Chapter 11

Dependent Types

We've seen types that depend on other types. They are defined using type constructors with type parameters, like `Maybe` or `[]`. Most programming languages have some support for generic data types—data types parameterized by other data types.

Categorically, such types are modeled as functors [1].

A natural generalization of this idea is to have types that are parameterized by values. For instance, it's often advantageous to encode the length of a list in its type. A list of length zero would have a different type than a list of length one, and so on.

Types parameterized by values are called *dependent types*. There are languages like Idris or Agda that have full support for dependent types. It's also possible to implement dependent types in Haskell, but support for them is still rather patchy.

The reason for using dependent types in programming is to make programs provably correct. In order to do that, the compiler must be able to check the assumptions made by the programmer.

Haskell, with its strong type system, is able to uncover a lot of bugs at compile time. For instance, it won't let you write `a <> b` (infix notation for `mappend`), unless you provide the `Monoid` instance for the type of your variables.

However, within Haskell's type system, there is no way to express or, much less enforce, the unit and associativity laws for the monoid. For that, the instance of the `Monoid` type class would have to carry with itself proofs of equality (not actual code):

```
assoc :: m <> (n <> p) = (m <> n) <> p
lunit :: mempty <> m = m
runit :: m <> mempty = m
```

Dependent types, and equality types in particular, pave the way towards this goal.

The material in this chapter is more advanced, and not used in the rest of the book, so you may safely skip it on first reading.

11.1 Dependent Vectors

We'll start with the standard example of a counted list, or a vector:

[1] A type constructor that has no `Functor` instance can be thought of as a functor from a discrete category—a category with no arrows other than identities

```
data Vec n a where
    VNil  :: Vec Z a
    VCons :: a -> Vec n a -> Vec (S n) a
```

The compiler will recognize this definition as dependently typed if you include the following language pragmas:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
```

The first argument to the type constructor is a natural number `n`. Notice: this is a value, not a type. The type checker is able to figure this out from the usage of `n` in the two data constructors. The first one creates a vector of the type `Vec Z a`, and the second creates a vector of the type `Vect (S n) a`, where `Z` and `S` are defined as the constructors of natural numbers:

```
data Nat = Z | S Nat
```

We can be more explicit about the parameters if we use the pragma:

```
{-# LANGUAGE KindSignatures #-}
```

and import the library:

```
import Data.Kind
```

We can then specify that `n` is a `Nat`, whereas `a` is a `Type`:

```
data Vec (n :: Nat) (a :: Type) where
    VNil  :: Vec Z a
    VCons :: a -> Vec n a -> Vec (S n) a
```

Using one of these definitions we can, for instance, construct a vector (of integers) of length zero:

```
emptyV :: Vec Z Int
emptyV = VNil
```

It has a different type than a vector of length one:

```
singleV :: Vec (S Z) Int
singleV = VCons 42 VNil
```

and so on.

We can now define a dependently typed function that returns the first element of a vector:

```
headV :: Vec (S n) a -> a
headV (VCons a _) = a
```

This function is guaranteed to work exclusively with non-zero-length vectors. These are the vectors whose size matches `(S n)`, which cannot be `Z`. If you try to call this function with `emptyV`, the compiler will flag the error.

Another example is a function that zips two vectors together. Encoded in its type signature is the requirement that the two vectors be of the same size `n` (the result is also of the size `n`):

```
zipV :: Vec n a -> Vec n b -> Vec n (a, b)
zipV (VCons a as) (VCons b bs) = VCons (a, b) (zipV as bs)
zipV VNil VNil = VNil
```

Exercise 11.1.1. *Implement the function `tailV` that returns the tail of the non-zero-length vector. Try calling it with `emptyV`.*

11.2 Dependent Types Categorically

The easiest way to visualize dependent types is to think of them as families of types indexed by elements of a set. In the case of counted vectors, the indexing set would be the set of natural numbers $\mathbb{N}$.

The zeroth type would be the unit type `()` representing an empty vector. The type corresponding to `(S Z)` would be `a`; then we'd have a pair `(a, a)`, a triple `(a, a, a)` and so on, with higher and higher powers of `a`.

If we want to talk about the whole family as one big set, we can take the sum of all these types. For instance, the sum of all powers of a is the familiar list type, a.k.a, a free monoid:

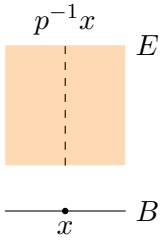
$$List(a) = \coprod_{n:\mathbb{N}} a^n$$

Fibrations

Although intuitively easy to visualize, this point of view doesn't generalize nicely to category theory, where we don't like mixing sets with objects. So we turn this picture on its head and instead of talking about injecting family members into the sum, we consider a mapping that goes in the opposite direction.

This, again, we can first visualize using sets. We have one big set E describing the whole family, and a function p called the projection, or a *display map*, that goes from E down to the indexing set B (also called the *base*).

This function will, in general, map multiple elements to one. We can then talk about the inverse image of a particular element $x \in B$ as the set of elements that get mapped down to it by p. This set is called the *fiber* and is written $p^{-1}x$ (even though, in general, p is not invertible in the usual sense). Seen as a collection of fibers, E is often called a *fiber bundle* or just a bundle.



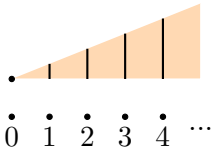
Now forget about sets. A *fibration* in an arbitrary category is a pair of objects E and B and an arrow $p\colon E \to B$.

So this is really just an arrow, but the context is everything. When an arrow is called a fibration, we use the intuition from sets, and imagine its source E as a collection of fibers, with p projecting each fiber down to a single point in the base B.

We will therefore model type families as fibrations. For instance, our counted-vector family can be represented as a fibration whose base is the type of natural numbers. The whole family is a sum (coproduct) of consecutive powers (products) of A:

$$List(A) = A^0 + A^1 + A^2 + ... = \coprod_{n\colon \mathbb{N}} A^n$$

with the zeroth power—the initial object—representing a vector of size zero.



The projection $p\colon List(A) \to \mathbb{N}$ is the familiar *length* function.

Slice categories

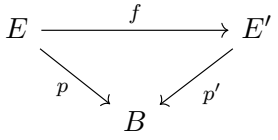
In category theory we like to describe things in bulk—defining internal structure of things by structure-preserving maps between them. Such is the case with fibrations.

If we fix the base object B and consider all possible source objects in the category $\mathcal{C}$, and all possible projections down to B, we get a *slice category* $\mathcal{C}/B$ (also known as an over-category).

An object in the slice category is a pair $\langle E, p\rangle$, with $p\colon E \to B$. An arrow between two objects $\langle E, p\rangle$ and $\langle E', p'\rangle$ is an arrow $f\colon E \to E'$ that commutes with the projections, that is:

$$p' \circ f = p$$

Again, the best way to visualize this is to notice that such an arrow maps fibers of p to fibers of p'. It's a "fiber-preserving" mapping between bundles.

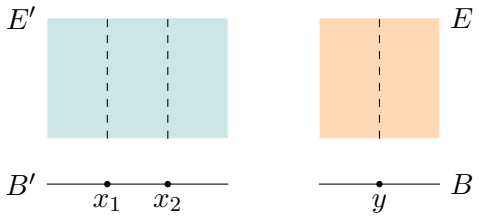


Our counted vectors can be seen as objects in the slice category $\mathcal{C}/\mathbb{N}$ given by pairs $\langle List(A), length\rangle$.

Pullbacks

Let's start with a particular fibration $p\colon E \to B$ and ask ourselves the question: what happens when we change the base from B to some B' that is related to it through a mapping $f\colon B' \to B$. Can we "pull the fibers back" along f?

Again, let's think about sets first. Imagine picking a fiber in E over some point $y \in B$ that is in the image of f. We can plant this fiber over all points in B' that are in the inverse image $f^{-1}y$. If multiple points in B' are mapped to the same point in B, we just duplicate the corresponding fiber. This way, every point in B' will have a fiber sticking out of it. The sum of all these fibers will form a new bundle E'.



We have thus constructed a new fibration with the base B'. Its projection $p'\colon E' \to B'$ maps each point in a given fiber to the point over which this fiber was planted. There is also an obvious mapping $g\colon E' \to E$ that maps fibers to their corresponding fibers.

By construction, this new fibration $\langle E', p' \rangle$ satisfies the condition:

$$p \circ g = f \circ p'$$

which can be represented as a commuting square:

$$\begin{array}{ccc} E' & \xrightarrow{g} & E \\ {\scriptstyle p'}\downarrow & & \downarrow{\scriptstyle p} \\ B' & \xrightarrow{f} & B \end{array}$$

In **Set**, we can explicitly construct E' as a *subset* of the cartesian product $B' \times E$ with $p' = \pi_1$ and $g = \pi_2$ (the two cartesian projections). An element of E' is a pair $\langle b, e \rangle$, such that:

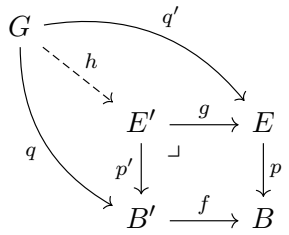
$$f(b) = p(e)$$

This commuting square is the starting point for the categorical generalization. However, even in **Set** there are many different fibrations over B' that make this diagram commute. We have to pick the universal one. Such a universal construction is called a *pullback*, or a *fibered product*.

A pullback of $p\colon E \to B$ along $f\colon B' \to B$ is an object E' together with two arrows $p'\colon E' \to B'$ and $g\colon E' \to E$ that makes the above diagram commute, and that satisfies the universal condition.

The universal condition says that, for any other candidate object G with two arrows $q'\colon G \to E$ and $q\colon G \to B'$ such that $p \circ q' = f \circ q$, there is a unique arrow $h\colon G \to E'$

that makes the two triangles commute:



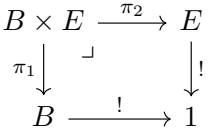
The angle symbol in the corner of the square is used to mark pullbacks.

If we look at the pullback through the prism of fibrations, E is a bundle over B, and we are constructing a new bundle E' out of the fibers taken from E. Where we plant these fibers over B' is determined by (the inverse image of) f. This procedure makes E' a bundle over both B' and B, the latter with the projection $p \circ g = f \circ p'$.

G in this picture is some other bundle over B' with the projection q. It is simultaneously a bundle over B with the projection $f \circ q = p \circ q'$. The unique mapping h maps the fibers of G given by q^{-1} to fibers of E' given by p'.

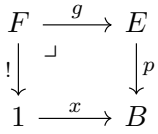
All mappings in this picture work on fibers. Some of them rearrange fibers over new bases—that's what a pullback does. This is analogous to what natural transformations do to containers. Others modify individual fibers—the mapping $h\colon G \to E'$ works like this. This is analogous to what `fmap` does to containers. The universal condition then tells us that q' can be factored into a transformation of fibers h, followed by the rearrangement g.

It's worth noting that picking the terminal object as the pullback target gives us automatically the definition of the categorical product:



Alternatively, we can think of this picture as a generalization of the diagonal functor. Normally, the diagonal functor ΔE would duplicate E. Here, you can imagine π_1 fibrating $B \times E$ into as many copies of E as there are elements in B. We'll use this analogy when we talk about the dependent sum and product.

Conversely, a single fiber can be extracted from a fibration by pulling it back to the terminal object. In this case the mapping $x\colon 1 \to B$ picks an element of the base, and the pullback along it extracts a single fiber F:



The arrow g injects this fiber back into E. By varying x we can pick different fibers in E.

Exercise 11.2.1. *Show that the pullback with the terminal object as the target is the product.*

Exercise 11.2.2. *Show that a pullback can be defined as a limit of the diagram from a stick-figure category with three objects:*

$$A \to B \leftarrow C$$

Exercise 11.2.3. *Show that a pullback in $\mathcal{C}$ with the target B is a product in the slice category $\mathcal{C}/B$. Hint: Define two projections as morphisms in the slice category. Use universality of the pullback to show the universality of the product.*

Base-change functor

We used a cartesian closed category as a model for programming. To model dependent types, we need to impose an additional condition: We require the category to be *locally cartesian closed*. This is a category in which all slice categories are cartesian closed.

In particular, such categories have all pullbacks, so it's always possible to change the base of any fibration. Base change induces a mapping between slice categories that is functorial.

Given two slice categories $\mathcal{C}/B'$ and $\mathcal{C}/B$ and an arrow between bases $f\colon B' \to B$ the base-change functor $f^*\colon \mathcal{C}/B \to \mathcal{C}/B'$ maps a fibration $\langle E, p\rangle$ to the fibration $f^*\langle E, p\rangle = \langle f^*E, f^*p\rangle$, which is given by the pullback:

$$\begin{array}{ccc} f^*E & \xrightarrow{g} & E \\ {\scriptstyle f^*p}\downarrow & \lrcorner & \downarrow{\scriptstyle p} \\ B' & \xrightarrow{f} & B \end{array}$$

Notice that the functor f^* goes in the opposite direction to the arrow f.

In a locally cartesian closed category, the base change functor has both the left and the right adjoints. The left adjoint is called the dependent sum, and the right adjoint is called the dependent product (or dependent function).

11.3 Dependent Sum

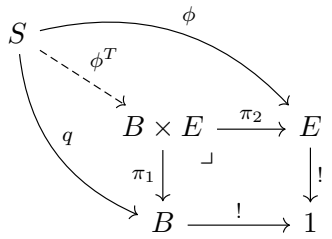
In type theory, the dependent sum, or the sigma type $\Sigma_{x:B}T(x)$, is defined as a type of pairs in which the type of the second component depends on the value of the first component. Our counted vector type can be thought of as a dependent sum. An element of this type is a natural number `n` paired with an element of an n-tuple `(a, a, ... a)`.

The introduction rule for the dependent sum assumes that there is a family of types $T(x)$ indexed by elements of the base type B. Then an element of $\Sigma_{x:B}T(x)$ is constructed from a pair of elements $x\colon B$ and $y\colon T(x)$.

Categorically, dependent sum is modeled as the left adjoint of the base-change functor.

To see this, let's first revisit the definition of a pair, which is an element of a product. We've noticed before that a product can be written as a pullback from the terminal object. Here's the universal construction for the product/pullback (the notation antic-

ipates the target of this construction):



We have also seen that the product can be defined using an adjunction. We can spot this adjunction in our diagram: for every pair of arrows $\langle \phi, q \rangle$ there is a unique arrow ϕ^T that makes the triangles commute.

Notice that, if we keep q fixed, we get a one-to-one correspondence between the arrows ϕ and ϕ^T. This is the adjunction we're interested in.

We can now put our fibrational glasses on and notice that $\langle S, q \rangle$ and $\langle B \times E, \pi_1 \rangle$ are two fibrations over the same base B. The commuting triangle makes ϕ^T a morphism in the slice category $\mathcal{C}/B$. In other words ϕ^T is a member of the hom-set:

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} B \times E \\ \pi_1 \end{matrix} \right\rangle\right)$$

We can rewrite the one-to-one correspondence between ϕ and ϕ^T as an isomorphism of hom-sets:

$$\mathcal{C}(S, E) \cong (\mathcal{C}/B)\left(\left\langle \begin{matrix} S \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} B \times E \\ \pi_1 \end{matrix} \right\rangle\right)$$

In fact, it's an adjunction in which the left functor is the forgetful functor that maps $\langle S, q \rangle$ to S, thus forgetting the fibration.

If you squint at this adjunction hard enough, you can see the outlines of the definition of a categorical sum (coproduct).

On the left we have a mapping out of S. Think of S as a sum of fibers that are defined by the fibration $\langle S, q \rangle$.

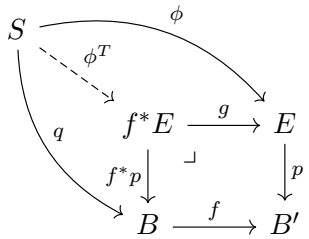
Recall that the fibration $\langle B \times E, \pi_1 \rangle$ can be though of as a generalization of the diagonal functor, producing many copies of E planted over B. Then the right hand side looks like a bunch of arrows, each mapping a different fiber of S to the same fiber E.

For comparison, this is the definition of a coproduct of two "fibers", F_1 and F_2:

$$\mathcal{C}(F_1 + F_2, E) \cong (\mathcal{C} \times \mathcal{C})(\langle F_1, F_2 \rangle, \Delta E)$$

Seen in this light, a dependent sum looks like a sum of many such fibers.

We can generalize our diagram by replacing the terminal object with an arbitrary base B'. We now have a fibration $\langle E, p \rangle$, and we get the pullback square that defines the base-change functor f^*:



The universality of the pullback results in the following isomorphism of hom-sets:

$$(\mathcal{C}/B')\left(\left\langle \begin{array}{c} S \\ f \circ q \end{array} \right\rangle, \left\langle \begin{array}{c} E \\ p \end{array} \right\rangle\right) \cong (\mathcal{C}/B)\left(\left\langle \begin{array}{c} S \\ q \end{array} \right\rangle, f^* \left\langle \begin{array}{c} E \\ p \end{array} \right\rangle\right)$$

Here, ϕ is an element of the left-hand side and ϕ^T is the corresponding element of the right-hand side.

We interpret this isomorphism as the adjunction between the base change functor f^* on the right and the dependent sum functor on the left:

$$\Sigma_f \left\langle \begin{array}{c} S \\ q \end{array} \right\rangle = \left\langle \begin{array}{c} S \\ f \circ q \end{array} \right\rangle$$

This simply says that, if S is fibered over B and there is a mapping from B to B', then S is automatically fibered over B'; the projection being the composition $f \circ q$.

In **Set**, if f merges several elements into one, the fiber over this element is the sum of individual fibers.

If f is identity, Σ_{id} is the identity in the slice category, which confirms our intuition of S being the sum of fibers.

Exercise 11.3.1. *Show that the right adjoint to the obvious forgetful functor $U : \mathcal{C}/B \to \mathcal{C}$ is:*

$$F(A) = \langle B \times A, \pi_1 \rangle$$

Existential quantification

In the *propositions as types* interpretation, type families correspond to families of propositions. The dependent sum type $\Sigma_{x:B}\, T(x)$ corresponds to the proposition: There exists an x for which $T(x)$ is true:

$$\exists_{x:B}\, T(x)$$

Indeed, a term of the type $\Sigma_{x:B}\, T(x)$ is a pair of an element $x : B$ and an element $y : T(x)$—which shows that $T(x)$ is inhabited for some x.

11.4 Dependent Product

In type theory, the dependent product, or dependent function, or pi-type $\Pi_{x:B} T(x)$, is defined as a function whose return type depends on the value of its argument.

It's called a function, because you can evaluate it. Given a dependent function $f : \Pi_{x:B} T(x)$, you may apply it to an argument $x : B$ to get a value $f(x) : T(x)$.

A simple example is a function that constructs a vector of a given size and fills it with copies of a given value:

```
replicateV :: a -> SNat n -> Vec n a
replicateV _ SZ  = VNil
replicateV x (SS n) = VCons x (replicateV x n)
```

At the time of this writing, Haskell's support for dependent types is limited, so the implementation of dependent functions requires the use of singleton types. In this case, the number that is the argument to `replicateV` is passed as a singleton natural:

```
data SNat n where
  SZ :: SNat Z
  SS :: SNat n -> SNat (S n)
```

(Note that `replicateV` is a function of two arguments, so it can be either considered a dependent function of a pair, or a regular function returning a dependent function.)

Before we describe the categorical model of dependent functions, it's instructive to consider how they work on sets. A dependent function selects one element from each set $T(x)$.

You may visualize this selection as a large tuple—or an element of the cartesian product that you get by multiplying together all the sets in the family. This is the meaning of the product notation, $\Pi_{x:B} T(x)$. For instance, in the trivial case of B a two-element set $\{1, 2\}$, a dependent function is just a cartesian product $T(1) \times T(2)$.

In our example, `replicateV` picks a particular counted vector for each value of `n`. Counted vectors are equivalent to tuples so, for `n` equal zero, `replicateV` returns an empty tuple `()`; for `n` equals one it returns a single value `x`; for `n` equals two, it returns a homogenous pair `(x, x)`; etc.

The function `replicateV`, evaluated at some `x :: a`, is equivalent to an infinite tuple of tuples:

$$((), x, (x, x), (x, x, x), ...)$$

which is a specific element of the more general type:

$$((), a, (a, a), (a, a, a), ...)$$

In order to build a categorical model of dependent functions, we need to change our perspective from a family of types to a fibration. We start with a bundle E/B fibered by the projection $p\colon E \to B$. A dependent function is called a *section* of this bundle.

If you visualize the bundle as a bunch of fibers sticking out from the base B, a section is like a haircut: it cuts through each fiber to produce a corresponding value. In physics, such sections are called fields—with spacetime as the base.

Just like we talked about a function object representing a set of functions, we can talk about an object $S(E)$ that represents a set of sections of a given bundle E.

Just like we defined function application as a mapping out of the product:

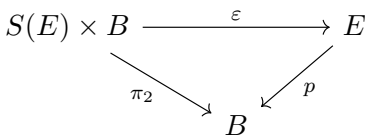
$$\varepsilon_{B,C}\colon C^B \times B \to C$$

we can define the dependent function application as a mapping:

$$\varepsilon\colon S(E) \times B \to E$$

We can visualize it as picking a section in S and an element of the base B and producing a value in the bundle E.

But this time we have to insist that this value be in the correct fiber. If we project the result of ε using p, we should get the same element of B that was used in the product $S(E) \times B$. In other words, this diagram should commute:



This makes ε a morphism in the slice category $\mathcal{C}/B$.

And just like the exponential object was universal, so is the object of sections. The universality condition has the same form: for any other object G with an arrow $\phi\colon G\times B \to E$ there is a unique arrow $\phi^T : G \to S(E)$ that makes the following diagram commute:

$$\begin{array}{ccc} G\times B & & \\ {\scriptstyle \phi^T\times B}\downarrow & \searrow^{\phi} & \\ S(E)\times B & \xrightarrow{\varepsilon} & E \end{array}$$

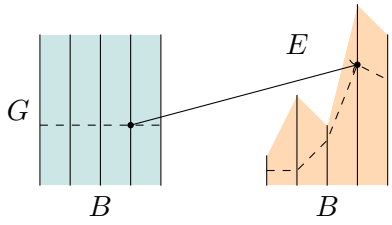
The difference is that now both ε and ϕ are morphisms in the slice category $\mathcal{C}/B$.

The one-to-one correspondence between ϕ and ϕ^T forms an adjunction:

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} G\times B \\ \pi_2 \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right) \cong \mathcal{C}(G, S(E))$$

which we can use as the definition of the object of sections $S(E)$.

Recall that we can interpret the product $G \times B$ as a generalization of the diagonal functor. It takes copies of G and plants them as identical fibers over each element of B. Fixing an element of G means slicing horizontally through $G \times B$. The left hand side of the adjunction maps this slice, fiber-by-fiber, to a section of E.



An element of the right hand side of the adjunction takes an element of G and maps it to an element of $S(E)$. Thus elements of $S(E)$ are in one-to-one correspondence with section of E.

These are all set-theoretical intuitions. We can generalize them by first noticing that the right hand side of the adjunction can be trivially expressed as a hom-set in the slice category $\mathcal{C}/1$ over the terminal object.

There is one-to-one correspondence between objects X in $\mathcal{C}$ and objects $\langle X, !\rangle$ in $\mathcal{C}/1$. Arrows in $\mathcal{C}/1$ are just the arrows of $\mathcal{C}$ with no additional constraints. We therefore have:

$$(\mathcal{C}/B)\left(\left\langle \begin{matrix} G\times B \\ \pi_2 \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right) \cong (\mathcal{C}/1)\left(\left\langle \begin{matrix} G \\ ! \end{matrix} \right\rangle, \left\langle \begin{matrix} S(E) \\ ! \end{matrix} \right\rangle\right)$$

The next step is to "blur the focus" by replacing the terminal object with a more general base B'.

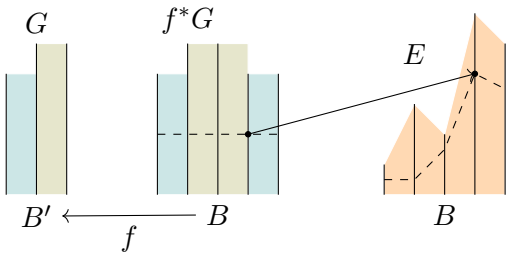
The right-hand side becomes a hom-set in the slice category $\mathcal{C}/B'$ and G gets fibrated by some $q\colon G \to B'$.

We can then replace the product $G \times B$ with a more general pullback of q along some $f\colon B \to B'$.

$$\begin{array}{ccc} G\times B & \xrightarrow{\pi_1} & G \\ \downarrow{\scriptstyle \pi_2} & \lrcorner & \downarrow{\scriptstyle !} \\ B & \xrightarrow{!} & 1 \end{array} \quad\longrightarrow\quad \begin{array}{ccc} f^*G & \xrightarrow{g} & G \\ {\scriptstyle f^*q}\downarrow & \lrcorner & \downarrow{\scriptstyle q} \\ B & \xrightarrow{f} & B' \end{array}$$

The result is that, instead of a bunch of G fibers over B, we get a pullback f^*G that is populated by groups of fibers from the fibration $q\colon G \to B'$.

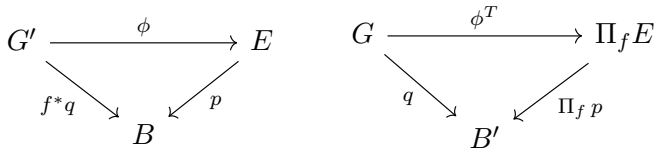
Imagine, for instance, that B' is a two-element set. The fibration q will split G into two fibers. These fibers will be replanted over B to form f^*G. The replanting is guided by f^{-1}.



The adjunction that defines the dependent function type is therefore:

$$(\mathcal{C}/B)\left(f^*\left\langle \begin{matrix} G \\ q \end{matrix} \right\rangle, \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right) \cong (\mathcal{C}/B')\left(\left\langle \begin{matrix} G \\ q \end{matrix} \right\rangle, \Pi_f \left\langle \begin{matrix} E \\ p \end{matrix} \right\rangle\right)$$

where $\Pi_f E$ generalizes the object of sections $S(E)$. The adjunction is a mapping between morphisms in their respective slice categories:



Universal quantification

The logical interpretation of the dependent product $\Pi_{x:B}\, T(x)$ is a universally quanitified proposition. An element of $\Pi_{x:B}\, T(x)$ is a section—the proof that it's possible to select an element from each member of the family $T(x)$. It means that non of them is empty. In other words, its a proof of the proposition:

$$\forall_{x:B}\, T(x)$$

11.5 Equality

Our first experience in mathematics involves equality. We learn that

$$1 + 1 = 2$$

and we don't think much of it afterwards.

But what does it mean that $1 + 1$ is equal to 2? Two is a number, but one plus one is an expression, so they are not the same thing. There is some mental processing that we have to perform before we pronounce these two things equal.

Contrast this with the statement $0 = 0$, in which both sides of equality are *the same thing.*

It makes sense that, if we are to define equality, we'll have to at least make sure that everything is equal to itself. We call this property *reflexivity.*

Recall our definition of natural numbers:

```
data Nat where
  Z :: Nat
  S :: Nat -> Nat
```

This is how we can define equality for natural numbers:

```
equal :: Nat -> Nat -> Bool
equal Z Z = True
equal (S m) (S n) = equal m n
equal _ _ = False
```

We are recursively stripping S's in each number until one of them reaches Z. If the other reaches Z at the same time, we pronounce the numbers we started with to be equal, otherwise they are not.

Equational reasoning

Notice that, when defining equality in Haskell, we were already using the equal sign. For instance, the equal sign in:

```
equal Z Z = True
```

tells us that wherever we see the expression `equal Z Z` we can replace it with `True` and vice versa.

This is the principle of substituting equals for equals, which is the basis for *equational reasoning* in Haskell. We can't encode proofs of equality directly in Haskell, but we can use equational reasoning to reason about Haskell programs. This is one of the main advantages of pure functional programming. You can't perform such substitutions in imperative languages, because of side effects.

If we want to prove that $1 + 1$ is 2, we have to first define addition. The definition can either be recursive in the first or in the second argument. This one recurses in the second argument:

```
add :: Nat -> Nat -> Nat
add n Z = n
add n (S m) = S (add n m)
```

We encode $1 + 1$ as:

```
add (S Z) (S Z)
```

We can now use the definition of `add` to simplify this expression. We try to match the first clause, and we fail, because `S Z` is not the same as `Z`. But the second clause matches. In it, `n` is an arbitrary number, so we can substitute `S Z` for it, and get:

```
add (S Z) (S Z) = S (add (S Z) Z)
```

In this expression we can perform another substitution of equals using the first clause of the definition of `add` (again, with `n` replaced by `S Z`):

```
add (S Z) Z = (S Z)
```

We arrive at:

```
add (S Z) (S Z) = S (S Z)
```

We can clearly see that the right-hand side is the encoding of 2. But we haven't shown that our definition of equality is reflexive so, in principle, we don't know if:

```
eq (S (S Z)) (S (S Z))
```

yields `True`. We have to use step-by-step equational reasoning again:

```
equal (S (S Z) (S (S Z)) =
{- second clause of the definition of equal -}
equal (S Z) (S Z) =
{- second clause of the definition of equal -}
equal Z Z =
{- first clause of the definition of equal -}
True
```

We can use this kind of reasoning to prove statements about concrete numbers, but we run into problems when reasoning about generic numbers—for instance, showing that something is true for all `n`. Using our definition of addition, we can easily show that `add n Z` is the same as `n`. But we can't prove that `add Z n` is the same as `n`. The latter proof requires the use of induction.

We end up distinguishing between two kinds of equality. One is proven using substitutions, or rewriting rules, and is called *definitional equality*. You can think of it as macro expansion or inline expansion in programming languages. It also involves β-reductions: performing function application by replacing formal parameters by actual arguments, as in:

```
(\x -> x + x) 2 =
{- beta reduction -}
2 + 2
```

The second more interesting kind of equality is called *propositional equality* and it may require actual proofs.

Equality vs isomorphism

We said that category theorists prefer isomorphism over equality—at least when it comes to objects. It is true that, within the confines of a category, there is no way to differentiate between isomorphic objects. In general, though, equality is stronger than isomorphism. This is a problem, because it's very convenient to be able to substi-

tute equals for equals, but it's not always clear that one can substitute isomorphic for isomorphic.

Mathematicians have been struggling with this problem, mostly trying to modify the definition of isomorphism—but a real breakthrough came when they decided to simultaneously weaken the definition of equality. This led to the development of *homotopy type theory*, or HoTT for short.

Roughly speaking, in type theory, specifically in Martin-Löf theory of dependent types, equality is represented as a type, and in order to prove equality one has to construct an element of that type—in the spirit of the Curry-Howard interpretation.

Furthermore, in HoTT, the proofs themselves can be compared for equality, and so on ad infinitum. You can picture this by considering proofs of equality not as points but as some abstract paths that can be morphed into each other; hence the language of homotopies.

In this setting, instead of isomorphism, which involves strict equalities of arrows:

$$f \circ g = id$$

$$g \circ f = id$$

one defines an *equivalence*, in which these equalities are treated as types.

The main idea of HoTT is that one can impose the *univalence axiom* which, roughly speaking, states that equalities are equivalent to equivalences, or symbolically:

$$(A = B) \cong (A \cong B)$$

Notice that this is an axiom, not a theorem. We can either take it or leave it and the theory is still valid (at least we think so).

Equality types

Suppose that you want to compare two terms for equality. The first requirement is that both terms be of the same type. You can't compare apples with oranges. Don't get confused by some programming languages allowing comparisons of unlike terms: in every such case there is an implicit conversion involved, and the final equality is always between same-type values.

For every pair of values there is, in principle, a separate type of proofs of equality. There is a type for $0 = 0$, there is a type for $1 = 1$, and there is a type for $1 = 0$; the latter hopefully uninhabited.

Equality type, a.k.a., identity type, is therefore a dependent type: it depends on the two values that we are comparing. It's usually written as Id_A, where A is the type of both values, or using an infix notation as $x =_A y$ (equal sign with the subscript A).

For instance, the type of equality of two zeros is written as $Id_{\mathbb{N}}(0, 0)$ or:

$$0 =_{\mathbb{N}} 0$$

Notice: this is not a statement or a term. It's a *type*, like `Int` or `Bool`. You can define a value of this type if you have an introduction rule for it.

Introduction rule

The introduction rule for the equality type is the dependent function:

$$\mathit{refl}_A : \Pi_{x:A} Id_A(x, x)$$

which can be interpreted in the spirit of propositions as types as the proof of the statement:

$$\forall_{x:A}\ x = x$$

This is the familiar reflexivity: it shows that, for all x of type A, x is equal to itself. You can apply this function to some concrete value x of type A, and it will produce a new value of type $Id_A(x, x)$.

We can now prove that $0 = 0$. We can execute $\mathit{refl}_{\mathbb{N}}(0)$ to get a value of the type $0 =_{\mathbb{N}} 0$. This value is the proof that the type is inhabited, and therefore corresponds to a true proposition.

This is the only introduction rule for equality, so you might think that all proofs of equality boil down to "they are equal because they are the same." Surprisingly, this is not the case.

β-reduction and η-conversion

In type theory we have this interplay of introduction and elimination rules that essentially makes them the inverse of each other.

Consider the definition of a product. We introduce it by providing two values, $x : A$ and $y : B$ and we get a value $p : A \times B$. We can then eliminate it by extracting two values using two projections. But how do we know if these are the same values that we used to construct it? This is something that we have to postulate (in the categorical model this follows from the universal construction). We call it the computation rule or the β-reduction rule.

Conversely, if we are given a value $p : A \times B$, we can extract the two components using projections, and then use the introduction rule to recompose it. But how do we know that we'll get the same p? This too has to be postulated. This is sometimes called the uniqueness condition, or the η-conversion rule.

The equality type also has the elimination rule, which we'll discuss shortly, but we don't impose the uniqueness condition. It means that it's possible that there are some equality proofs that were not obtained using *refl*.

This is exactly the weakening of the notion of equality that makes HoTT interesting to mathematicians.

Induction principle for natural numbers

Before formulating the elimination rule for equality, it's instructive to first discuss a simpler elimination rule for natural numbers. We've already seen such rule describing primitive recursion. It allowed us to define recursive functions by specifying a value *init* and a function *step*.

Using dependent types, we can generalize this rule to define the *dependent elimination rule* that is equivalent to the principle of mathematical induction.

The principle of induction can be described as a device to prove, in one fell swoop, whole families of propositions indexed by natural numbers. For instance, the statement

that `add Z n` is equal to `n` is really an infinite number of propositions, one per each value of `n`.

We could, in principle, write a program that would meticulously verify this statement for a very large number of cases, but we'd never be sure if it holds in general. There are some conjectures about natural numbers that have been tested this way using computers but, obviously, they can never exhaust an infinite set of cases.

Roughly speaking, we can divide all mathematical theorems into two groups: the ones that can be easily formulated and the ones whose formulation is complex. They can be further subdivided into the ones whose proofs are simple, and the ones that are hard or impossible to prove. For instance, the famous Fermat's Last Theorem was extremely easy to formulate, but its proof required some massively complex mathematical machinery.

Here, we are interested in theorems about natural numbers that are both easy to formulate and easy to prove. We'll assume that we know how to generate a family of propositions or, equivalently, a dependent type $T(n)$, where n is a natural number.

We'll also assume that we have a value:

$$\mathit{init} : T(Z)$$

or, equivalently, the proof of the zeroth proposition; and a dependent function:

$$\mathit{step} : \Pi_{n:\mathbb{N}}\ (T(n) \to T(Sn))$$

This function is interpreted as generating a proof of the $(n+1)$st proposition from the proof of the nth proposition.

The *dependent elimination rule* for natural numbers tells us that, given such *init* and *step*, there exists a dependent function:

$$f : \Pi_{n:\mathbb{N}}\, T(n)$$

This function is interpreted as providing the proof that $T(n)$ is true for all n.

Moreover, this function, when applied to zero reproduces *init*:

$$f(Z) = \mathit{init}$$

and, when applied to the successor of n, is consistent with taking a *step*:

$$f(Sn) = (\mathit{step}(n))(f(n))$$

(Here, $\mathit{step}(n)$ produces a function, which is then applied to the value $f(n)$.) These are the two *computation rules* for natural numbers.

Notice that the induction principle is not a theorem about natural numbers. It's part of the *definition* of the type of natural numbers.

Not all dependent mappings out of natural numbers can be decomposed into *init* and *step*, just as not all theorems about natural numbers can be proven inductively. There is no η-conversion rule for natural numbers.

Equality elimination rule

The elimination rule for equality type is somewhat analogous to the induction principle for natural numbers. There we used *init* to ground ourselves at the start of the journey, and *step* to make progress. The elimination rule for equality requires a more powerful grounding, but it doesn't have a *step*. There really is no good analogy for how it works, other than through a leap of faith.

The idea is that we want to construct a mapping *out* of the equality type. But since equality type is itself a two-parameter family of types, the mapping out should be a dependent function. The target of this function is another family of types:

$$T(x, y, p)$$

that depends on the pair of values that are being compared $x, y\colon A$ and the proof of equality $p\colon Id(x, y)$.

The function we are trying to construct is:

$$f\colon \Pi_{x,y:A}\Pi_{p:Id(x,y)}\, T(x, y, p)$$

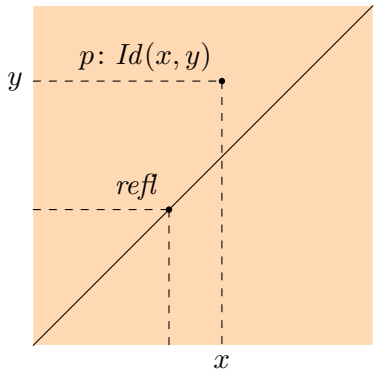
It's convenient to think of it as generating a proof that for all points x and y, and for every proof that the two are equal, the proposition $T(x, y, p)$ is true. Notice that, potentially, we have a different proposition for *every proof* that the two points are equal.

The least that we can demand from $T(x, y, p)$ is that it should be true when x and y are literally the same, and the equality proof is the obvious *refl*. This requirement can be expressed as a dependent function:

$$t\colon \Pi_{x:A}\, T\,(x, x, \mathit{refl}(x))$$

Notice that we are not even considering proofs of $x = x$, other than those given by reflexivity. Do such proofs exist? We don't know and we don't care.

So this is our grounding, the starting point of a journey that should lead us to defining our f for all pairs of points and all proofs of equality. The intuition is that we are defining f as a function on a plane (x, y), with a third dimension given by p. To do that, we're given something that's defined on the diagonal (x, x), with p restricted to *refl*.



You would think that we need something more, some kind of a *step* that would move us from one point to another. But, unlike with natural numbers, there is no *next*

point or *next* equality proof to jump to. All we have at our disposal is the function t and nothing else.

Therefore we postulate that, given a type family $T(x, y, p)$ and a function:

$$t\colon \Pi_{x:A}\, T\,(x, x, \mathit{refl}(x))$$

there exists a function:

$$f\colon \Pi_{x,y:A}\Pi_{p:Id(x,y)}\, T(x, y, p)$$

such that (computation rule):

$$f(x, x, \mathit{refl}(x)) = t(x)$$

Notice that the equality in the computation rule is *definitional equality*, not a type.

Equality elimination tells us that it's alway possible to extend the function t, which is defined on the diagonal, to the whole 3-d space.

This is a very strong postulate. One way to understand it is to argue that, within the framework of type theory—which is formulated using the language of introduction and elimination rules, and the rules for manipulating those—it's *impossible* to define a type family $T(x, y, p)$ that would *not* satisfy the equality elimination rule.

The closest analogy that we've seen so far is the result of parametricity, which states that, in Haskell, all polymorphic functions between endofunctors are automatically natural transformations. Another example, this time from calculus, is that any analytic function defined on the real axis has a unique extension to the whole complex plane.

The use of dependent types blurs the boundary between programming and mathematics. There is a whole spectrum of languages, with Haskell barely dipping its toes in dependent types while still firmly established in commercial usage; all the way to theorem provers, which are helping mathematicians formalize mathematical proofs.