homotopy Type Theory

Basics

2.1 Types are higher groupoids

Lemma 2.1.1. *For every type A and every $x, y : A$ there is a function*

$$(x = y) \to (y = x)$$

*denoted $p \mapsto p^{-1}$, such that $\mathsf{refl}_x{}^{-1} \equiv \mathsf{refl}_x$ for each $x : A$. We call p^{-1} the **inverse** of p.*

Lemma 2.1.2. *For every type A and every $x, y, z : A$ there is a function*

$$(x = y) \to (y = z) \to (x = z)$$

*written $p \mapsto q \mapsto p \cdot q$, such that $\mathsf{refl}_x \cdot \mathsf{refl}_x \equiv \mathsf{refl}_x$ for any $x : A$. We call $p \cdot q$ the **concatenation** or **composite** of p and q.*

Equality	Homotopy	∞-Groupoid
reflexivity	constant path	identity morphism
symmetry	inversion of paths	inverse morphism
transitivity	concatenation of paths	composition of morphisms

Lemma 2.1.3. *Suppose $A : \mathcal{U}$, that $x, y, z, w : A$ and that $p : x = y$ and $q : y = z$ and $r : z = w$. We have the following:*

(i) $p = p \cdot \mathsf{refl}_y$ *and* $p = \mathsf{refl}_x \cdot p$.
(ii) $p^{-1} \cdot p = \mathsf{refl}_y$ *and* $p \cdot p^{-1} = \mathsf{refl}_x$.
(iii) $(p^{-1})^{-1} = p$.
(iv) $p \cdot (q \cdot r) = (p \cdot q) \cdot r$.

Theorem 2.1.4 (Eckmann–Hilton)**.** *The composition operation on the second loop space*

$$\Omega^2(A) \times \Omega^2(A) \to \Omega^2(A)$$

is commutative: $\alpha \cdot \beta = \beta \cdot \alpha$, for any $\alpha, \beta : \Omega^2(A)$.

Definition 2.1.5. A **pointed type** (A, a) is a type $A : \mathcal{U}$ together with a point $a : A$, called its **basepoint**. We write $\mathcal{U}_\bullet :\equiv \sum_{(A:\mathcal{U})} A$ for the type of pointed types in the universe $\mathcal{U}$.

Definition 2.1.6. Given a pointed type (A, a), we define the **loop space** of (A, a) to be the following pointed type:

$$\Omega(A, a) :\equiv ((a =_A a), \mathsf{refl}_a).$$

An element of it will be called a **loop** at a. For $n : \mathbb{N}$, the **n-fold iterated loop space** $\Omega^n(A, a)$ of a pointed type (A, a) is defined recursively by:

$$\Omega^0(A, a) :\equiv (A, a)$$
$$\Omega^{n+1}(A, a) :\equiv \Omega^n(\Omega(A, a)).$$

An element of it will be called an **n-loop** or an **n-dimensional loop** at a.

2.2 Functions are functors

Lemma 2.2.1. *Suppose that $f : A \to B$ is a function. Then for any $x, y : A$ there is an operation*

$$\mathsf{ap}_f : (x =_A y) \to (f(x) =_B f(y)).$$

Moreover, for each $x : A$ we have $\mathsf{ap}_f(\mathsf{refl}_x) \equiv \mathsf{refl}_{f(x)}$.

The notation ap_f can be read either as the application of f to a path, or as the action on paths of f.
We note that ap behaves functorially, in all the ways that one might expect.

Lemma 2.2.2. *For functions $f : A \to B$ and $g : B \to C$ and paths $p : x =_A y$ and $q : y =_A z$, we have:*

(i) $\mathsf{ap}_f(p \cdot q) = \mathsf{ap}_f(p) \cdot \mathsf{ap}_f(q)$.
(ii) $\mathsf{ap}_f(p^{-1}) = \mathsf{ap}_f(p)^{-1}$.
(iii) $\mathsf{ap}_g(\mathsf{ap}_f(p)) = \mathsf{ap}_{g \circ f}(p)$.
(iv) $\mathsf{ap}_{\mathsf{id}_A}(p) = p$.

2.3 Type families are fibrations

Lemma 2.3.1 (Transport)**.** *Suppose that P is a type family over A and that $p : x =_A y$. Then there is a function $p_* : P(x) \to P(y)$.*

Sometimes, it is necessary to notate the type family P in which the transport operation happens.

$$\mathsf{transport}^P(p, -) : P(x) \to P(y).$$

Lemma 2.3.2 (Path lifting property)**.** *Let $P : A \to \mathcal{U}$ be a type family over A and assume we have $u : P(x)$ for some $x : A$. Then for any $p : x = y$, we have*

$$\mathsf{lift}(u, p) : (x, u) = (y, p_*(u))$$

in $\sum_{(x:A)} P(x)$, such that $\mathsf{pr}_1(\mathsf{lift}(u, p)) = p$.

Remark 2.3.3. Although we may think of a type family $P : A \to \mathcal{U}$ as like a fibration, it is generally not a good idea to say things like "the fibration $P : A \to \mathcal{U}$", since this sounds like we are talking about a fibration with base $\mathcal{U}$ and total space A. To repeat, when a type family $P : A \to \mathcal{U}$ is regarded as a fibration, the base is A and the total space is $\sum_{(x:A)} P(x)$. We may also occasionally use other topological terminology when speaking about type families. For instance, we may refer to a dependent function $f : \prod_{(x:A)} P(x)$ as a **section** of the fibration P, and we may say that something happens **fiberwise** if it happens for each $P(x)$. For instance, a section $f : \prod_{(x:A)} P(x)$ shows that P is "fiberwise inhabited".

Lemma 2.3.4 (Dependent map)**.** *Suppose $f : \prod_{(x:A)} P(x)$; then we have a map*

$$\mathsf{apd}_f : \prod_{p:x=y} (p_*(f(x)) =_{P(y)} f(y)).$$

Lemma 2.3.5. *If $P : A \to \mathcal{U}$ is defined by $P(x) :\equiv B$ for a fixed $B : \mathcal{U}$, then for any $x, y : A$ and $p : x = y$ and $b : B$ we have a path*

$$\mathsf{transportconst}^B_p(b) : \mathsf{transport}^P(p, b) = b.$$

Lemma 2.3.6. *For $f : A \to B$ and $p : x =_A y$, we have*

$$\mathsf{apd}_f(p) = \mathsf{transportconst}^B_p(f(x)) \cdot \mathsf{ap}_f(p).$$

Lemma 2.3.7. *Given $P : A \to \mathcal{U}$ with $p : x =_A y$ and $q : y =_A z$ while $u : P(x)$, we have*

$$q_*(p_*(u)) = (p \cdot q)_*(u).$$

Lemma 2.3.8. *For a function $f : A \to B$ and a type family $P : B \to \mathcal{U}$, and any $p : x =_A y$ and $u : P(f(x))$, we have*

$$\mathsf{transport}^{P \circ f}(p, u) = \mathsf{transport}^P(\mathsf{ap}_f(p), u).$$

Lemma 2.3.9. *For $P, Q : A \to \mathcal{U}$ and a family of functions $f : \prod_{(x:A)} P(x) \to Q(x)$, and any $p : x =_A y$ and $u : P(x)$, we have*

$$\mathsf{transport}^Q(p, f_x(u)) = f_y(\mathsf{transport}^P(p, u)).$$

2.4 Homotopies and equivalences

Definition 2.4.1. Let $f, g : \prod_{(x:A)} P(x)$ be two sections of a type family $P : A \to \mathcal{U}$. A **homotopy** from f to g is a dependent function of type

$$(f \sim g) :\equiv \prod_{x:A} (f(x) = g(x)).$$

Note that a homotopy is not the same as an identification $(f = g)$. However, in §2.9 we will introduce an axiom making homotopies and identifications "equivalent".
The following proofs are left to the reader.

Lemma 2.4.2. *Homotopy is an equivalence relation on each dependent function type $\prod_{(x:A)} P(x)$. That is, we have elements of the types*

$$\prod_{f:\prod_{(x:A)} P(x)} (f \sim f)$$
$$\prod_{f,g:\prod_{(x:A)} P(x)} (f \sim g) \to (g \sim f)$$
$$\prod_{f,g,h:\prod_{(x:A)} P(x)} (f \sim g) \to (g \sim h) \to (f \sim h).$$

Lemma 2.4.3. *Suppose $H : f \sim g$ is a homotopy between functions $f, g : A \to B$ and let $p : x =_A y$. Then we have*

$$H(x) \cdot g(p) = f(p) \cdot H(y).$$

We may also draw this as a commutative diagram:

$$\begin{array}{ccc} f(x) & \xrightarrow{f(p)} & f(y) \\ {\scriptstyle H(x)}\downarrow & & \downarrow{\scriptstyle H(y)} \\ g(x) & \xrightarrow[g(p)]{} & g(y) \end{array}$$

Corollary 2.4.4. *Let $H : f \sim \mathsf{id}_A$ be a homotopy, with $f : A \to A$. Then for any $x : A$ we have*

$$H(f(x)) = f(H(x)).$$

$$\sum_{g:B\to A} \big((f\circ g \sim \mathsf{id}_B) \times (g\circ f \sim \mathsf{id}_A)\big) \tag{2.4.5}$$

Definition 2.4.6. For a function $f : A \to B$, a **quasi-inverse** of f is a triple (g, α, β) consisting of a function $g : B \to A$ and homotopies $\alpha : f \circ g \sim \mathsf{id}_B$ and $\beta : g \circ f \sim \mathsf{id}_A$.

Thus, (2.4.5) is *the type of quasi-inverses of* f; we may denote it by $\mathsf{qinv}(f)$.

Example 2.4.7. For any $p : x =_A y$ and $z : A$, the functions

$$(p \cdot -) : (y =_A z) \to (x =_A z) \qquad \text{and}$$
$$(- \cdot p) : (z =_A x) \to (z =_A y)$$

have quasi-inverses given by $(p^{-1} \cdot -)$ and $(- \cdot p^{-1})$, respectively;

Example 2.4.8. For any $p : x =_A y$ and $P : A \to \mathcal{U}$, the function

$$\mathsf{transport}^P(p, -) : P(x) \to P(y)$$

has a quasi-inverse given by $\mathsf{transport}^P(p^{-1}, -)$; this follows from Lemma 2.3.7.

$$(A \simeq B) :\equiv \sum_{f:A\to B} \mathsf{isequiv}(f). \tag{2.4.9}$$

Lemma 2.4.10. *Type equivalence is an equivalence relation on* $\mathcal{U}$. *More specifically:*

(i) For any A, the identity function id_A is an equivalence; hence $A \simeq A$.
(ii) For any $f : A \simeq B$, we have an equivalence $f^{-1} : B \simeq A$.
(iii) For any $f : A \simeq B$ and $g : B \simeq C$, we have $g \circ f : A \simeq C$.

2.5 The higher groupoid structure of type formers

2.6 Cartesian product types

$$(x =_{A\times B} y) \to (\mathsf{pr}_1(x) =_A \mathsf{pr}_1(y)) \times (\mathsf{pr}_2(x) =_B \mathsf{pr}_2(y)). \tag{2.6.1}$$

Theorem 2.6.2. *For any x and y, the function* (2.6.1) *is an equivalence.*

Theorem 2.6.3. *In the above situation, we have*

$$\mathsf{transport}^{A\times B}(p, x) =_{A(w)\times B(w)} (\mathsf{transport}^A(p, \mathsf{pr}_1 x), \mathsf{transport}^B(p, \mathsf{pr}_2 x)).$$

Theorem 2.6.4. *In the above situation, given $x, y : A \times B$ and $p : \mathsf{pr}_1 x = \mathsf{pr}_1 y$ and $q : \mathsf{pr}_2 x = \mathsf{pr}_2 y$, we have*

$$f(\mathsf{pair}^{=}(p, q)) =_{(f(x)=f(y))} \mathsf{pair}^{=}(g(p), h(q)).$$

2.7 Σ-types

Theorem 2.7.1. *Suppose that $P : A \to \mathcal{U}$ is a type family over a type A and let $w, w' : \sum_{(x:A)} P(x)$. Then there is an equivalence*

$$(w = w') \;\simeq\; \sum_{(p:\mathsf{pr}_1(w)=\mathsf{pr}_1(w'))} p_*(\mathsf{pr}_2(w)) = \mathsf{pr}_2(w').$$

Corollary 2.7.2. *For $z : \sum_{(x:A)} P(x)$, we have $z = (\mathsf{pr}_1(z), \mathsf{pr}_2(z))$.*

Theorem 2.7.3. *Suppose we have type families*

$$P : A \to \mathcal{U} \qquad \text{and} \qquad Q : \Big(\sum_{x:A} P(x)\Big) \to \mathcal{U}.$$

Then we can construct the type family over A defined by

$$x \mapsto \sum_{u:P(x)} Q(x, u).$$

For any path $p : x = y$ and any $(u, z) : \sum_{(u:P(x))} Q(x, u)$ we have

$$p_*(u, z) = \big(p_*(u),\, \mathsf{pair}^{=}(p, \mathsf{refl}_{p_*(u)})_*(z)\big).$$

2.8 The unit type

Theorem 2.8.1. *For any $x, y : \mathbf{1}$, we have $(x = y) \simeq \mathbf{1}$.*

2.9 Π-types and the function extensionality axiom

$$\mathsf{happly} : (f = g) \to \prod_{x:A} (f(x) =_{B(x)} g(x)) \tag{2.9.1}$$

Axiom 2.9.2 (Function extensionality)**.** *For any A, B, f, and g, the function* (2.9.1) *is an equivalence.*

In particular, Axiom 2.9.2 implies that (2.9.1) has a quasi-inverse

$$\mathsf{funext} : \Big(\prod_{x:A} (f(x) = g(x))\Big) \to (f = g).$$

This function is also referred to as "function extensionality".

$$\mathsf{refl}_f = \mathsf{funext}(x \mapsto \mathsf{refl}_{f(x)})$$
$$\alpha^{-1} = \mathsf{funext}(x \mapsto \mathsf{happly}(\alpha, x)^{-1})$$
$$\alpha \cdot \beta = \mathsf{funext}(x \mapsto \mathsf{happly}(\alpha, x) \cdot \mathsf{happly}(\beta, x)).$$

Given a type X, a path $p : x_1 =_X x_2$, type families $A, B : X \to \mathcal{U}$, and a function $f : A(x_1) \to B(x_1)$, we have

$$\mathsf{transport}^{A\to B}(p, f) = \Big(x \mapsto \mathsf{transport}^B(p, f(\mathsf{transport}^A(p^{-1}, x)))\Big) \tag{2.9.3}$$

where $A \to B$ denotes abusively the type family $X \to \mathcal{U}$ defined by

$$(A \to B)(x) :\equiv (A(x) \to B(x)).$$

Transporting dependent functions is similar, but more complicated. Suppose given X and p as before, type families $A : X \to \mathcal{U}$ and $B : \prod_{(x:X)}(A(x) \to \mathcal{U})$, and also a dependent function $f : \prod_{(a:A(x_1))} B(x_1, a)$. Then for $a : A(x_2)$, we have

$$\mathsf{transport}^{\Pi_A(B)}(p, f)(a) = \mathsf{transport}^{\widehat{B}}\Big((\mathsf{pair}^{=}(p^{-1}, \mathsf{refl}_{p^{-1}{}_*(a)}))^{-1},\, f(\mathsf{transport}^A($$

where $\Pi_A(B)$ and $\widehat{B}$ denote respectively the type families

$$\begin{aligned} \Pi_A(B) &:\equiv \big(x \mapsto \textstyle\prod_{(a:A(x))} B(x, a)\big) &&: X \to \mathcal{U} \\ \widehat{B} &:\equiv \big(w \mapsto B(\mathsf{pr}_1 w, \mathsf{pr}_2 w)\big) &&: \big(\textstyle\sum_{(x:X)} A(x)\big) \to \mathcal{U}. \end{aligned} \tag{2.9.4}$$

Lemma 2.9.5. *Given type families $A, B : X \to \mathcal{U}$ and $p : x =_X y$, and also $f : A(x) \to B(x)$ and $g : A(y) \to B(y)$, we have an equivalence*

$$\big(p_*(f) = g\big) \;\simeq\; \prod_{a:A(x)} (p_*(f(a)) = g(p_*(a))).$$

Moreover, if $q : p_*(f) = g$ corresponds under this equivalence to $\widehat{q}$, then for $a : A(x)$, the path

$$\mathsf{happly}(q, p_*(a)) : (p_*(f))(p_*(a)) = g(p_*(a))$$

is equal to the concatenated path $i \cdot j \cdot k$, where

- *$i : (p_*(f))(p_*(a)) = p_*\big(f(p^{-1}{}_*(p_*(a)))\big)$ comes from* (2.9.3),
- *$j : p_*\big(f(p^{-1}{}_*(p_*(a)))\big) = p_*(f(a))$ comes from Lemmas 2.1.3 and 2.3.7, and*
- *$k : p_*(f(a)) = g(p_*(a))$ is $\widehat{q}(a)$.*

Lemma 2.9.6. *Given type families $A : X \to \mathcal{U}$ and $B : \prod_{(x:X)} A(x) \to \mathcal{U}$ and $p : x =_X y$, and also $f : \prod_{(a:A(x))} B(x, a)$ and $g : \prod_{(a:A(y))} B(y, a)$, we have an equivalence*

$$\big(p_*(f) = g\big) \;\simeq\; \Big(\prod_{a:A(x)} \mathsf{transport}^{\widehat{B}}(\mathsf{pair}^{=}(p, \mathsf{refl}_{p_*(a)}), f(a)) = g(p_*(a))\Big)$$

with $\widehat{B}$ as in (2.9.4).

2.10 Universes and the univalence axiom

Lemma 2.10.1. *For types $A, B : \mathcal{U}$, there is a certain function,*

$$\mathsf{idtoeqv} : (A =_{\mathcal{U}} B) \to (A \simeq B), \tag{2.10.2}$$

defined in the proof.

Axiom 2.10.3 (Univalence)**.** *For any $A, B : \mathcal{U}$, the function* (2.10.2) *is an equivalence.*

- An introduction rule for $(A =_{\mathcal{U}} B)$, denoted ua for "univalence axiom":
$$\mathsf{ua} : (A \simeq B) \to (A =_{\mathcal{U}} B).$$
- The elimination rule, which is $\mathsf{idtoeqv}$,
$$\mathsf{idtoeqv} \equiv \mathsf{transport}^{X\mapsto X} : (A =_{\mathcal{U}} B) \to (A \simeq B).$$

- The propositional computation rule,

$$\mathsf{transport}^{X\mapsto X}(\mathsf{ua}(f), x) = f(x).$$

- The propositional uniqueness principle: for any $p : A = B$,

$$p = \mathsf{ua}(\mathsf{transport}^{X\mapsto X}(p)).$$

We can also identify the reflexivity, concatenation, and inverses of equalities in the universe with the corresponding operations on equivalences:

$$\begin{aligned} \mathsf{refl}_A &= \mathsf{ua}(\mathsf{id}_A) \\ \mathsf{ua}(f) \cdot \mathsf{ua}(g) &= \mathsf{ua}(g \circ f) \\ \mathsf{ua}(f)^{-1} &= \mathsf{ua}(f^{-1}). \end{aligned}$$

Lemma 2.10.4. *For any type family $B : A \to \mathcal{U}$ and $x, y : A$ with a path $p : x = y$ and $u : B(x)$, we have*

$$\begin{aligned} \mathsf{transport}^{B}(p, u) &= \mathsf{transport}^{X\mapsto X}(\mathsf{ap}_B(p), u) \\ &= \mathsf{idtoeqv}(\mathsf{ap}_B(p))(u). \end{aligned}$$

2.11 Identity type

Theorem 2.11.1. *If $f : A \to B$ is an equivalence, then for all $a, a' : A$, so is*

$$\mathsf{ap}_f : (a =_A a') \to (f(a) =_B f(a')).$$

Lemma 2.11.2. *For any A and $a : A$, with $p : x_1 = x_2$, we have*

$$\begin{aligned} &\mathsf{transport}^{x\mapsto(a=x)}(p, q) = q \cdot p && \text{for } q : a = x_1, \\ &\mathsf{transport}^{x\mapsto(x=a)}(p, q) = p^{-1} \cdot q && \text{for } q : x_1 = a, \\ &\mathsf{transport}^{x\mapsto(x=x)}(p, q) = p^{-1} \cdot q \cdot p && \text{for } q : x_1 = x_1. \end{aligned}$$

Theorem 2.11.3. *For $f, g : A \to B$, with $p : a =_A a'$ and $q : f(a) =_B g(a)$, we have*

$$\mathsf{transport}^{x\mapsto f(x)=_B g(x)}(p, q) =_{f(a')=g(a')} (\mathsf{ap}_f p)^{-1} \cdot q \cdot \mathsf{ap}_g p.$$

Theorem 2.11.4. *Let $B : A \to \mathcal{U}$ and $f, g : \prod_{(x:A)} B(x)$, with $p : a =_A a'$ and $q : f(a) =_{B(a)} g(a)$. Then we have*

$$\mathsf{transport}^{x\mapsto f(x)=_{B(x)} g(x)}(p, q) = (\mathsf{apd}_f(p))^{-1} \cdot \mathsf{ap}_{(\mathsf{transport}^B p)}(q) \cdot \mathsf{apd}_g(p).$$

Theorem 2.11.5. *For $p : a =_A a'$ with $q : a = a$ and $r : a' = a'$, we have*

$$\big(\mathsf{transport}^{x\mapsto(x=x)}(p, q) = r\big) \;\simeq\; \big(q \cdot p = p \cdot r\big).$$

2.12 Coproducts

Theorem 2.12.1. *For all $x : A + B$ we have $(\mathsf{inl}(a_0) = x) \simeq \mathsf{code}(x)$.*

2.13 Natural numbers

We use the encode-decode method to characterize the path space of the natural numbers, which are also a positive type.

$$\mathsf{code} : \mathbb{N} \to \mathbb{N} \to \mathcal{U},$$

defined by double recursion over $\mathbb{N}$ as follows:

$$\begin{aligned} \mathsf{code}(0, 0) &:\equiv \mathbf{1} \\ \mathsf{code}(\mathsf{succ}(m), 0) &:\equiv \mathbf{0} \\ \mathsf{code}(0, \mathsf{succ}(n)) &:\equiv \mathbf{0} \\ \mathsf{code}(\mathsf{succ}(m), \mathsf{succ}(n)) &:\equiv \mathsf{code}(m, n). \end{aligned}$$

We also define by recursion a dependent function $r : \prod_{(n:\mathbb{N})} \mathsf{code}(n, n)$, with

$$\begin{aligned} r(0) &:\equiv \star \\ r(\mathsf{succ}(n)) &:\equiv r(n). \end{aligned}$$

Theorem 2.13.1. *For all $m, n : \mathbb{N}$ we have $(m = n) \simeq \mathsf{code}(m, n)$.*

2.14 Example: equality of structures

Definition 2.14.1. Given a type A, the type SemigroupStr(A) of **semigroup structures** with carrier A is defined by

$$\mathsf{SemigroupStr}(A) :\equiv \sum_{(m:A\to A\to A)} \prod_{(x,y,z:A)} m(x, m(y, z)) = m(m(x, y), z).$$

A **semigroup** is a type together with such a structure:

$$\mathsf{Semigroup} :\equiv \sum_{A:\mathcal{U}} \mathsf{SemigroupStr}(A)$$

2.14.1 Lifting equivalences

$$\mathsf{transport}^{\mathsf{SemigroupStr}}(\mathsf{ua}(e)) : \mathsf{SemigroupStr}(A) \to \mathsf{SemigroupStr}(B).$$

Moreover, this map is an equivalence, because $\mathsf{transport}^C(\alpha)$ is always an equivalence with inverse $\mathsf{transport}^C(\alpha^{-1})$, see Lemmas 2.1.3 and 2.3.7.

2.15 Universal properties

$$(X \to A \times B) \to (X \to A) \times (X \to B) \tag{2.15.1}$$

defined by $f \mapsto (\mathsf{pr}_1 \circ f, \mathsf{pr}_2 \circ f)$.

Theorem 2.15.2. (2.15.1) *is an equivalence.*

$$\Big(\prod_{x:X} (A(x) \times B(x))\Big) \to \Big(\prod_{x:X} A(x)\Big) \times \Big(\prod_{x:X} B(x)\Big) \tag{2.15.3}$$

defined as before by $f \mapsto (\mathsf{pr}_1 \circ f, \mathsf{pr}_2 \circ f)$.

Theorem 2.15.4. (2.15.3) *is an equivalence.*

$$\Big(\prod_{x:X} \sum_{(a:A(x))} P(x, a)\Big) \to \Big(\sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x, g(x))\Big). \tag{2.15.5}$$

Theorem 2.15.6. (2.15.5) *is an equivalence.*

Homotopy Type Theory

Sets and logic

3.16 Sets and n-types

Definition 3.16.1. A type A is a **set** if for all $x, y : A$ and all $p, q : x = y$, we have $p = q$.

More precisely, the proposition $\mathsf{isSet}(A)$ is defined to be the type

$$\mathsf{isSet}(A) :\equiv \prod_{(x,y:A)} \prod_{(p,q:x=y)} (p = q).$$

Example 3.16.2. The type $\mathbf{1}$ is a set. For any $x, y : \mathbf{1}$ the type $(x = y)$ is equivalent to $\mathbf{1}$. Since any two elements of $\mathbf{1}$ are equal, this implies that any two elements of $x = y$ are equal.

Example 3.16.3. The type $\mathbf{0}$ is a set, for given any $x, y : \mathbf{0}$ we may deduce anything we like, by the induction principle of $\mathbf{0}$.

Example 3.16.4. The type $\mathbb{N}$ of natural numbers is also a set. Since all equality types $x =_{\mathbb{N}} y$ are equivalent to either $\mathbf{1}$ or $\mathbf{0}$, and any two inhabitants of $\mathbf{1}$ or $\mathbf{0}$ are equal.

Most of the type forming operations we have considered so far also preserve sets.

Example 3.16.5. If A and B are sets, then so is $A \times B$. For given $x, y : A \times B$ and $p, q : x = y$, then we have $p = \mathsf{pair}^{=}(\mathsf{ap}_{\mathsf{pr}_1}(p), \mathsf{ap}_{\mathsf{pr}_2}(p))$ and $q = \mathsf{pair}^{=}(\mathsf{ap}_{\mathsf{pr}_1}(q), \mathsf{ap}_{\mathsf{pr}_2}(q))$. But $\mathsf{ap}_{\mathsf{pr}_1}(p) = \mathsf{ap}_{\mathsf{pr}_1}(q)$ since A is a set, and $\mathsf{ap}_{\mathsf{pr}_2}(p) = \mathsf{ap}_{\mathsf{pr}_2}(q)$ since B is a set; hence $p = q$.
Similarly, if A is a set and $B : A \to \mathcal{U}$ is such that each $B(x)$ is a set, then $\sum_{(x:A)} B(x)$ is a set.

Example 3.16.6. If A is *any* type and $B : A \to \mathcal{U}$ is such that each $B(x)$ is a set, then the type $\prod_{(x:A)} B(x)$ is a set. For suppose $f, g : \prod_{(x:A)} B(x)$ and $p, q : f = g$. By function extensionality, we have

$$p = \mathsf{funext}(x \mapsto \mathsf{happly}(p, x)) \quad \text{and} \quad q = \mathsf{funext}(x \mapsto \mathsf{happly}(q, x)).$$

But for any $x : A$, we have

$$\mathsf{happly}(p, x) : f(x) = g(x) \qquad \text{and} \qquad \mathsf{happly}(q, x) : f(x) = g(x),$$

so since $B(x)$ is a set we have $\mathsf{happly}(p, x) = \mathsf{happly}(q, x)$. Now using function extensionality again, the dependent functions $(x \mapsto \mathsf{happly}(p, x))$ and $(x \mapsto \mathsf{happly}(q, x))$ are equal, and hence (applying $\mathsf{ap}_{\mathsf{funext}}$) so are p and q.

Definition 3.16.7. A type A is a **1-type** if for all $x, y : A$ and $p, q : x = y$ and $r, s : p = q$, we have $r = s$.

Lemma 3.16.8. *If A is a set (that is, $\mathsf{isSet}(A)$ is inhabited), then A is a 1-type.*

3.17 Propositions as types?

Remark 3.17.1. (Statement) If for all $x : X$ there exists an $a : A(x)$ such that $P(x, a)$, then there exists a function $g : \prod_{(x:A)} A(x)$ such that for all $x : X$ we have $P(x, g(x))$".
This looks like the classical *axiom of choice*, is always true under this reading.

Remark 3.17.2. The classical *law of double negation* and *law of excluded middle* are incompatible with the univalence axiom.

Theorem 3.17.3. *It is not the case that for all $A : \mathcal{U}$ we have $\neg(\neg A) \to A$.*

Remark 3.17.4. For any A, $\neg\neg\neg A \to \neg A$ for any A.

Corollary 3.17.5. *It is not the case that for all $A : \mathcal{U}$ we have $A + (\neg A)$.*

3.18 Mere propositions

Definition 3.18.1. A type P is a **mere proposition** if for all $x, y : P$ we have $x = y$.

Specifically, for any $P : \mathcal{U}$, the type $\mathsf{isProp}(P)$ is defined to be

$$\mathsf{isProp}(P) :\equiv \prod_{x,y:P} (x = y).$$

Lemma 3.18.2. *If P is a mere proposition and $x_0 : P$, then $P \simeq \mathbf{1}$.*

Lemma 3.18.3. *If P and Q are mere propositions such that $P \to Q$ and $Q \to P$, then $P \simeq Q$.*

Lemma 3.18.4. *Every mere proposition is a set.*

Lemma 3.18.5. *For any type A, the types $\mathsf{isProp}(A)$ and $\mathsf{isSet}(A)$ are mere propositions.*

3.19 Classical vs. intuitionistic logic

With the notion of mere proposition in hand, we can now give the proper formulation of the **law of excluded middle** in homotopy type theory:

$$\mathsf{LEM} :\equiv \prod_{A:\mathcal{U}} \Big(\mathsf{isProp}(A) \to (A + \neg A)\Big). \tag{3.19.1}$$

Similarly, the **law of double negation** is

$$\prod_{A:\mathcal{U}} \Big(\mathsf{isProp}(A) \to (\neg\neg A \to A)\Big). \tag{3.19.2}$$

Definition 3.19.3.

(i) A type A is called **decidable** if $A + \neg A$.
(ii) Similarly, a type family $B : A \to \mathcal{U}$ is **decidable** if $\prod_{(a:A)} (B(a) + \neg B(a))$.
(iii) In particular, A has **decidable equality** if $\prod_{(a,b:A)} ((a = b) + \neg(a = b))$.

3.20 Subsets and propositional resizing

Lemma 3.20.1. *Suppose $P : A \to \mathcal{U}$ is a type family such that $P(x)$ is a mere proposition for all $x : A$. If $u, v : \sum_{(x:A)} P(x)$ are such that $\mathsf{pr}_1(u) = \mathsf{pr}_1(v)$, then $u = v$.*

For instance, recall that

$$(A \simeq B) :\equiv \sum_{f:A\to B} \mathsf{isequiv}(f),$$

where each type $\mathsf{isequiv}(f)$ was supposed to be a mere proposition. It follows that if two equivalences have equal underlying functions, then they are equal as equivalences.
If $P : A \to \mathcal{U}$ is a family of mere propositions (i.e. each $P(x)$ is a mere proposition), we may write

$$\{\, x : A \mid P(x) \,\} \tag{3.20.2}$$

as an alternative notation for $\sum_{(x:A)} P(x)$. We may define the "subuniverses" of sets and of mere propositions in a universe $\mathcal{U}$:

$$\mathsf{Set}_{\mathcal{U}} :\equiv \{\, A : \mathcal{U} \mid \mathsf{isSet}(A) \,\},$$
$$\mathsf{Prop}_{\mathcal{U}} :\equiv \{\, A : \mathcal{U} \mid \mathsf{isProp}(A) \,\}.$$

An element of $\mathsf{Set}_{\mathcal{U}}$ is a type $A : \mathcal{U}$ together with evidence $s : \mathsf{isSet}(A)$, and similarly for $\mathsf{Prop}_{\mathcal{U}}$.

Axiom 3.20.3 (Propositional resizing)**.** *The map $\mathsf{Prop}_{\mathcal{U}_i} \to \mathsf{Prop}_{\mathcal{U}_{i+1}}$ is an equivalence.*

With propositional resizing, we can define the power set to be

$$\mathcal{P}(A) :\equiv (A \to \Omega),$$

which is then independent of $\mathcal{U}$.

3.21 The logic of mere propositions

Example 3.21.1. If A and B are mere propositions, so is $A \times B$. This is easy to show using the characterization of paths in products, just like Example 3.16.5 but simpler. Thus, the connective "and" preserves mere propositions.

Example 3.21.2. If A is any type and $B : A \to \mathcal{U}$ is such that for all $x : A$, the type $B(x)$ is a mere proposition, then $\prod_{(x:A)} B(x)$ is a mere proposition. The proof is just like Example 3.16.6 but simpler: given $f, g : \prod_{(x:A)} B(x)$, for any $x : A$ we have $f(x) = g(x)$ since $B(x)$ is a mere proposition. But then by function extensionality, we have $f = g$.
In particular, if B is a mere proposition, then so is $A \to B$ regardless of what A is. In even more particular, since $\mathbf{0}$ is a mere proposition, so is $\neg A \equiv (A \to \mathbf{0})$. Thus, the connectives "implies" and "not" preserve mere propositions, as does the quantifier "for all".

3.22 Propositional truncation

The *propositional truncation*, also called the (-1)-*truncation*, *bracket type*, or *squash type*, is an additional type former which "squashes" or "truncates" a type down to a mere proposition, forgetting all information contained in inhabitants of that type other than their existence.
More precisely, for any type A, there is a type $\|A\|$. It has two constructors:

- For any $a : A$ we have $|a| : \|A\|$.
- For any $x, y : \|A\|$, we have $x = y$.

The recursion principle of $\|A\|$ says that:

- If B is a mere proposition and we have $f : A \to B$, then there is an induced $g : \|A\| \to B$ such that $g(|a|) \equiv f(a)$ for all $a : A$.

Definition 3.22.1. We define **traditional logical notation** using truncation as follows, where P and Q denote mere propositions (or families thereof):

$$\begin{aligned}
\top &:\equiv \mathbf{1} \\
\bot &:\equiv \mathbf{0} \\
P \wedge Q &:\equiv P \times Q \\
P \Rightarrow Q &:\equiv P \to Q \\
P \Leftrightarrow Q &:\equiv P = Q \\
\neg P &:\equiv P \to \mathbf{0} \\
P \vee Q &:\equiv \|P + Q\| \\
\forall(x : A).\, P(x) &:\equiv \prod_{x:A} P(x) \\
\exists(x : A).\, P(x) &:\equiv \Big\| \sum_{x:A} P(x) \Big\|
\end{aligned}$$

The notations $\wedge$ and $\vee$ are also used in homotopy theory for the smash product and the wedge of pointed spaces.

$$\begin{aligned}
\{\, x : A \mid P(x) \,\} \cap \{\, x : A \mid Q(x) \,\} &:\equiv \{\, x : A \mid P(x) \wedge Q(x) \,\}, \\
\{\, x : A \mid P(x) \,\} \cup \{\, x : A \mid Q(x) \,\} &:\equiv \{\, x : A \mid P(x) \vee Q(x) \,\}, \\
A \setminus \{\, x : A \mid P(x) \,\} &:\equiv \{\, x : A \mid \neg P(x) \,\}.
\end{aligned}$$

Of course, in the absence of LEM, the latter are not "complements" in the usual sense: we may not have $B \cup (A \setminus B) = A$ for every subset B of A.

3.23 The axiom of choice

$$A : X \to \mathcal{U} \qquad \text{and} \qquad P : \prod_{x:X} A(x) \to \mathcal{U},$$

and moreover that

- X is a set,
- $A(x)$ is a set for all $x : X$, and
- $P(x, a)$ is a mere proposition for all $x : X$ and $a : A(x)$.

The **axiom of choice** AC asserts that under these assumptions,

$$\Big(\prod_{x:X} \Big\| \sum_{a:A(x)} P(x,a) \Big\|\Big) \to \Big\| \sum_{(g:\prod_{(x:X)} A(x))} \prod_{(x:X)} P(x, g(x)) \Big\|. \tag{3.23.1}$$

Of course, this is a direct translation of (3.17.1) where we read "there exists $x : A$ such that $B(x)$" as $\left\|\sum_{(x:A)} B(x)\right\|$, so we could have written the statement in the familiar logical notation as

$$\Big(\forall(x : X).\, \exists(a : A(x)).\, P(x,a)\Big) \Rightarrow \Big(\exists(g : \textstyle\prod_{(x:X)} A(x)).\, \forall(x : X).\, P(x, g(x))\Big).$$

Lemma 3.23.2. *The axiom of choice* (3.23.1) *is equivalent to the statement that for any set X and any $Y : X \to \mathcal{U}$ such that each $Y(x)$ is a set, we have*

$$\Big(\prod_{x:X} \Big\| Y(x) \Big\|\Big) \to \Big\| \prod_{x:X} Y(x) \Big\|. \tag{3.23.3}$$

Remark 3.23.4. The right side of (3.23.3) always implies the left. Since both are mere propositions, by Lemma 3.18.3 the axiom of choice is also equivalent to asking for an equivalence

$$\Big(\prod_{x:X} \Big\| Y(x) \Big\|\Big) \simeq \Big\| \prod_{x:X} Y(x) \Big\|$$

Lemma 3.23.5. *There exists a type X and a family $Y : X \to \mathcal{U}$ such that each $Y(x)$ is a set, but such that* (3.23.3) *is false.*

3.24 The principle of unique choice

Lemma 3.24.1. *If P is a mere proposition, then $P \simeq \|P\|$.*

Corollary 3.24.2 (The principle of unique choice)**.** *Suppose a type family $P : A \to \mathcal{U}$ such that*

(i) For each x, the type $P(x)$ is a mere proposition, and
(ii) For each x we have $\|P(x)\|$.

Then we have $\prod_{(x:A)} P(x)$.

3.26 Contractibility

Definition 3.26.1. A type A is **contractible**, or a **singleton**, if there is $a : A$, called the **center of contraction**, such that $a = x$ for all $x : A$. We denote the specified path $a = x$ by contr_x.

In other words, the type $\mathsf{isContr}(A)$ is defined to be

$$\mathsf{isContr}(A) :\equiv \sum_{(a:A)} \prod_{(x:A)} (a = x).$$

Lemma 3.26.2. *For a type A, the following are logically equivalent.*

(i) A is contractible in the sense of Definition 3.26.1.
(ii) A is a mere proposition, and there is a point $a : A$.
(iii) A is equivalent to $\mathbf{1}$.

Lemma 3.26.3. *For any type A, the type $\mathsf{isContr}(A)$ is a mere proposition.*

Corollary 3.26.4. *If A is contractible, then so is $\mathsf{isContr}(A)$.*

Lemma 3.26.5. *If $P : A \to \mathcal{U}$ is a type family such that each $P(a)$ is contractible, then $\prod_{(x:A)} P(x)$ is contractible.*

Of course, if A is equivalent to B and A is contractible, then so is B. More generally, it suffices for B to be a *retract* of A. By definition, a **retraction** is a function $r : A \to B$ such that there exists a function $s : B \to A$, called its **section**, and a homotopy $\epsilon : \prod_{(y:B)} (r(s(y)) = y)$; then we say that B is a **retract**of A.

Lemma 3.26.6. *If B is a retract of A, and A is contractible, then so is B.*

Lemma 3.26.7. *For any A and any $a : A$, the type $\sum_{(x:A)} (a = x)$ is contractible.*

Lemma 3.26.8. *Let $P : A \to \mathcal{U}$ be a type family.*

(i) If each $P(x)$ is contractible, then $\sum_{(x:A)} P(x)$ is equivalent to A.
(ii) If A is contractible with center a, then $\sum_{(x:A)} P(x)$ is equivalent to $P(a)$.

Lemma 3.26.9. *A type A is a mere proposition if and only if for all $x, y : A$, the type $x =_A y$ is contractible.*

Homotopy Type Theory

Formal type theory

3.27 The first presentation

Convertibility $t \downarrow t'$ between terms t and t' is the equivalence relation generated by the defining equations for constants, the computation rule

3.27.1 Type universes

We postulate a hierarchy of **universes** denoted by primitive constants

$$\mathcal{U}_0, \quad \mathcal{U}_1, \quad \mathcal{U}_2, \quad \dots$$

The first two rules for universes say that they form a cumulative hierarchy of types:

- $\mathcal{U}_m : \mathcal{U}_n$ for $m < n$,
- if $A : \mathcal{U}_m$ and $m \leq n$, then $A : \mathcal{U}_n$,

and the third expresses the idea that an object of a universe can serve as a type and stand to the right of a colon in judgments:

- if $\Gamma \vdash A : \mathcal{U}_n$, and x is a new variable,[1] then $\vdash (\Gamma, x : A)\ \mathsf{ctx}$.

In the body of the book, an equality judgment $A \equiv B : \mathcal{U}_n$ between types A and B is usually abbreviated to $A \equiv B$. This is an instance of typical ambiguity, as we can always switch to a larger universe, which however does not affect the validity of the judgment.
The following conversion rule allows us to replace a type by one equal to it in a typing judgment:

- if $a : A$ and $A \equiv B$ then $a : B$.

3.27.2 Dependent function types (Π-types)

We introduce a primitive constant c_Π, but write $c_\Pi(A, \lambda x.\, B)$ as $\prod_{(x:A)} B$. Judgments concerning such expressions and expressions of the form $\lambda x.\, b$ are introduced by the following rules:

- if $\Gamma \vdash A : \mathcal{U}_n$ and $\Gamma, x : A \vdash B : \mathcal{U}_n$, then $\Gamma \vdash \prod_{(x:A)} B : \mathcal{U}_n$
- if $\Gamma, x : A \vdash b : B$ then $\Gamma \vdash (\lambda x.\, b) : (\prod_{(x:A)} B)$
- if $\Gamma \vdash g : \prod_{(x:A)} B$ and $\Gamma \vdash t : A$ then $\Gamma \vdash g(t) : B[t/x]$

If x does not occur freely in B, we abbreviate $\prod_{(x:A)} B$ as the non-dependent function type $A \to B$ and derive the following rule:

- if $\Gamma \vdash g : A \to B$ and $\Gamma \vdash t : A$ then $\Gamma \vdash g(t) : B$

Using non-dependent function types and leaving implicit the context Γ, the rules above can be written in the following alternative style that we use in the rest of this section of the appendix:

- if $A : \mathcal{U}_n$ and $B : A \to \mathcal{U}_n$, then $\prod_{(x:A)} B(x) : \mathcal{U}_n$
- if $x : A \vdash b : B$ then $\lambda x.\, b : \prod_{(x:A)} B(x)$
- if $g : \prod_{(x:A)} B(x)$ and $t : A$ then $g(t) : B(t)$

[1] By "new" we mean that it does not appear in Γ or A.

3.27.3 Dependent pair types (Σ-types)

We introduce primitive constants c_Σ and c_{pair}. An expression of the form $c_\Sigma(A, \lambda a.\, B)$ is written as $\sum_{(a:A)} B$, and an expression of the form $c_{\mathsf{pair}}(a, b)$ is written as (a, b). We write $A \times B$ instead of $\sum_{(x:A)} B$ if x is not free in B. Judgments concerning such expressions are introduced by the following rules:

- if $A : \mathcal{U}_n$ and $B : A \to \mathcal{U}_n$, then $\sum_{(x:A)} B(x) : \mathcal{U}_n$
- if, in addition, $a : A$ and $b : B(a)$, then $(a, b) : \sum_{(x:A)} B(x)$

If we have A and B as above, $C : (\sum_{(x:A)} B(x)) \to \mathcal{U}_m$, and

$$d : \prod_{(x:A)} \prod_{(y:B(x))} C((x, y))$$

we can introduce a defined constant

$$f : \prod_{(p:\sum_{(x:A)} B(x))} C(p)$$

with the defining equation

$$f((x, y)) :\equiv d(x, y).$$

Note that C, d, x, and y may contain extra implicit parameters $x_1, \dots, x_n$ if they were obtained in some non-empty context; therefore, the fully explicit recursion schema is

$$f(x_1, \dots, x_n, (x(x_1, \dots, x_n), y(x_1, \dots, x_n))) :\equiv d(x_1, \dots, x_n, (x(x_1, \dots, x_n), y(x_1, \dots, x_n))).$$

3.27.4 Coproduct types

We introduce primitive constants c_+, c_{inl}, and c_{inr}. We write $A + B$ instead of $c_+(A, B)$, $\mathsf{inl}(a)$ instead of $c_{\mathsf{inl}}(a)$, and $\mathsf{inr}(a)$ instead of $c_{\mathsf{inr}}(a)$:

- if $A, B : \mathcal{U}_n$ then $A + B : \mathcal{U}_n$
- moreover, $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$

If we have A and B as above, $C : A + B \to \mathcal{U}_m$, $d : \prod_{(x:A)} C(\mathsf{inl}(x))$, and $e : \prod_{(y:B)} C(\mathsf{inr}(y))$, then we can introduce a defined constant $f : \prod_{(z:A+B)} C(z)$ with the defining equations

$$f(\mathsf{inl}(x)) :\equiv d(x) \qquad \text{and} \qquad f(\mathsf{inr}(y)) :\equiv e(y).$$

3.27.5 The finite types

We introduce primitive constants $\star$, $\mathbf{0}$, $\mathbf{1}$, satisfying the following rules:

- $\mathbf{0} : \mathcal{U}_0$, $\mathbf{1} : \mathcal{U}_0$
- $\star : \mathbf{1}$

Given $C : \mathbf{0} \to \mathcal{U}_n$ we can introduce a defined constant $f : \prod_{(x:\mathbf{0})} C(x)$, with no defining equations.
Given $C : \mathbf{1} \to \mathcal{U}_n$ and $d : C(\star)$ we can introduce a defined constant $f : \prod_{(x:\mathbf{1})} C(x)$, with defining equation $f(\star) :\equiv d$.

3.27.6 Natural numbers

The type of natural numbers is obtained by introducing primitive constants $\mathbb{N}$, 0, and succ with the following rules:

- $\mathbb{N} : \mathcal{U}_0$,
- $0 : \mathbb{N}$,
- $\mathsf{succ} : \mathbb{N} \to \mathbb{N}$.

Furthermore, we can define functions by primitive recursion. If we have $C : \mathbb{N} \to \mathcal{U}_k$ we can introduce a defined constant $f : \prod_{(x:\mathbb{N})} C(x)$ whenever we have

$$d : C(0)$$
$$e : \prod_{(x:\mathbb{N})} (C(x) \to C(\mathsf{succ}(x)))$$

with the defining equations

$$f(0) :\equiv d \qquad \text{and} \qquad f(\mathsf{succ}(x)) :\equiv e(x, f(x)).$$

3.27.7 W-types

For W-types we introduce primitive constants c_{W} and c_{sup}. An expression of the form $c_{\mathsf{W}}(A, \lambda x.\, B)$ is written as $\mathsf{W}_{(x:A)} B$, and an expression of the form $c_{\mathsf{sup}}(x, u)$ is written as $\mathsf{sup}(x, u)$:

- if $A : \mathcal{U}_n$ and $B : A \to \mathcal{U}_n$, then $\mathsf{W}_{(x:A)} B(x) : \mathcal{U}_n$
- if moreover, $a : A$ and $u : B(a) \to \mathsf{W}_{(x:A)} B(x)$ then $\mathsf{sup}(a, u) : \mathsf{W}_{(x:A)} B(x)$.

Here also we can define functions by total recursion. If we have A and B as above and $C : (\mathsf{W}_{(x:A)} B(x)) \to \mathcal{U}_m$, then we can introduce a defined constant $f : \prod_{(z:\mathsf{W}_{(x:A)} B(x))} C(z)$ whenever we have

$$d : \prod_{(a:A)} \prod_{(u:B(a) \to \mathsf{W}_{(x:A)} B(x))} ((\prod_{(y:B(a))} C(u(y))) \to C(\mathsf{sup}(a, u)))$$

with the defining equation

$$f(\mathsf{sup}(a, u)) :\equiv d(a, u, f \circ u).$$

3.27.8 Identity types

We introduce primitive constants $c_=$ and c_{refl}. We write $a =_A b$ for $c_=(A, a, b)$ and refl_a for $c_{\mathsf{refl}}(A, a)$, when $a : A$ is understood:

- If $A : \mathcal{U}_n$, $a : A$, and $b : A$ then $a =_A b : \mathcal{U}_n$.
- If $a : A$ then $\mathsf{refl}_a : a =_A a$.

Given $a : A$, if $y : A, z : a =_A y \vdash C : \mathcal{U}_m$ and $\vdash d : C[a, \mathsf{refl}_a / y, z]$ then we can introduce a defined constant

$$f : \prod_{(y:A)} \prod_{(z:a=_A y)} C$$

with defining equation

$$f(a, \mathsf{refl}_a) :\equiv d.$$

3.28 The second presentation

In this section, there are three kinds of judgments

$$\Gamma\ \mathsf{ctx} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash a \equiv a' : A$$

which we specify by providing inference rules for deriving them. A typical **inference rule** has the form

$$\frac{\mathcal{J}_1 \qquad \cdots \qquad \mathcal{J}_k}{\mathcal{J}}\ \textsc{Name}$$

It says that we may derive the **conclusion** $\mathcal{J}$, provided that we have already derived the **hypotheses** $\mathcal{J}_1, \dots, \mathcal{J}_k$.
A **derivation** of a judgment is a tree constructed from such inference rules, with the judgment at the root of the tree.

3.28.1 Contexts

A context is a list

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

The judgment $\Gamma\ \mathsf{ctx}$ formally expresses the fact that Γ is a well-formed context, and is governed by the rules of inference

$$\frac{}{\cdot\ \mathsf{ctx}}\ \mathsf{ctx\text{-}EMP} \qquad \frac{x_1{:}A_1, \ldots, x_{n-1}{:}A_{n-1} \vdash A_n : \mathcal{U}_i}{(x_1{:}A_1, \ldots, x_n{:}A_n)\ \mathsf{ctx}}\ \mathsf{ctx\text{-}EXT}$$

3.29 Homotopy type theory

In this section we state the additional axioms of homotopy type theory which distinguish it from standard Martin-Löf type theory: function extensionality, the univalence axiom, and higher inductive types. We state them in the style of the second presentation **??**, although the first presentation §3.27 could be used just as well.

3.29.1 Function extensionality and univalence

There are two basic ways of introducing axioms which do not introduce new syntax or judgmental equalities (function extensionality and univalence are of this form): either add a primitive constant to inhabit the axiom, or prove all theorems which depend on the axiom by hypothesizing a variable that inhabits the axiom, cf. **??**. While these are essentially equivalent, we opt for the former approach because we feel that the axioms of homotopy type theory are an essential part of the core theory.

Axiom 2.9.2 is formalized by introduction of a constant funext which asserts that happly is an equivalence:

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B \qquad \Gamma \vdash g : \prod_{(x:A)} B}{\Gamma \vdash \mathsf{funext}(f,g) : \mathsf{isequiv}(\mathsf{happly}_{f,g})}\ \Pi\text{-EXT}$$

The definitions of happly and $\mathsf{isequiv}$ can be found in (2.9.1) and **??**, respectively.

Axiom 2.10.3 is formalized in a similar fashion, too:

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash \mathsf{univalence}(A,B) : \mathsf{isequiv}(\mathsf{idtoeqv}_{A,B})}\ \mathcal{U}_i\text{-UNIV}$$

The definition of $\mathsf{idtoeqv}$ can be found in (2.10.2).

3.29.2 The circle

Here we give an example of a basic higher inductive type; others follow the same general scheme, albeit with elaborations.

Note that the rules below do not precisely follow the pattern of the ordinary inductive types in **??**: the rules refer to the notions of transport and functoriality of maps (§2.2), and the second computation rule is a propositional, not judgmental, equality. These differences are discussed in **??**.

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathbb{S}^1 : \mathcal{U}_i}\ \mathbb{S}^1\text{-FORM} \qquad \frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{base} : \mathbb{S}^1}\ \mathbb{S}^1\text{-INTRO}_1$$

$$\frac{\Gamma\ \mathsf{ctx}}{\Gamma \vdash \mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}}\ \mathbb{S}^1\text{-INTRO}_2$$

$$\frac{\begin{array}{c}\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \\ \Gamma \vdash b : C[\mathsf{base}/x] \qquad \Gamma \vdash \ell : b =^{C}_{\mathsf{loop}} b \qquad \Gamma \vdash p : \mathbb{S}^1\end{array}}{\Gamma \vdash \mathsf{ind}_{\mathbb{S}^1}(x.C, b, \ell, p) : C[p/x]}\ \mathbb{S}^1\text{-ELIM}$$

$$\frac{\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash b : C[\mathsf{base}/x] \qquad \Gamma \vdash \ell : b =^{C}_{\mathsf{loop}} b}{\Gamma \vdash \mathsf{ind}_{\mathbb{S}^1}(x.C, b, \ell, \mathsf{base}) \equiv b : C[\mathsf{base}/x]}\ \mathbb{S}^1\text{-COMP}_1$$

$$\frac{\Gamma, x{:}\mathbb{S}^1 \vdash C : \mathcal{U}_i \qquad \Gamma \vdash b : C[\mathsf{base}/x] \qquad \Gamma \vdash \ell : b =^{C}_{\mathsf{loop}} b}{\Gamma \vdash \mathbb{S}^1\text{-loopcomp} : \mathsf{apd}_{(\lambda y.\, \mathsf{ind}_{\mathbb{S}^1}(x.C,b,\ell,y))}(\mathsf{loop}) = \ell}\ \mathbb{S}^1\text{-COMP}_2$$

In $\mathsf{ind}_{\mathbb{S}^1}$, x is bound in C. The notation $b =^{C}_{\mathsf{loop}} b$ for dependent paths was introduced in **??**.