MATH 721: HOMOTOPY TYPE THEORY

EMILY RIEHL

Contents

Part 1. Martin-Löf's Dependent Type Theory

August 30: Dependent Type Theory

Martin-Löf's dependent type theory is a formal language for writing mathematics: both constructions of mathematical objects and proofs of mathematical propositions. As we shall discover, these two things are treated in parallel (in contrast to classical Set theory plus first-order logic, where the latter supplies the proof calculus and the former gives the language which you use to state things to prove).

Date: Fall 2021.

Judgments and contexts. I find it helpful to imagine I'm teaching a computer to do mathematics. It's also helpful to forget that you know other ways of doing mathematics.[1]

defn. There are four kinds of **judgments** in dependent type theory, which you can think of as the "grammatically correct" expressions:

(i) $\Gamma \vdash A \ \texttt{type}$, meaning that A is a well-formed type in **context** Γ (more about this soon).
(ii) $\Gamma \vdash a : A$, meaning that a is a well-formed term of type A in context Γ.
(iii) $\Gamma \vdash A \doteq B \ \texttt{type}$, meaning that A and B are **judgmentally** or **definitionally** equal types in context Γ.
(iv) $\Gamma \vdash a \doteq b : A$, meaning that a and b are judgmentally equal terms of type A in context Γ.

These might be collectively abbreviated by $\Gamma \vdash \mathcal{J}$.

The statement of a mathematical theorem, often begins with an expression like "Let n and m be positive integers, with $n < m$, and let $\vec{v}_1, \ldots, \vec{v}_m$ be vectors in $\mathbb{R}^n$. Then ..." This statement of the hypotheses defines a **context**, a finite list of types and hypothetical terms (called **variables**[2]) satisfying an inductive condition that that each type can be derived in the context of the previous types and terms using the inference rules of type theory.

defn. A **context** is a finite list of variable declarations:

$$x : A_1, x_2 : A_2(x_1), \ldots, x_n : A_n(x_1, \ldots, x_{n-1})$$

satisfying the condition that for each $1 \leq k \leq n$ we can derive the judgment

$$x_1 : A_1, \ldots, x_{k-1} : A_{k-1}(x_1, \ldots, x_{k-2}) \vdash A_k(x_1, \ldots, x_{k-1}) \ \texttt{type}$$

using the inference rules of type theory.

We'll introduce the inference rules shortly but the idea is that it needs to be possible to form the type $A_k(x_1, \ldots, x_{k-1})$ given terms $x_1, \ldots, x_{k-1}$ of the previously-formed types.

ex. For example, there is a unique context of length zero: the empty context.

ex. $n : \mathbb{N}, m : \mathbb{N}, p : n < m, \vec{v} : (\mathbb{R}^n)^m$ is a context. Here $n : \mathbb{N}, m : \mathbb{N} \vdash n < m$ is a dependent type that corresponds to the relation $\{n < m \mid n, m \in \mathbb{N}\} \subset \mathbb{N} \times \mathbb{N}$ and the variable p is a witness that $n < m$ is true (more about this later).

Type families. Absolutely everything in dependent type theory is context dependent so we always assume we're working in a background context Γ. Let's focus on the primary two judgment forms.

defn. Given a type A in context Γ a **family** of types over A in context Γ is a type $B(x)$ is context $\Gamma, x : A$, as represented by the judgment:

$$\Gamma, x : A \vdash B(x) \ \texttt{type}$$

We also say that $B(x)$ is a type **indexed** by $x : A$, in context Γ.

ex. $\mathbb{R}^n$ is a type indexed by $n \in \mathbb{N}$.

defn. Consider a type family B over A in context Γ. A **section** of the family B over A in context Γ is a term of type $B(x)$ in context $\Gamma, x : A$, as represented by the judgment:

$$\Gamma, x : A \vdash b(x) : B(x)$$

We say that b is a **section** of the family B over A in context Γ or that $b(x)$ is a term of type $B(x)$ indexed by $x : A$ in context Γ.

ex. $\vec{0}_n : \mathbb{R}^n$ is a term dependent on $n \in \mathbb{N}$.

Exercise. If you've heard the word "section" before you should think about what it is being used here.

[1] Indeed, there are very deep theorems that describe how to interpret dependent type theory into classical set-based mathematics. You're welcome to investigate these for your final project but they are beyond the scope of this course.

[2] We're not going to say anything about proper syntax for variables and instead rely on instinct to recognize proper and improper usage.

Inference rules. There are five types of inference rules that collectively describe the structural rules of dependent type theory. They are

(i) Rules postulating that judgmental equality is an equivalence relation:

$$\frac{\Gamma \vdash A \ \mathsf{type}}{\Gamma \vdash A \doteq A \ \mathsf{type}} \qquad \frac{\Gamma \vdash A \doteq B \ \mathsf{type}}{\Gamma \vdash B \doteq A \ \mathsf{type}} \qquad \frac{\Gamma \vdash A \doteq B \ \mathsf{type} \qquad \Gamma \vdash B \doteq C \ \mathsf{type}}{\Gamma \vdash A \doteq C \ \mathsf{type}}$$

and similarly for judgmental equality between terms.

(ii) Variable conversion rules for judgmental equality between types:

$$\frac{\Gamma \vdash A \doteq A' \ \mathsf{type} \qquad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, x : A', \Delta \vdash \mathcal{J}}$$

(iii) Substitution rules:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta \vdash \mathcal{J}}{\Gamma, \Delta[a/x] \vdash \mathcal{J}[a/x]}$$

If Δ is the context $y_1 : B_1(x), \dots, y_n : B_n(x, y_1, \dots, y_{n-1})$ then $\Delta[a/x]$ is the context $y_1 : B(a), \dots, y_n : B_n(a, y_1, \dots, y_{n-1})$. A similar substitution is performed in the judgment $\mathcal{J}[a/x]$. Further rules indicate that substitution by judgmentally equal terms gives judgmentally equal results.

(iv) Weakening rules:

$$\frac{\Gamma \vdash A \ \mathsf{type} \qquad \Gamma, \Delta \vdash \mathcal{J}}{\Gamma, x : A, \Delta \vdash \mathcal{J}}$$

Eg if A and B are types in context Γ, then B is also a type in context $\Gamma, x : A$.

(v) The generic term:

$$\frac{\Gamma \vdash A \ \mathsf{type}}{\Gamma, x : A \vdash x : A}$$

This will be used to define the identity function of any type.

Derivations. A derivation in type theory is a finite rooted tree where each node is a valid rule of inference. The root is the conclusion.

ex. The interchange rule is derived as follows

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash B \ \mathsf{type}}{\Gamma, y : B \vdash y : B}}{\Gamma, y : B, x : A \vdash y : B} \qquad \dfrac{\Gamma \vdash B \ \mathsf{type} \qquad \dfrac{\Gamma, x : A, y : B, \Delta \vdash \mathcal{J}}{\Gamma, x : A, z : B, \Delta[z/y] \vdash \mathcal{J}[z/y]}}{\Gamma, y : B, x : A, z : B, \Delta[z/y] \vdash \mathcal{J}[z/y]}}{\Gamma, y : B, x : A, \Delta \vdash \mathcal{J}}$$

September 1: Dependent function types & the natural numbers

The rules for dependent function types. Consider a section b of a family B over A in context Γ, as encoded by a judgment:

$$\Gamma, x : A \vdash b(x) : B(x).$$

We think of the section b as a function that takes as input $x : A$ and produces a term $b(x) : B(x)$. Since the type of the output is allowed to depend on the term being input, this isn't quite an ordinary function but a **dependent function**. The type of all dependent functions is the **dependent function type**

$$\prod_{x:A} B(x)$$

What is a thing in mathematics? Structuralism says the ontology of a thing is determined by its behavior. In dependent type theory, we define dependent function types by stating their rules, which have the following forms:

(i) **formation rules** tell us how a type may be formed
(ii) **introduction rules** tells us how to introduce new terms of the type
(iii) **elimination rules** tell us how the terms of a type may be used
(iv) **computation rules** tell us how the introduction and elimination rules interact

There are also **congruence rules** that tell us that all constructions respect judgmental equality. See your book for more details.

defn (dependent function types)**.** The Π-**formation rule** has the form:

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

The Π-**introduction rule** has the form:

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

The λ-**abstraction** $\lambda x.b(x)$ can be thought of as notation for $x \mapsto b(x)$.

The Π-**elimination rule** has the form of an evaluation rule:

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

Finally, there are two computation rules: the β-**rule**

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

and the η-**rule**, which says that all elements of a Π-type are dependent functions:

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

Ordinary function types.

defn (function types)**.** The formation rule is derived from the formation rule for Π-types together with weakening:

$$\cfrac{\cfrac{\Gamma \vdash A \; \texttt{type} \qquad \Gamma \vdash B \; \texttt{type}}{\Gamma, x : A \vdash B \; \texttt{type}}}{\Gamma \vdash \prod_{x:A} B \; \texttt{type}}$$

We adopt the notation

$$A \to B :\equiv \prod_{x:A} B$$

for the dependent function type in the case where the type family B is constant over $x : A$.

The introduction, evaluation, and computation rules are instances of term conversion: eg

$$\frac{\Gamma \vdash B \; \texttt{type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma \vdash \lambda x.b(x) : A \to B} \qquad \frac{\Gamma \vdash f : A \to B}{\Gamma, x : A \vdash f(x) : B}$$

plus the two computation rules:

$$\frac{\Gamma \vdash B \; \texttt{type} \qquad \Gamma, x : A \vdash b(x) : B}{\Gamma, x : A \vdash (\lambda y.b(y))(x) \doteq b(x) : B} \qquad \frac{\Gamma \vdash f : A \to B}{\Gamma \vdash \lambda x.f(x) \doteq f : A \to B}$$

defn. Identity functions are defined as follows:

$$\cfrac{\cfrac{\Gamma \vdash A \; \texttt{type}}{\Gamma, x : A \vdash x : A}}{\Gamma \vdash \lambda x.x : A \to A}$$

which is traditionally denoted by $\mathbf{id}_A :\equiv \lambda x.x$.

The idea of composition is that given a function $f\colon A \to B$ and $g\colon B \to C$ you should get a function $g \circ f\colon A \to C$. Using infix notation you might denote this function by $\_ \circ \_$.

Q. $\_ \circ \_$ is itself a function, so it's a term of some type. What type?[3]

defn. Composition has the form:

$$\frac{\Gamma \vdash A\ \mathtt{type} \qquad \Gamma \vdash B\ \mathtt{type} \qquad \Gamma \vdash C\ \mathtt{type}}{\Gamma \vdash \_ \circ \_ : (B \to C) \to ((A \to B) \to (A \to C))}$$

It is defined by

$$\_ \circ \_ := \lambda g.\lambda f.\lambda x.g(f(x))$$

which can be understood as the term constructed by three applications of the Π-introduction rule followed by two applications of the Π-elimination rule.

Composition is associative essentially because both $(h \circ g) \circ f$ and $h \circ (g \circ f)$ are defined by $\lambda x.h(g(f(x)))$. We'll think about this more formally when we come back to identity types.

Similarly, you can compute that for all $f : A \to B$, $\mathrm{id}_B \circ f \doteq f : A \to B$ and $f \circ \mathrm{id}_A \doteq f : A \to B$.

The type of natural numbers. The type $\mathbb{N}$ of natural numbers is the archetypical example of an **inductive type** about more which soon. It is given by rules which say that it has a term $0_{\mathbb{N}} : \mathbb{N}$, it has a successor function $\mathtt{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$ and it satisfies the induction principle.

The $\mathbb{N}$-formation rule is

$$\frac{}{\vdash \mathbb{N}\ \mathtt{type}}$$

In other words, $\mathbb{N}$ is a type in the empty context.

There are two $\mathbb{N}$-introduction rules:

$$\frac{}{\vdash 0_{\mathbb{N}} : \mathbb{N}} \qquad \frac{}{\vdash \mathtt{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}}$$

Digression (traditional induction). In traditional first-order logic, the principle of $\mathbb{N}$-induction is stated in terms of a **predicate** P over $\mathbb{N}$. One way to think about P is as a function $P\colon \mathbb{N} \to \{\top, \bot\}$. That is, for each $n \in \mathbb{N}$, $P(n)$ is either true or false. We could also think of P as an indexed family of sets $(P(n))_{n \in \mathbb{N}}$ where for each n either $P(n) = \varnothing$ (corresponding to $P(n)$ being false) or $P(n) = *$ (corresponding to $P(n)$ being true).

The induction principle then says

$$\forall P : \{0,1\}^{\mathbb{N}}, (P(0) \wedge (\forall n, P(n) \to P(n+1)) \to \forall n, P(n)).$$

In dependent type theory it is most natural to let P be an arbitrary type family over $\mathbb{N}$. This is a stronger assumption, as we'll see.

Q. What then corresponds to a proof that $\forall n, P(n)$?

The induction principle is encoded by the following rule:

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

Remark. There are other forms this rule might take that are interderivable with this one.

The computation rules say that the function $\mathtt{ind}_{\mathbb{N}}(p_0, p_S) : \prod_{n:\mathbb{N}} P(n)$ behaves like it should on $0_{\mathbb{N}}$ and successors:

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

and under the same premises

$$\Gamma, n : \mathbb{N} \vdash \mathtt{ind}_{\mathbb{N}}(p_0, p_S)(\mathtt{succ}_{\mathbb{N}}(n)) \doteq p_S(n, \mathtt{ind}_{\mathbb{N}}(p_0, p_S, n)) : P(\mathtt{succ}_{\mathbb{N}}(n)).$$

[3]Really the type should involve three universe variables but let's save this for next week.

These computation rules don't matter so much if the type family $n : \mathbb{N} \vdash P(n)$ is really a predicate — $P(n)$ is either true or false and that's the end of the story — but they do matter if $P(n)$ is more like an indexed family of sets. In the latter case, $\mathtt{ind}_{\mathbb{N}}(p_0, p_S)$ is the recursive function defined from p_0 and p_S and these are the computation rules for that recursion.

Remark. Recall Peano's axioms for the natural numbers:

(i) $0_{\mathbb{N}} \in \mathbb{N}$
(ii) $\mathsf{succ}_{\mathbb{N}} : \mathbb{N} \to \mathbb{N}$
(iii) $\forall n, \mathsf{succ}_{\mathbb{N}}(n) \neq 0_{\mathbb{N}}$
(iv) $\forall n, m, \mathsf{succ}_{\mathbb{N}}(n) = \mathsf{succ}_{\mathbb{N}}(m) \to n = m$
(v) induction

We'll be able to *prove* this missing two axioms from the induction principle we've assumed once we have identity types and universes. We'll come back to this in a few weeks.

Addition on the natural numbers.

Remark. When addition is defined by recursion on the second variable, from the computation rules associated to function types and the natural numbers type you can derive judgmental equalities

$$m + 0 \doteq m \quad \text{and} \quad m + \mathsf{succ}_{\mathbb{N}}(n) = \mathsf{succ}_{\mathbb{N}}(m + n).$$

But you can't derive the symmetric judgmental equalities.

We *will* be able to prove such equalities using the identity types, to be introduced shortly.

Pattern matching. To define a dependent function $f : \prod_{n:\mathbb{N}} P(n)$ by induction on n it suffices, by the elimination rule for the natural numbers type, to provide two terms:

$$p_0 : P(0_{\mathbb{N}}) \qquad p_S : \prod_{n:\mathbb{N}} P(n) \to P(\mathsf{succ}_{\mathbb{N}}(n)).$$

Thus the definition of f may be presented by writing

$$f(0_{\mathbb{N}}) :\equiv p_0 \qquad f(\mathsf{succ}_{\mathbb{N}}(n)) :\equiv p_S(n, f(n)).$$

This defines the function f by **pattern matching** on the variable n. When a function is defined in this form, the judgmental equalities accompanying the definition are immediately displayed.

September 8: The formal proof assistant `agda`

See https://github.com/emilyriehl/721/blob/master/introduction.agda

September 13: Inductive types

The rules for the natural numbers type $\mathbb{N}$ tell us:

(i) how to form terms in $\mathbb{N}$, and
(ii) how to define dependent functions in $\prod_{n:\mathbb{N}} P(n)$ for any type family $n : \mathbb{N} \vdash P(n)$ $\mathtt{type}$,

while providing two computation rules for those dependent functions.

Many types can be specified by stating how to form their terms and how to define dependent functions out of them. Such types are called **inductive types**.

The idea of inductive types. Recall a type is specified by its formation rules, its introduction rules, its elimination rules, and its computation rules. For inductive types, the introduction rules specify the **constructors** of the inductive type, while the elimination rule provides the **induction principle**. The computation rules provide definitional equalities for the induction principle.

In more detail:

(i) The constructors tell us what structure the identity type is given with.
(ii) The induction principle defines sections of any type family over the inductive type by specifying the behavior at the constructors.
(iii) The computation rules assert that the inductively defined section agrees on the constructors with the data used to define it. So there is one computation rule for each constructor.

The unit type. The formal definition of the **unit type** is as follows:

$$\vdash \mathbb{1}\ \mathtt{type} \qquad \vdash \star : \mathbb{1} \qquad \frac{x : \mathbb{1} \vdash P(x)\ \mathtt{type} \qquad p : P(\star)}{x : \mathbb{1} \vdash \mathtt{ind}_{\mathbb{1}}(p, x) : P(x)} \qquad \frac{x : \mathbb{1} \vdash P(x)\ \mathtt{type} \qquad p : P(\star)}{x : \mathbb{1} \vdash \mathtt{ind}_1(p, \star) \doteq p : P(\star)}$$

As an inductive type, the definition is packaged as follows:

defn. The **unit type** is a type $\mathbb{1}$ equipped with a term $\star : \mathbb{1}$ satisfying the inductive principle that for any family $x : \mathbb{1} \vdash P(x)$ there is a function

$$\mathtt{ind}_{\mathbb{1}} : P(\star) \to \prod_{x:1} P(x)$$

with the computation rule $\mathtt{ind}_1(p, \star) \doteq p$.

In agda, this definition has the form:

```
data unit : UU lzero where
  star : unit
```

Q. What does the induction rule look like for a constant type family A that does not depend on $\mathbb{1}$?

The empty type.

defn. The empty type is a type $\varnothing$ satisfying the induction principle that for any family of types $x : \varnothing \vdash P(x)$ there is a term

$$\mathtt{ind}_{\varnothing} : \prod_{x:\varnothing} P(x).$$

That is the empty type is the inductive type with no constructors. Thus there are no computation rules. In agda, this definition has the form:

```
data empty : UU lzero where
```

Remark. As a special case of the elimination rule for the empty type we have

$$\frac{\vdash A\ \mathtt{type}}{\mathtt{ex\text{-}falso} \coloneqq \mathtt{ind}_{\varnothing} : \varnothing \to A}$$

By the elimination rule for function types it follows that if we had a term $x : \varnothing$ then we could get a term in any type. The name comes from latin *ex falso quodlibet*: "from falsehood, anything."

We've already seen a few glimpses of logic in type theory, something we'll discuss more formally soon. The basic idea is that we can interpret the formation of a type as akin to the process of formulating a mathematical statement that could be a sentence (if its a type in the empty context) or a predicate (if it's a dependent type). The act of constructing a term in that type is then analogous to proving the proposition so-encoded. These ideas motivate the logically-inflected terms in what follows.

For instance, we can use the empty type to define a negation operation on types:

defn. For any type A, we define its **negation** by $\neg A \coloneqq A \to \varnothing$ and say the type A **is empty** if there is a term in this type.

Remark. To construct a term of type $\neg A$, use the introduction rule for function types and assume given a term $a : A$. The task then is to derive a term of $\varnothing$. In other words, we prove $\neg A$ by assuming A and deriving a contradiction. This proof technique is called **proof of negation**.

This should be contrasted with **proof by contradiction**, which aims to prove a proposition P by assuming $\neg P$ and deriving a contradiction. This uses the logical step "$\neg\neg P$ implies P." In type theory, however, $\neg\neg A$ is the type of functions

$$\neg\neg A \coloneqq (A \to \varnothing) \to \varnothing)$$

and it is not possible in general to use a term in this type to construct a term of type A.

The law of contraposition does work, at least in one direction.

Proposition. *For any types P and Q there is a function*

$$(P \to Q) \to (\neg Q \to \neg P).$$

Proof. By λ-abstraction assume given $f : P \to Q$ and $\tilde{q} : Q \to \varnothing$. We seek a term in $P \to \varnothing$, which we obtain simply by composing: $\tilde{q} \circ f : P \to \varnothing$. Thus

$$\lambda f.\lambda\tilde{q}.\lambda p.\tilde{q}(f(p)) : (P \to Q) \to (\neg Q \to \neg P). \qquad \square$$

Coproducts. Inductive types can be defined outside the empty context. For instance, the formation and introduction rules for the coproduct type have the form:

$$\frac{\Gamma \vdash A\ \mathtt{type} \qquad \Gamma \vdash B\ \mathtt{type}}{\Gamma \vdash A + B\ \mathtt{type}}$$

$$\frac{\Gamma \vdash A\ \mathtt{type} \qquad \Gamma \vdash B\ \mathtt{type} \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathtt{inl} a : A + B} \qquad \frac{\Gamma \vdash A\ \mathtt{type} \qquad \Gamma \vdash B\ \mathtt{type} \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathtt{inr} b : A + B}$$

defn. Given types A and B the **coproduct type** is the type equipped with

$$\mathtt{inl} : A \to A + B \qquad \mathtt{inr} : B \to A + B$$

satisfying the induction principle that says that for any family of types $x : A + B \vdash P(x)\ \mathtt{type}$ there is a term

$$\mathtt{ind}_+ : \left(\prod_{x:A} P(\mathtt{inl}(x))\right) \to \left(\prod_{y:B} P(\mathtt{inr}(y))\right) \to \prod_{z:A+B} P(z)$$

satisfying the computation rules

$$\mathtt{ind}_+(f, g, \mathtt{inl}(x)) \doteq f(x) \qquad \mathtt{ind}_+(f, g, \mathtt{inr}(y)) \doteq g(y).$$

Not as a special case we have

$$\mathtt{ind}_+ : (A \to X) \to (B \to X) \to (A + B \to X)$$

which is similar to the elimination rule for disjunction in first order logic: if you've proven that A implies X and that B implies X then you can conclude that A or B implies X.

The type of integers. There are many ways to define the integers in Martin-Löf type theory, one of which is as follows:

defn. Define the **integers** to be the type $\mathbb{Z} :\equiv \mathbb{N} + (\mathbb{1} + \mathbb{N})$ which comes equipped with inclusions:

$$\mathtt{in\text{-}pos} :\equiv \mathtt{inr} \circ \mathtt{inr} : \mathbb{N} \to \mathbb{Z} \qquad \mathtt{in\text{-}neg} :\equiv \mathtt{inl} : \mathbb{N} \to \mathbb{Z}$$

and constants

$$-1_{\mathbb{Z}} :\equiv \mathtt{in\text{-}neg}(0_{\mathbb{N}}) \qquad 0_{\mathbb{Z}} :\equiv \mathtt{inr}(\mathtt{inl}(\star)) \qquad 1_{\mathbb{Z}} :\equiv \mathtt{in\text{-}pos}(0_{\mathbb{N}}).$$

Since $\mathbb{Z}$ is built from inductive types it is then an inductive type given with its own induction principle.

Dependent pair types. Of all the inductive types we've introduced, the final one is perhaps the most important.

Recall a **dependent function** $\lambda x.f(x) : \prod_{x:A} B(x)$ is like an ordinary function except the output type is allowed to vary with the input term. Similarly, a **dependent pair** $(a, b) : \sum_{x:A} B(x)$ is like an ordinary (ordered) pair except the type of the second term $b : B(a)$ is allowed to vary with the first term $a : A$.

defn. Consider a type family $x : A \vdash B(x)\ \mathtt{type}$. The **dependent pair type** or **Σ-type** $\sum_{x:A} B(x)$ is the inductive type equipped with the function

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

The induction principle asserts that for any family of types $p : \sum_{x:A} B(x) \vdash P(p)\ \mathtt{type}$ there is a function

$$\mathtt{ind}_\Sigma : \left(\prod_{x:A} \prod_{y:B} P(\mathtt{pair}(x, y)\right) \to \left(\prod_{z:\sum_{x:A} B(x)} P(z)\right)$$

satisfying the computation rule $\mathtt{ind}_\Sigma(g, \mathtt{pair}(x, y)) \doteq g(x, y)$.

It is common to write "(x, y)" as shorthand for "$\mathtt{pair}(x, y)$."

defn. Given a type family $x : A \vdash B(x)\ \mathtt{type}$ by the induction principle for Σ-types, we have a function

$$\mathtt{pr}_1 : \sum_{x:A} B(x) \to A$$

defined by $\mathtt{pr}_1(x, y) :\equiv x$ and a dependent function

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

defined by $\mathtt{pr}_2(x, y) :\equiv y$.

When B is a constant type family over A, the type $\sum_{x:A} B$ is the type of ordinary pairs (x, y) where $x : A$ and $y : B$. Thus **product types** arise as special cases of Σ-types.

defn. Given types A and B their product type is the type $A \times B :\equiv \sum_{x:A} B$. It comes with a pairing function

$$(-,-) : A \to B \to A \times B$$

and satisfies an induction principle:

$$\mathtt{ind}_\times : \prod_{x:A} \prod_{y:B} P(x, y) \to \prod_{z:A\times B} P(z)$$

satisfying the computation rule $\mathtt{ind}_\times(g, (x, y)) \doteq g(x, y)$.

As a special case, we have

$$\mathtt{ind}_\times : (A \to B \to C) \to ((A \times B) \to C).$$

This is the inverse of the **currying function**. Thus $\mathtt{ind}_\times$ and $\mathtt{ind}_\Sigma$ sometimes go by the name **uncurrying**.

Dept. of Mathematics, Johns Hopkins University, 3400 N Charles St, Baltimore, MD 21218
E-mail address: `eriehl@math.jhu.edu`