Constructive Type Theory

JT Paasch

This is a basic introduction to the constructive type theory (CTT) of Per Martin-Löf. It is also sometimes called intuitionistic type theory (ITT). It will be helpful to have some familiarity with natural deduction, first order logic, or perhaps even the lambda calculus. However, I have tried to make no assumptions, and have attempted to explain everything (without being too verbose).

Even the simplest fully-worked CTT examples are not trivial, and derivations can quickly become rather large. For this reason, many introductions to constructive type theory do not include fully worked examples.

Here, I have tried to provide lots of fully worked examples, in order to illustrate all of the basic concepts. In particular, I focus on how to use elimination rules to build functions, and I spend a good deal of time explaining how dependent types play out. To keep everything as simple as possible, I only use finite types, and I avoid recursive types.

1 Introduction

Constructive type theory (CTT) was designed as a system that we can use to reason about mathematics constructively. It is something like a metalanguage, that we can use to reason about arbitrary data structures and other logics.

CTT typically comes with a basic collection of types like the booleans and the natural numbers, and it also comes with a set of operations that let us build more complex types out of simpler types. However, CTT is designed so it can be extended. You can specify your own types, so that you can reason about whatever sorts of objects you like. Here I will start with no types, and then explicitly build up all of the types that get used in the examples.

There are many variants of CTT. Even Martin-Löf developed a few different versions of his type theory over the course of his career. In what follows, I will only discuss a simple fragment that is more or less common to all variants of Martin-Löf's type theory.

2 Types

In CTT, the fundamental entity is a judgment. There are a variety of judgments that we can make. One judgment is this:

$$\mathsf{A} : \mathsf{Type}$$

This is a judgment that A is a type, where the symbol A is a placeholder for any legal type.

If you like, you can think of a type as a set or collection of objects. However, it is not exactly the same thing as a mathematical set, because technically, a mathematical set has no type restrictions. The elements of a set can be incongruous items that do not match each other in type.

A type, on the other hand, is a classifier of sorts. It classifies objects that are the same in kind, and puts them together under the same type. So a type is a collection of items that are all the same in kind.

An example is the set of boolean values. There are only two values — true and false — but both are of the same type: they are booleans. So it makes intuitive sense to say that $\mathbb{B}$ is a type:

$$\mathbb{B} : \mathsf{Type}$$

Another example are the natural numbers, $\mathbb{N}$. Strictly speaking, zero is a natural number, and so is the successor of zero (i.e., one), and so is the successor of the successor of zero (i.e., two). Each natural number is a different, but all are the same sorts of items: they are all natural numbers. So it makes intuitive sense to think that $\mathbb{N}$ is a type:

$$\mathbb{N} : \mathsf{Type}$$

That being said, we are not limited to thinking about types as sets (whose members are all of the same kind). We can think about types in a variety of other ways:

A is a type.
A is a set (whose members are the same in kind).
A is a problem to be solved.
A is the specification of a computer program.
A is a proposition.

One of the core philosophical ideas that underlies CTT is that all of the above turn out to be the same at deep levels.

Another judgment we can make that is related to types takes this form:

$$\mathsf{A} = \mathsf{B} : \mathsf{Type}$$

This judgment says that A and B are just different names for the same type.

3 The inhabitants of types

Another judgment we can make is this:

$$\mathsf{a} : \mathsf{A}$$

This is a judgment that a is an inhabitant of the type A. There are a variety of different ways that we can read this judgment:

a is an inhabitant of A.
a is an element of the set A.
a is a solution to the problem A.
a is a program that computes the specified output A.
a is a proof of the proposition A.

Another form of judgment we can make that is related to the inhabitants of types takes this shape:

$$\mathsf{a} = \mathsf{b} : \mathsf{A}$$

This judgment says that a and b are different names for the same inhabitant of type A.

4 Substitution rules

In CTT, there are substitution rules that let us substitute names, if they are names for the same things.

For example, if A and B are names for the same type, then we can substitute one for the other:

$$\frac{\mathsf{A} = \mathsf{B} : \mathsf{Type} \qquad \mathsf{A} : \mathsf{Type}}{\mathsf{B} : \mathsf{Type}}\ \mathsf{Sub}_1$$

If a and b are names for the same inhabitant of a type, then we can substitute one for the other:

$$\frac{\mathsf{A} : \mathsf{Type} \qquad \mathsf{a} = \mathsf{b} : \mathsf{A} \qquad \mathsf{a} : \mathsf{A}}{\mathsf{b} : \mathsf{A}}\ \mathsf{Sub}_2$$

5 Uniqueness of types

In the version of type theory I will use here, objects belong to only one base type. Consider this judgment:

$$\mathsf{a} : \mathsf{A}$$

By making this judgment, we are saying that a is intrinsically an object of type A. It is, at the very core of what it is, an A.

Moreover, a cannot belong to another base type. For instance, if $\mathsf{a} : \mathsf{A}$, then the following is illegal:

$$\mathsf{a} : \mathsf{B}$$

This restriction only applies to base types. As we will see, we can build up complex types out of simpler types, and we will see that the object a can then be a constituent part of more complex types. So there is a sense in which a can belong to more than one type.

However, complex types are not base types. The restriction is that objects can only belong to one base type. This restriction is here for practical purposes. It keeps the system simpler, and gives it nicer properties.

6 Defining types

To define any type in CTT, we must lay down four different kinds of rules. They are these:

1. Formation rules
2. Introduction rules
3. Elimination rules
4. Computation rules

I will not give examples of these rules just now. Instead, I will give examples later on, when I introduce some types. At this point though, let me briefly discuss what each type of rule is for.

1. The formation rules stipulate when a type is well-formed. We might say that the formation rules tell us when a type is a legal type, or that they tell us what types exist in our system and when exactly it is legitimate to use them.

2. The introduction rules specify how we introduce inhabitants of the type. We might say that the introduction rules enumerate or spell out exactly which objects inhabit the type. So if the formation rules tell us which types there are in the system, the introduction rules tell us which objects inhabit those types.

3. The elimination rules specify how to iterate over every inhabitant of a type, and compute some further value for each inhabitant. This is a general induction principle, and at the same time a way to define total functions.

4. The computation rules specify how to compute functions defined through an elimination rule. The elimination rule is a static rule that tells us how to define the function, but the computation rules are dynamic, computational rules, that tell us how to compute values with those functions.

If we specify all of these rules for a type, then we have fully defined the type. A type must have explicit rules that tell us how to form the type, we must have explicit rules that tell us which objects inhabit the type, we must have explicit rules that tell us how to define functions over the type, and we must have explicit rules that tell us how to compute answers using those functions.

It is worth noting that in CTT, we do not need to "interpret" any of its symbols relative to a model. The semantics of every type gets entirely baked in. These four types of rules are designed to provide all the information that we need to work with a type, so there is no need for a separate realm of interpretation and models.

7 Finite type formation

A very simple type is a finite type, which has a finite and definite set of inhabitants. To define a finite type, we first must specify formation rules. For example, here is the formation rule for the boolean type $\mathbb{B}$:

$$\frac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\text{-}\mathsf{F}$$

Formation rules are written as inference rules. We write any premises or preconditions above the line, and we put the conclusion or result below the line. In this case, there is nothing written above the line, so we can conclude that $\mathbb{B}$ is a type that is based on no other types.

Another way to think of this is to say that the $\mathbb{B}$ type is a given base type, which is not built from any other types. If we wanted to form a complex type, we would have to build it from other, simpler types. For example, suppose we want to combine a type A and a type B into the product type $\mathsf{A} \times \mathsf{B}$. The formation rule would look like this:

$$\frac{\mathsf{A} : \mathsf{Type} \qquad \mathsf{B} : \mathsf{Type}}{\mathsf{A} \times \mathsf{B}}\ \times\text{-}\mathsf{F}$$

Here we can see that there are types above the line, which means that we can form the type $\mathsf{A} \times \mathsf{B}$ only if A is itself a type and B is itself a type.

But for $\mathbb{B}$, we have nothing written above the line, so $\mathbb{B}$ is a simple base type, which is not constructed out of any other types.

We can define any other finite type in similar ways. For example, suppose we want to set up a type for phone number extensions, called Ext. We can put together a similar formation rule for this type:

$$\frac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext}\text{-}\mathsf{F}$$

This says that Ext is a base type that can be used in our system, and that it is constructed from no other types.

Similarly, suppose we want to set up a type for the employees in a company, called Em. Here is the formation rule:

$$\frac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}$$

This too says that Em is a base type that is available for use in the system, and it is not constructed from any other types.

8 Finite type introduction

After we define the formation rules for a type, we must define the introduction rules for the inhabitants of the type. There can be as many introduction rules as we like, but altogether they must tell us how to construct each and every inhabitant of a type.

For finite types, this is easy, because we simply enumerate each one. For example, the boolean type $\mathbb{B}$ has two inhabitants: tt (which is an abbreviation for "true") and ff (for "false"). So, we set down a separate introduction for each of these:

$$\frac{\mathbb{B} : \mathsf{Type}}{\mathsf{tt} : \mathbb{B}}\ \mathsf{tt\text{-}I} \qquad\qquad \frac{\mathbb{B} : \mathsf{Type}}{\mathsf{ff} : \mathbb{B}}\ \mathsf{ff\text{-}I}$$

The first of these rules says that if $\mathbb{B}$ is a legal type, then tt inhabits that type. The second says that if $\mathbb{B}$ is a legal type, then ff inhabits that type.

Notice that we can chain together the formation rules and the introduction rules to offer a complete proof or derivation of each of these inhabitants:

$$\frac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\mathsf{\text{-}F}}{\mathsf{tt} : \mathbb{B}}\ \mathsf{tt\text{-}I} \qquad\qquad \frac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\mathsf{\text{-}F}}{\mathsf{ff} : \mathbb{B}}\ \mathsf{ff\text{-}I}$$

The derivation on the left starts with no assumptions are preconditions above the line, and it uses the $\mathbb{B}$-F rule to conclude that $\mathbb{B}$ is a legal type. Then, using that as a premise, it uses the tt-I rule to conclude that tt inhabits that type. The derivation on the right is similar, except it derives that ff inhabits the type $\mathbb{B}$.

We can specify the inhabitants of any other finite type in just the same way: you simply enumerate each inhabitant by constructing a separate introduction rule for it. For example, suppose that there are two employees, Alice and Bob. Here are their introduction rules:

$$\frac{\mathsf{Em} : \mathsf{Type}}{\mathsf{Alice} : \mathsf{Em}}\ \mathsf{Alice\text{-}I} \qquad\qquad \frac{\mathsf{Em} : \mathsf{Type}}{\mathsf{Bob} : \mathsf{Em}}\ \mathsf{Bob\text{-}I}$$

Similarly, let us suppose that there are three phone extensions, x11, x12, and x13. Here are their introduction rules:

$$\frac{\mathsf{Ext} : \mathsf{Type}}{\mathsf{x11} : \mathsf{Ext}}\ \mathsf{x11\text{-}I} \qquad \frac{\mathsf{Ext} : \mathsf{Type}}{\mathsf{x12} : \mathsf{Ext}}\ \mathsf{x12\text{-}I} \qquad \frac{\mathsf{Ext} : \mathsf{Type}}{\mathsf{x13} : \mathsf{Ext}}\ \mathsf{x13\text{-}I}$$

The first rule says that if Ext is a legal type, then x11 inhabits it. The second and third rules are similar, but for x12 and x13.

As another example, suppose we have another finite type for office rooms, which we'll call Rm:

$$\dfrac{}{\mathsf{Rm} : \mathsf{Type}}\ \mathsf{Rm\text{-}F}$$

Suppose there are three rooms, r01, r02, and r03. Here are introduction rules for each one:

$$\dfrac{\mathsf{Rm} : \mathsf{Type}}{\mathsf{r01} : \mathsf{Rm}}\ \mathsf{r01\text{-}I} \qquad \dfrac{\mathsf{Rm} : \mathsf{Type}}{\mathsf{r02} : \mathsf{Rm}}\ \mathsf{r02\text{-}I} \qquad \dfrac{\mathsf{Rm} : \mathsf{Type}}{\mathsf{r03} : \mathsf{Rm}}\ \mathsf{r03\text{-}I}$$

To construct the inhabitants of finite types like those mentioned above, we have to manually specify each and every one that we want in the type. But we might also want a type with an unlimited number of inhabitants. For that, we can lay down recursive introduction rules, that tell us how to construct any given inhabitant of that type.

For example, here are the introduction rules for the natural numbers:

$$\dfrac{\mathbb{N} : \mathsf{Type}}{\mathsf{0} : \mathbb{N}}\ \mathsf{0\text{-}I} \qquad\qquad \dfrac{\mathbb{N} : \mathsf{Type} \qquad \mathsf{n} : \mathbb{N}}{\mathsf{succ}(\mathsf{n}) : \mathbb{N}}\ \mathsf{succ\text{-}I}$$

The rule on the left says that if ℕ is a legal type, then zero is an inhabitant of that type. The rule on the right is more complex. It says that, so long as ℕ is a legal type, if we also have some object n that inhabits ℕ, then the successor succ(n) is also an inhabitant of ℕ.

Using these rules, we can see how to construct any element of ℕ. To start, we can use the 0-I rule to construct zero:

$$\dfrac{\dfrac{}{\mathbb{N} : \mathsf{Type}}\ \mathbb{N}\text{-}\mathsf{F}}{\mathsf{0} : \mathbb{N}}\ \mathsf{0\text{-}I}$$

Then we can take that, and use the succ-I rule to form the successor of zero (which is one):

$$\dfrac{\dfrac{}{\mathbb{N} : \mathsf{Type}}\ \mathbb{N}\text{-}\mathsf{F} \qquad \dfrac{\dfrac{}{\mathbb{N} : \mathsf{Type}}\ \mathbb{N}\text{-}\mathsf{F}}{\mathsf{0} : \mathbb{N}}\ \mathsf{0\text{-}I}}{\mathsf{succ}(\mathsf{0}) : \mathbb{N}}\ \mathsf{succ\text{-}I}$$

Then we can use the succ-I rule again to form the successor of the successor of zero (which is two):

$$\dfrac{\dfrac{}{\mathbb{N} : \mathsf{Type}}\ \mathbb{N}\text{-}\mathsf{F} \qquad \dfrac{\dfrac{}{\mathbb{N} : \mathsf{Type}}\ \mathbb{N}\text{-}\mathsf{F} \qquad \dfrac{\dfrac{}{\mathbb{N} : \mathsf{Type}}\ \mathbb{N}\text{-}\mathsf{F}}{\mathsf{0} : \mathbb{N}}\ \mathsf{0\text{-}I}}{\mathsf{succ}(\mathsf{0}) : \mathbb{N}}\ \mathsf{succ\text{-}I}}{\mathsf{succ}(\mathsf{succ}(\mathsf{0})) : \mathbb{N}}\ \mathsf{succ\text{-}I}$$

Of course, there is an unending supply of natural numbers here, but the two introduction rules are complete because they show us how to construct each and every inhabitant. Using these rules, we can always construct another number (namely, the successor of the one before).

9 Selectors

In CTT, elimination rules use a special device called a selector. A selector is like a function or a program that says: if you give me an object from type A, I will select an object from B and return that to you.

9.1 Switches.

For finite types, we can think of a selector as a case switch. Suppose we have a type A, which has three inhabitants, a_1, a_2, and a_3. Suppose we also have another type, B, which has only two inhabitants, b_1 and b_2.

Here is one possible case switch:

```
switch on $(x : A)$ :
    case 1 (if $x$ is $a_1$) : return $b_1 : B$
    case 2 (if $x$ is $a_2$) : return $b_2 : B$
    case 3 (if $x$ is $a_3$) : return $b_1 : B$
```

This takes an object of type A and returns an object from type B. But which item from B it returns depends on which item of A we give it. If we give it a_1, it will give us back b_1. If we hive it a_3, it will also give us back b_1. If we give it a_2, it will give us back b_2.

This is just one example of selecting objects of type B, given an object of type A. The selector mechanism does not force us to use any particular assignment. As users of the system, we get to specify which objects from B we want to assign to which objects of A.

Here is another possible switch that we could put together:

```
switch on $(x : A)$ :
    case 1 (if $x$ is $a_1$) : return $b_2 : B$
    case 2 (if $x$ is $a_2$) : return $b_2 : B$
    case 3 (if $x$ is $a_3$) : return $b_2 : B$
```

This one selects the same object from B for every case of A. Any possible assignment of objects from B to A is a legal selector.

I should note that we can pick any type for the second type. It need not be B. We might choose to select objects from C for every case of A, or objects of D, and so on.

We can even select objects from the same type for every case of A. Here is an example of a switch that assigns to every object of A another object of A:

```
switch on (x : A) :
    case 1 (if x is $a_1$) : return $a_3$ : A
    case 2 (if x is $a_2$) : return $a_2$ : A
    case 3 (if x is $a_3$) : return $a_1$ : A
```

In principle, we can construct any switch over A that we like, provided that we can cover every case of A. (We will see examples later where we cannot construct a switch over A, because we cannot cover every case of A.)

9.2 Elimination rule template.

In CTT, we do not write switches in the way I wrote them above. Instead, we write them down in two steps. First, we construct an elimination rule, and second we construct computation rules.

To construct the elimination rule, we first compress the switch's inputs and outputs into a compact expression that has this shape:

$$\mathsf{switch}(\mathsf{x} : \mathsf{A}, \mathsf{option\ 1} : \mathsf{B}, \mathsf{option\ 2} : \mathsf{B}, \ldots) : \mathsf{B}$$

The first parameter to the switch statement is $\mathsf{x} : \mathsf{A}$, which indicates that we are switching over objects of type A. The remaining parameters to the switch statement list out (in order) the outputs for each case. Finally, on the far right, after the last colon, we stipulate the type of the switch's output (namely B).

Of course, for any of this to make sense, x must be a valid inhabitant of A, and each of option 1, option 2, ... must be valid inhabitants of B. The elimination rule for A makes this explicit:

$$\frac{\mathsf{x} : \mathsf{A} \qquad \mathsf{option\ 1} : \mathsf{B} \qquad \mathsf{option\ 2} : \mathsf{B} \qquad \ldots}{\mathsf{switch}(\mathsf{x} : \mathsf{A}, \mathsf{option\ 1} : \mathsf{B}, \mathsf{option\ 2} : \mathsf{B}, \ldots) : \mathsf{B}}\ \mathsf{A\text{-}E}$$

There is a lot of redundant information in the switch statement, so we can drop the type annotations on the parameters:

$$\frac{\mathsf{x} : \mathsf{A} \qquad \mathsf{option\ 1} : \mathsf{B} \qquad \mathsf{option\ 2} : \mathsf{B} \qquad \ldots}{\mathsf{switch}(\mathsf{x}, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) : \mathsf{B}}\ \mathsf{A\text{-}E}$$

9.3 Computation rules

Once we have formed the elimination rule, we then need to specify separately how the switch statement works. To do this, we explicitly list out what the switch statement should return for each case of A. Let us denote the inhabitants of type A as a_1, a_2, and so on. Then we can write out a separate rule for each case of A:

$$\begin{aligned} \mathsf{switch}(\mathsf{a}_1, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) &= \mathsf{option\ 1} : \mathsf{B} \\ \mathsf{switch}(\mathsf{a}_2, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) &= \mathsf{option\ 2} : \mathsf{B} \\ &\ldots \end{aligned}$$

The first line says that, if x is a_1, then the switch statement outputs or is equivalent to $\mathsf{option\ 1}$ of type B. The second line says that, if x is a_2, then the switch statement outputs or is equivalent to $\mathsf{option\ 2}$ of type B. And so on for all the other cases of A. In this way, we explicitly write out how to evaluate each case.

Again, these statements only make sense if a_1 is a valid inhabitant of type A, and each of $\mathsf{option\ 1}$, $\mathsf{option\ 2}$, and so on are valid inhabitants of type B. We make this explicit by putting each of these computation rules into a separate rule.

Here is the full version of the first computation rule listed above:

$$\frac{\mathsf{a}_1 : \mathsf{A} \qquad \mathsf{option\ 1} : \mathsf{B} \qquad \mathsf{option\ 2} : \mathsf{B} \qquad \ldots}{\mathsf{switch}(\mathsf{a}_1, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) = \mathsf{option\ 1} : \mathsf{B}}\ \mathsf{A\text{-}Eq}_1$$

This says that, so long as a_1 is a valid inhabitant of type A, and so long as each of $\mathsf{option\ 1}$, $\mathsf{option\ 2}$, and so on are valid inhabitants of type B, then the switch statement $\mathsf{switch}(\mathsf{a}_1, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots)$ evaluates to (or is just another name for) $\mathsf{option\ 1}$ of type B.

Here is the full version of the second computation listed above:

$$\frac{\mathsf{a}_2 : \mathsf{A} \qquad \mathsf{option\ 1} : \mathsf{B} \qquad \mathsf{option\ 2} : \mathsf{B} \qquad \ldots}{\mathsf{switch}(\mathsf{a}_2, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) = \mathsf{option\ 2} : \mathsf{B}}\ \mathsf{A\text{-}Eq}_1$$

9.4 Substitution

In their conclusions, the computation rules express an equality. They say that the switch statement is the same name as an inhabitant of B. For example, take the first computation rule:

$$\frac{\mathsf{a}_1 : \mathsf{A} \qquad \mathsf{option\ 1} : \mathsf{B} \qquad \mathsf{option\ 2} : \mathsf{B} \qquad \ldots}{\mathsf{switch}(\mathsf{a}_1, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) = \mathsf{option\ 1} : \mathsf{B}}\ \mathsf{A\text{-}Eq}_1$$

This says that the switch statement $\mathsf{switch}(\mathsf{a}_1, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots)$ is just another name for the item $\mathsf{option\ 1}$ of type B.

We can use the substitution rules to extract just the computed value, namely $\mathsf{option\ 1} : \mathsf{B}$. In particular, we can use Sub_2, like this:

$$\frac{\dfrac{\mathsf{a}_1 : \mathsf{A} \qquad \mathsf{option\ 1} : \mathsf{B} \qquad \mathsf{option\ 2} : \mathsf{B} \qquad \ldots}{\mathsf{switch}(\mathsf{a}_1, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) = \mathsf{option\ 1} : \mathsf{B}}\ \mathsf{A\text{-}Eq}_1 \qquad \mathsf{option\ 1} : \mathsf{B}}{\mathsf{option\ 1} : \mathsf{B}}\ \mathsf{Sub}_2$$

Similarly, if we use the second computation rule, we can extract the result in the same way:

$$\dfrac{\dfrac{\mathsf{a}_2 : \mathsf{A} \quad \mathsf{option\ 1 : B} \quad \mathsf{option\ 2 : B} \quad \ldots}{\mathsf{switch}(\mathsf{a}_2, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) = \mathsf{option\ 2 : B}}\,\mathsf{A\text{-}Eq}_2 \qquad \mathsf{option\ 2 : B}}{\mathsf{option\ 2 : B}}\,\mathsf{Sub}_2$$

9.5 Derivations

The full elimination and computation rules require that the inhabitants and types they switch over are valid objects in the system. So, the required types and inhabitants appear as premises in the inference rules.

But of course, those premises must be derived from introduction and formation rules in the system too. If we fill out the derivation steps for the premises as well, and then if we fill out the derivation steps for whatever their premises are, and so on until we have fully fleshed out everything, then we end up with a full derivation.

Take the computation of the first value of the switch above$\mathsf{option\ 2}$ from the switch above:

$$\dfrac{\dfrac{\mathsf{a}_2 : \mathsf{A} \quad \mathsf{option\ 1 : B} \quad \mathsf{option\ 2 : B} \quad \ldots}{\mathsf{switch}(\mathsf{a}_2, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) = \mathsf{option\ 2 : B}}\,\mathsf{A\text{-}Eq}_2 \qquad \mathsf{option\ 2 : B}}{\mathsf{option\ 2 : B}}\,\mathsf{Sub}_2$$

To save space, I will write o_1, o_2, and so on in place of $\mathsf{option\ 1}$, $\mathsf{option\ 2}$, etc.:

$$\dfrac{\dfrac{\mathsf{a}_2 : \mathsf{A} \quad \mathsf{o}_1 : \mathsf{B} \quad \mathsf{o}_2 : \mathsf{B} \quad \ldots}{\mathsf{switch}(\mathsf{a}_2, \mathsf{o}_1, \mathsf{o}_2, \ldots) = \mathsf{o}_2 : \mathsf{B}}\,\mathsf{A\text{-}Eq}_2 \qquad \mathsf{o}_2 : \mathsf{B}}{\mathsf{o}_2 : \mathsf{B}}\,\mathsf{Sub}_2$$

In order to fully flesh out this derivation, we need to show that all of the premises are valid in the system. So, for example, we would need to show that a_2 is a valid inhabitant of A by using its introduction rule:

$$\dfrac{\dfrac{\dfrac{\mathsf{A} : \mathsf{Type}}{\mathsf{a}_2 : \mathsf{A}}\,\mathsf{a}_2\text{-}\mathsf{I} \quad \mathsf{o}_1 : \mathsf{B} \quad \mathsf{o}_2 : \mathsf{B} \quad \ldots}{\mathsf{switch}(\mathsf{a}_2, \mathsf{o}_1, \mathsf{o}_2, \ldots) = \mathsf{o}_2 : \mathsf{B}}\,\mathsf{A\text{-}Eq}_2 \qquad \mathsf{o}_2 : \mathsf{B}}{\mathsf{o}_2 : \mathsf{B}}\,\mathsf{Sub}_2$$

Now we have shown that a_2 is a valid inhabitant of A, provided that A is a valid type. So, we need to show that A is a valid type too. We would have to do that with its formation rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{A} : \mathsf{Type}}\,\mathsf{A\text{-}F}}{\mathsf{a}_2 : \mathsf{A}}\,\mathsf{a}_2\mathsf{\text{-}I} \qquad \mathsf{o}_1 : \mathsf{B} \quad \mathsf{o}_2 : \mathsf{B} \quad \ldots}{\mathsf{switch}(\mathsf{a}_2, \mathsf{o}_1, \mathsf{o}_2, \ldots) = \mathsf{o}_2 : \mathsf{B}}\,\mathsf{A\text{-}Eq}_2 \qquad \mathsf{o}_2 : \mathsf{B}}{\mathsf{o}_2 : \mathsf{B}}\,\mathsf{Sub}_2$$

Now we have fully shown that a_2 is a valid type in the system, and so it can be used in the switch. We must also show that all of the other objects in play are valid too. Here is what a fully fleshed out derivation tree would look like:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{A} : \mathsf{Type}}\,\mathsf{A\text{-}F}}{\mathsf{a}_2 : \mathsf{A}}\,\mathsf{a}_2\mathsf{\text{-}I} \quad \dfrac{\dfrac{}{\mathsf{B} : \mathsf{Type}}\,\mathsf{B\text{-}F}}{\mathsf{o}_1 : \mathsf{B}}\,\mathsf{o}_1\mathsf{\text{-}I} \quad \dfrac{\dfrac{}{\mathsf{B} : \mathsf{Type}}\,\mathsf{B\text{-}F}}{\mathsf{o}_2 : \mathsf{B}}\,\mathsf{o}_2\mathsf{\text{-}I} \quad \ldots}{\mathsf{switch}(\mathsf{a}_2, \mathsf{o}_1, \mathsf{o}_2, \ldots) = \mathsf{o}_2 : \mathsf{B}}\,\mathsf{A\text{-}Eq}_2 \qquad \dfrac{\dfrac{}{\mathsf{B} : \mathsf{Type}}\,\mathsf{B\text{-}F}}{\mathsf{o}_2 : \mathsf{B}}\,\mathsf{o}_2\mathsf{\text{-}I}}{\mathsf{o}_2 : \mathsf{B}}\,\mathsf{Sub}_2$$

10 Selector examples

Let us construct some switches over the employee type Em that we defined earlier. There are two inhabitants of the type: Alice and Bob.

Suppose we want to assign a phone extension to each of them. For instance, like this:

```
switch on (x : Em) :
    case 1 (if x is Alice) : return x11 : Ext
    case 2 (if x is Bob) : return x12 : Ext
```

In other words, we want to assign the extension x11 to Alice, and x12 to Bob. We can convert this into an elimination rule, by following the template above.

To use the template, we need to substitute our own values in place of the values for the types and inhabitants of A and B. We will use the type Em in place of A, with Alice and Bob in place of a_1 and a_2. We will use the type Ext in place of B, with x11 and x12 in place of option 1 and option 2.

Here is the elimination rule:

$$\dfrac{\mathsf{x} : \mathsf{Em} \qquad \mathsf{x11} : \mathsf{Ext} \qquad \mathsf{x12} : \mathsf{Ext}}{\mathsf{switch}(\mathsf{x}, \mathsf{x11}, \mathsf{x12}) : \mathsf{Ext}}\,\mathsf{Em\text{-}E}$$

And here are the computation rules:

$$\dfrac{\mathsf{Alice} : \mathsf{Em} \qquad \mathsf{x11} : \mathsf{Ext} \qquad \mathsf{x12} : \mathsf{Ext}}{\mathsf{switch}(\mathsf{Alice}, \mathsf{x11}, \mathsf{x12}) = \mathsf{x11} : \mathsf{Ext}}\,\mathsf{Em\text{-}Eq}_1$$

$$\frac{\mathsf{Bob} : \mathsf{Em} \qquad \mathsf{x11} : \mathsf{Ext} \qquad \mathsf{x12} : \mathsf{Ext}}{\mathsf{switch}(\mathsf{Bob}, \mathsf{x11}, \mathsf{x12}) = \mathsf{x12} : \mathsf{Ext}}\ \mathsf{Em\text{-}Eq}_2$$

Using this, we can construct a full derivation to show that, using this particular switch, Alice's extension is x11.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}}{\mathsf{Alice} : \mathsf{Em}}\ \mathsf{Alice\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x11} : \mathsf{Ext}}\ \mathsf{x11\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x12} : \mathsf{Ext}}\ \mathsf{x12\text{-}I}}{\mathsf{switch}(\mathsf{Alice}, \mathsf{x11}, \mathsf{x12}) = \mathsf{x11} : \mathsf{Ext}}\ \mathsf{Em\text{-}Eq}_1 \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x11} : \mathsf{Ext}}\ \mathsf{x11\text{-}I}}{\mathsf{x11} : \mathsf{Ext}}\ \mathsf{Sub}_2$$

If we wanted to, we could also define a different switch. For example, maybe we want to assign x12 to Alice and x13 to Bob. Given that particular assignment of extensions to employees, we could show that Bob's extension is x13:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}}{\mathsf{Alice} : \mathsf{Em}}\ \mathsf{Alice\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x12} : \mathsf{Ext}}\ \mathsf{x12\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x13} : \mathsf{Ext}}\ \mathsf{x13\text{-}I}}{\mathsf{switch}(\mathsf{Alice}, \mathsf{x12}, \mathsf{x13}) = \mathsf{x12} : \mathsf{Ext}}\ \mathsf{Em\text{-}Eq}_1 \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x13} : \mathsf{Ext}}\ \mathsf{x13\text{-}I}}{\mathsf{x13} : \mathsf{Ext}}\ \mathsf{Sub}_2$$

As a final example, let us construct a switch over the boolean values, which assigns to each boolean value its opposite value. Like this:

```
switch on $(x : \mathbb{B})$ :
    case 1 (if x is tt) : return ff : $\mathbb{B}$
    case 2 (if x is ff) : return tt : $\mathbb{B}$
```

From that, we can generate the elimination and computation rules. Here is the elimination rule:

$$\frac{\mathsf{x} : \mathbb{B} \qquad \mathsf{ff} : \mathbb{B} \qquad \mathsf{tt} : \mathbb{B}}{\mathsf{switch}(\mathsf{x}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\ \mathbb{B}\mathsf{\text{-}E}$$

Here are the computation rules:

$$\frac{\mathsf{tt} : \mathbb{B} \qquad \mathsf{ff} : \mathbb{B} \qquad \mathsf{tt} : \mathbb{B}}{\mathsf{switch}(\mathsf{tt}, \mathsf{ff}, \mathsf{tt}) = \mathsf{ff} : \mathbb{B}}\ \mathbb{B}\mathsf{\text{-}Eq}_1$$

$$\frac{\mathsf{ff} : \mathbb{B} \qquad \mathsf{ff} : \mathbb{B} \qquad \mathsf{tt} : \mathbb{B}}{\mathsf{switch}(\mathsf{ff}, \mathsf{ff}, \mathsf{tt}) = \mathsf{tt} : \mathbb{B}}\ \mathbb{B}\mathsf{\text{-}Eq}_2$$

With that, we can now show that, if we start with tt, we get its opposite value, namely ff:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{tt} : \mathbb{B}}\,\mathsf{tt}\text{-}\mathsf{I} \quad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{ff} : \mathbb{B}}\,\mathsf{ff}\text{-}\mathsf{I} \quad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{tt} : \mathbb{B}}\,\mathsf{tt}\text{-}\mathsf{I}}{\mathsf{switch}(\mathsf{tt}, \mathsf{ff}, \mathsf{tt}) = \mathsf{ff} : \mathbb{B}}\,\mathbb{B}\text{-}\mathsf{Eq}_2 \quad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{ff} : \mathbb{B}}\,\mathsf{ff}\text{-}\mathsf{I}}{\mathsf{ff} : \mathbb{B}}\,\mathsf{Sub}_2$$

11 Abstraction and application

We can form functions and then compute with them by using abstraction and application from the typed lambda calculus. Abstraction and application formalize the process of rewriting strings and symbols.

11.1 Abstraction and application.

Suppose that every week, I purchase 3 lbs of dried corn. This week, the cost is 15$ per pound. I might jot down the formula on a piece of paper:

$$\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD} : \mathsf{Cost}$$

This expression tells me that multiplying 3 (which is a unit of type lbs) by 15 (which is a unit of type USD) will give me the total cost (so the resulting value is of type Cost).

Suppose now that the price changes each week. To remind me how to perform the new calculation each week, I could mark "15" as a value that should be replaced by a new USD value:

$$\text{Replace } \mathsf{15} \text{ with new } \mathsf{USD} \text{ value in: } (\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD})$$

What is the type of this? This expression does not give me a Cost. Rather it tells me how to calculate a Cost, given a USD. So it is a function from USD to Cost:

$$\text{Replace } \mathsf{15} \text{ with new } \mathsf{USD} \text{ value in: } (\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD}) : \mathsf{USD} \to \mathsf{Cost}$$

Then, each week, I could specify what new value I should replace it with. For instance, if the price per pound becomes 17 USD, I would write:

$$\text{Replace } \mathsf{15} \text{ with new } \mathsf{USD} \text{ value } \mathsf{17} \text{ in: } (\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD})$$

What is the type of this? Well, now I have stipulated a new USD value to use, so this will give me an actual Cost:

$$\text{Replace } \mathsf{15} \text{ with new } \mathsf{USD} \text{ value } \mathsf{17} \text{ in: } (\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD}) : \mathsf{Cost}$$

If I go in and actually do the replacement, the rewritten formula then becomes:

$$\mathsf{3\ lbs} \ \times \ \mathsf{17\ USD} : \mathsf{Cost}$$

The typed lambda calculus formalizes this process. Marking a symbol as replaceable is called abstraction. I will mark the replaceable symbol with a λ (e.g., $\lambda 15$), and specify its type (e.g., $\lambda 15 : \$$). Hence, this:

$$\text{Replace } \mathsf{15} \text{ with new } \mathsf{USD} \text{ value in: } (\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD}) : \mathsf{USD} \rightarrow \mathsf{Cost}$$

becomes this:

$$\lambda \mathsf{15} : \mathsf{USD}.(\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD}) : \mathsf{USD} \rightarrow \mathsf{Cost}$$

Specifying a replacement value is called application. Often the specified replacement is written after the original formula. In that. case, this:

$$\text{Replace } \mathsf{15} \text{ with new } \mathsf{USD} \text{ value } \mathsf{17} \text{ in: } (\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD}) : \mathsf{Cost}$$

becomes this:

$$\lambda \mathsf{15} : \mathsf{USD}.(\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD}) \ \mathsf{17} : \mathsf{Cost}$$

However, to make the application more visible, I will write it like this:

$$\mathsf{app}(\lambda \mathsf{15} : \mathsf{USD}.(\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD}), \mathsf{17}) : \mathsf{Cost}$$

The process of actually rewriting the original expression with the new value is called β-reduction. I will write it with a squiggly arrow, like this:

$$\mathsf{app}(\lambda \mathsf{15} : \mathsf{USD}.(\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD}), \mathsf{17}) : \mathsf{Cost} \quad \rightsquigarrow \quad \mathsf{3\ lbs} \ \times \ \mathsf{17\ USD} : \mathsf{Cost}$$

I will also sometimes write it like this:

$$(\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD})[\mathsf{15} := \mathsf{17}] : \mathsf{Cost}$$

Writing "$[\mathsf{15} := \mathsf{17}]$" after "$(\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD})$" denotes the expression that we get after we replace every occurrence of "$\mathsf{15}$" with "$\mathsf{17}$" in "$(\mathsf{3\ lbs} \ \times \ \mathsf{15\ USD})$".

11.2 Abstraction and application rules.

Suppose I have a derivation tree that results in some judgment. To use some placeholders, let us suppose that we obtain the judgment that some object b is of type B:

$$\begin{array}{c} \vdots \\ \mathsf{b : B} \end{array}$$

Suppose also that at the top of one of the leaves of the derivation tree, there is an inhabitant of some other type. To use placeholders again, suppose one of the leaves starts with the object x of type A:

$$\begin{array}{c} \mathsf{x : A} \\ \vdots \\ \mathsf{b : B} \end{array}$$

If I am in this situation, then I can mark x as replaceable, using lambda abstraction:

$$\frac{\begin{array}{c} \mathsf{x : A} \\ \vdots \\ \mathsf{b : B} \end{array}}{\mathsf{\lambda x : A.b : A \to B}}$$

This forms a template for a λ introduction rule, which I will denote as $\lambda\text{-}\mathsf{I}$:

$$\frac{\begin{array}{c} \mathsf{x : A} \\ \vdots \\ \mathsf{b : B} \end{array}}{\mathsf{\lambda x : A.b : A \to B}}\ \lambda\text{-}\mathsf{I}$$

To indicate that we have bound $\mathsf{x : A}$ to the new lambda expression, I will mark the bound expression with a unique label (I will just use numbers):

$$\frac{\begin{array}{c} \mathsf{x : A}^1 \\ \vdots \\ \mathsf{b : B} \end{array}}{\mathsf{\lambda x : A.b : A \to B}}\ \lambda\text{-}\mathsf{I}^1$$

Of course, this is only going to be valid if A and B are valid types in the system, and if x is a valid inhabitant of A. So we have a formation rule that tells us when $\mathsf{A \to B}$ is a valid type:

$$\frac{\mathsf{A : Type} \quad \mathsf{B : Type} \quad \mathsf{x : A}}{\mathsf{A \to B : Type}}\ \lambda\text{-}\mathsf{F}$$

If I have an abstraction, I can apply it to a value. This allows us to from a λ elimination rule:

$$\frac{\mathsf{\lambda x : A.b : A \to B} \quad \mathsf{x : A}}{\mathsf{app(\lambda x : A.b, a) : B}}\ \lambda\text{-}\mathsf{E}$$

And β-reduction gives us a computation rule:

$$\dfrac{\lambda x : A.b : A \to B \qquad x : A}{\mathsf{app}(\lambda x : A.b, a) = b[x := a] : B}\ \lambda\text{-Eq}$$

If we have an application like this, we can then us substitution to get the computed value:

$$\dfrac{\dfrac{\lambda x : A.b : A \to B \qquad x : A}{\mathsf{app}(\lambda x : A.b, a) = b[x := a] : B}\ \lambda\text{-Eq} \qquad b : B}{b[x := a] : B}\ \mathsf{Sub}_2$$

Here is a full derivation that combines the λ-I, λ-E, and Sub_2 rules:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{\dfrac{}{A : \mathsf{Type}}\ \text{A-F}}{x : A^1}\ \text{x-I} \\ \vdots \\ b : B\end{array}}{\lambda x : A.b : A \to B}\ \lambda\text{-I} \qquad \dfrac{\dfrac{}{A : \mathsf{Type}}\ \text{A-F}}{x : A}\ \text{x-I}}{\mathsf{app}(\lambda x : A.b, a) = b[x := a] : B}\ \lambda\text{-Eq} \qquad \dfrac{\dfrac{}{B : \mathsf{Type}}\ \text{B-F}}{b : B}\ \text{b-I}}{b[x := a] : B}\ \mathsf{Sub}_2$$

11.3 Examples.

Consider the derivation from above which switches boolean values:

$$\dfrac{\dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\text{-F}}{\mathsf{tt} : \mathbb{B}}\ \text{tt-I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\text{-F}}{\mathsf{ff} : \mathbb{B}}\ \text{ff-I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\text{-F}}{\mathsf{tt} : \mathbb{B}}\ \text{tt-I}}{\mathsf{switch}(\mathsf{tt}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\ \mathbb{B}\text{-E}$$

This is a useful derivation, and we might want to use it for the other boolean value. We could do that by copying the whole thing, and then replacing every instance of the assumption tt : $\mathbb{B}$ that appears at the top left with the other value of $\mathbb{B}$, namely ff : $\mathbb{B}$.

But we can also just use abstraction and mark that top left assumption as replaceable. So, let's mark every occurrence of the term tt : $\mathbb{B}$ from the top left as an x:

$$\dfrac{\dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\text{-F}}{x : \mathbb{B}}\ \text{x-I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\text{-F}}{\mathsf{ff} : \mathbb{B}}\ \text{ff-I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\text{-F}}{\mathsf{tt} : \mathbb{B}}\ \text{tt-I}}{\mathsf{switch}(x, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\ \mathbb{B}\text{-E}$$

This way we can clearly see which occurrences are replaceable. Now, let's use the λ-I rule to do generate the abstraction:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{x} : \mathbb{B}^1}\,\mathsf{x\text{-}I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{ff} : \mathbb{B}}\,\mathsf{ff\text{-}I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{tt} : \mathbb{B}}\,\mathsf{tt\text{-}I}}{\mathsf{switch}(\mathsf{x}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\,\mathbb{B}\text{-}\mathsf{E}}{\lambda \mathsf{x} : \mathbb{B}.\mathsf{switch}(\mathsf{x}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B} \to \mathbb{B}}\,\lambda\text{-}\mathsf{I}^1$$

Now we have a function that can take any value of type $\mathbb{B}$, and put it into the switch. To save space, let us use the term g as an abbreviation for the full lambda expression:

$$\mathsf{g} \equiv \lambda \mathsf{x} : \mathbb{B}.\mathsf{switch}(\mathsf{x}, \mathsf{ff}, \mathsf{tt})$$

The derivation is now:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{x} : \mathbb{B}^1}\,\mathsf{x\text{-}I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{ff} : \mathbb{B}}\,\mathsf{ff\text{-}I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{tt} : \mathbb{B}}\,\mathsf{tt\text{-}I}}{\mathsf{switch}(\mathsf{x}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\,\mathbb{B}\text{-}\mathsf{E}}{\mathsf{g} : \mathbb{B} \to \mathbb{B}}\,\lambda\text{-}\mathsf{I}^1$$

Let us apply the function to ff:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{x} : \mathbb{B}^1}\,\mathsf{x\text{-}I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{ff} : \mathbb{B}}\,\mathsf{ff\text{-}I} \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{tt} : \mathbb{B}}\,\mathsf{tt\text{-}I}}{\mathsf{switch}(\mathsf{x}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\,\mathbb{B}\text{-}\mathsf{E}}{\mathsf{g} : \mathbb{B} \to \mathbb{B}}\,\lambda\text{-}\mathsf{I}^1 \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\,\mathbb{B}\text{-}\mathsf{F}}{\mathsf{ff} : \mathbb{B}}\,\mathsf{ff\text{-}I}}{\mathsf{app}(\mathsf{g}, \mathsf{ff}) = \mathsf{switch}(\mathsf{ff}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\,\lambda\text{-}\mathsf{Eq}$$

We can then use substitution:

$$\dfrac{\dfrac{\vdots}{\mathsf{app}(\mathsf{g}, \mathsf{ff}) = \mathsf{switch}(\mathsf{ff}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\,\lambda\text{-}\mathsf{Eq} \qquad \dfrac{\vdots}{\mathsf{switch}(\mathsf{ff}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\,\mathbb{B}\text{-}\mathsf{E}}{\mathsf{switch}(\mathsf{ff}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\,\mathsf{Sub}_2$$

And we can use the computation rules to get a value:

$$\dfrac{\dfrac{\vdots}{\mathsf{switch}(\mathsf{ff}, \mathsf{ff}, \mathsf{tt}) : \mathbb{B}}\,\mathsf{Sub}_2}{\mathsf{switch}(\mathsf{ff}, \mathsf{ff}, \mathsf{tt}) = \mathsf{tt} : \mathbb{B}}\,\mathbb{B}\text{-}\mathsf{Eq}_2$$

Finally, we can use substitution again to get the final result:

$$\dfrac{\dfrac{\dfrac{\vdots}{\mathsf{switch}(\mathsf{ff},\mathsf{ff},\mathsf{tt}) : \mathbb{B}}\ \mathsf{Sub}_2}{\mathsf{switch}(\mathsf{ff},\mathsf{ff},\mathsf{tt}) = \mathsf{tt} : \mathbb{B}}\ \mathbb{B}\text{-}\mathsf{Eq}_2 \qquad \dfrac{\dfrac{}{\mathbb{B} : \mathsf{Type}}\ \mathbb{B}\text{-}\mathsf{F}}{\mathsf{tt} : \mathbb{B}}\ \mathsf{tt}\text{-}\mathsf{I}}{\mathsf{tt} : \mathbb{B}}\ \mathsf{Sub}_2$$

We could use the same technique to construct functions for any of the switches we built earlier. For example:

$$\dfrac{\dfrac{\begin{matrix}\mathsf{x} : \mathsf{Em}^1 \\ \vdots\end{matrix}}{\mathsf{switch}(\mathsf{x},\mathsf{x11},\mathsf{x12}) : \mathsf{Ext}}\ \mathsf{Em}\text{-}\mathsf{E}}{\lambda\mathsf{x} : \mathsf{Em}.\mathsf{switch}(\mathsf{x},\mathsf{x11},\mathsf{x12}) : \mathsf{Em} \to \mathsf{Ext}}\ \lambda\text{-}\mathsf{I}^1$$

11.4 Function space.

Arrow types with the shape $\mathsf{A} \to \mathsf{B}$ are function types. The inhabitants of the type $\mathsf{A} \to \mathsf{B}$ are all functions that map elements from A to elements of B.

For example, above we constructed a number of switches, and if we use the λ-I rule on each one, we get a function that maps employees to extensions. Consider the last example:

$$\lambda\mathsf{x} : \mathsf{Em}.\mathsf{switch}(\mathsf{x},\mathsf{x11},\mathsf{x12}) : \mathsf{Em} \to \mathsf{Ext}$$

We can write this function in the more traditional way. Let us assign the name h to the function:

$$\mathsf{h} \equiv \lambda\mathsf{x} : \mathsf{Em}.\mathsf{switch}(\mathsf{x},\mathsf{x11},\mathsf{x12})$$

Then we can write this function out in a more traditional way:

$$\mathsf{h}(\mathsf{x}) = \begin{cases} \mathsf{x11}, & \text{when } \mathsf{x} = \mathsf{Alice} \\ \mathsf{x12}, & \text{when } \mathsf{x} = \mathsf{Bob} \end{cases}$$

Or, we could write the mapping like this:

$$\begin{aligned} \mathsf{Alice} &\longrightarrow \mathsf{x11} \\ \mathsf{Bob} &\longrightarrow \mathsf{x12} \end{aligned}$$

Another switch we could define is this:

$$\dfrac{\mathsf{x} : \mathsf{Em} \quad \mathsf{x12} : \mathsf{Ext} \quad \mathsf{x13} : \mathsf{Ext}}{\mathsf{switch}(\mathsf{x},\mathsf{x12},\mathsf{x13}) : \mathsf{Ext}}\ \mathsf{Em}\text{-}\mathsf{E}$$

We could form a function from this too, by abstracting over x with the λ-I rule:

$$\dfrac{\dfrac{\mathsf{x} : \mathsf{Em}^1 \quad \mathsf{x12} : \mathsf{Ext} \quad \mathsf{x13} : \mathsf{Ext}}{\mathsf{switch}(\mathsf{x}, \mathsf{x12}, \mathsf{x13}) : \mathsf{Ext}}\ \mathsf{Em\text{-}E}}{\lambda \mathsf{x} : \mathsf{Em}.\mathsf{switch}(\mathsf{x}, \mathsf{x12}, \mathsf{x13}) : \mathsf{Em} \to \mathsf{Ext}}\ \lambda\text{-}\mathsf{I}^1$$

We could call this function j:

$$\mathsf{j} \equiv \lambda \mathsf{x} : \mathsf{Em}.\mathsf{switch}(\mathsf{x}, \mathsf{x12}, \mathsf{x13})$$

We could write it out in a more traditional way:

$$\mathsf{j}(\mathsf{x}) = \begin{cases} \mathsf{x12}, & \text{when } \mathsf{x} = \mathsf{Alice} \\ \mathsf{x13}, & \text{when } \mathsf{x} = \mathsf{Bob} \end{cases}$$

Or, we could write the mapping like this:

$$\begin{aligned} \mathsf{Alice} &\longrightarrow \mathsf{x12} \\ \mathsf{Bob} &\longrightarrow \mathsf{x13} \end{aligned}$$

All of these mappings are mappings from employees Em to extensions Ext. So all of these examples are inhabitants of the type $\mathsf{Em} \to \mathsf{Ext}$.

11.5 Boolean function space.

Above, we defined a function over the booleans, which we named g:

$$\mathsf{g} \equiv \lambda \mathsf{x} : \mathbb{B}.\mathsf{switch}(\mathsf{x}, \mathsf{ff}, \mathsf{tt})$$

This is a function from $\mathbb{B}$ to $\mathbb{B}$. Here is the mapping:

$$\begin{aligned} \mathsf{tt} &\longrightarrow \mathsf{ff} \\ \mathsf{ff} &\longrightarrow \mathsf{tt} \end{aligned}$$

Using the same techniques that we used to define that function, we could define a different function:

$$\dfrac{\dfrac{\mathsf{x} : \mathbb{B}^1 \quad \mathsf{tt} : \mathbb{B} \quad \mathsf{ff} : \mathbb{B}}{\mathsf{switch}(\mathsf{x}, \mathsf{tt}, \mathsf{ff}) : \mathbb{B}}\ \mathbb{B}\text{-}\mathsf{E}}{\lambda \mathsf{x} : \mathbb{B}.\mathsf{switch}(\mathsf{x}, \mathsf{tt}, \mathsf{ff}) : \mathbb{B} \to \mathbb{B}}\ \lambda\text{-}\mathsf{I}^1$$

This is also a function from $\mathbb{B}$ to $\mathbb{B}$. Here is its mapping:

$$\begin{aligned} \mathsf{tt} &\longrightarrow \mathsf{tt} \\ \mathsf{ff} &\longrightarrow \mathsf{ff} \end{aligned}$$

Another option would be this:

$$\dfrac{\dfrac{\mathsf{x} : \mathbb{B}^1 \quad \mathsf{tt} : \mathbb{B} \quad \mathsf{tt} : \mathbb{B}}{\mathsf{switch}(\mathsf{x}, \mathsf{tt}, \mathsf{tt}) : \mathbb{B}}\ \mathbb{B}\text{-}\mathsf{E}}{\lambda \mathsf{x} : \mathbb{B}.\mathsf{switch}(\mathsf{x}, \mathsf{tt}, \mathsf{tt}) : \mathbb{B} \to \mathbb{B}}\ \lambda\text{-}\mathsf{I}^1$$

This too is a function from $\mathbb{B}$ to $\mathbb{B}$. Here is the mapping:

$$\mathsf{tt} \longrightarrow \mathsf{tt}$$
$$\mathsf{ff} \longrightarrow \mathsf{tt}$$

The type $\mathbb{B} \to \mathbb{B}$ is the type of all functions from $\mathbb{B}$ to $\mathbb{B}$. All of these examples are inhabitants.

12 Pairs

In CTT, we can build more complex types from simpler types. Arrow types are one example of this. Another example is a 2-tuple, i.e., a pair. To form a pair type, we take two other types, and we put them together. We write a pair type like this:

$$\mathsf{A} \times \mathsf{B}$$

That indicates the type of all pairs whose first component is an element of type A and whose second component is an element of type B.

Of course, this requires that A and B are legal types in the system. Here is the formation rule:

$$\dfrac{\mathsf{A} : \mathsf{Type} \quad \mathsf{B} : \mathsf{Type}}{\mathsf{A} \times \mathsf{B} : \mathsf{Type}}\ \times\text{-}\mathsf{F}$$

To construct an element of this type, we need an inhabitant of A, and an inhabitant of B. Here is the introduction rule:

$$\dfrac{\mathsf{a} : \mathsf{A} \quad \mathsf{b} : \mathsf{B}}{\langle \mathsf{a}, \mathsf{b} \rangle : \mathsf{A} \times \mathsf{B}}\ \times\text{-}\mathsf{I}$$

We could follow a similar pattern and define 3-tuples, 4-tuples, and so on. For example, the formation for 3-tuples would be:

$$\dfrac{\mathsf{A} : \mathsf{Type} \quad \mathsf{B} : \mathsf{Type} \quad \mathsf{C} : \mathsf{Type}}{\mathsf{A} \times \mathsf{B} \times \mathsf{C} : \mathsf{Type}}$$

12.1 Examples.

We can form pairs of, say, employees and phone extensions, $\mathsf{Em} \times \mathsf{Ext}$. Here is the derivation showing that this is a legal type in the system:

$$\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F} \qquad \dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{Em} \times \mathsf{Ext}}\ \times\text{-}\mathsf{F}$$

We can then construct any pair of employees and phone extensions. Here are two such examples:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}}{\mathsf{Alice} : \mathsf{Em}}\ \mathsf{Alice\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x11} : \mathsf{Ext}}\ \mathsf{x11\text{-}I}}{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext}}\ \times\text{-}\mathsf{I} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}}{\mathsf{Bob} : \mathsf{Em}}\ \mathsf{Bob\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x12} : \mathsf{Ext}}\ \mathsf{x12\text{-}I}}{\langle \mathsf{Bob}, \mathsf{x12} \rangle : \mathsf{Em} \times \mathsf{Ext}}\ \times\text{-}\mathsf{I}$$

We can even form pairs of employees, i.e., of type $\mathsf{Em} \times \mathsf{Em}$. For example:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}}{\mathsf{Alice} : \mathsf{Em}}\ \mathsf{Alice\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}}{\mathsf{Bob} : \mathsf{Em}}\ \mathsf{Bob\text{-}I}}{\langle \mathsf{Alice}, \mathsf{Bob} \rangle : \mathsf{Em} \times \mathsf{Em}}\ \times\text{-}\mathsf{I} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}}{\mathsf{Bob} : \mathsf{Em}}\ \mathsf{Bob\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}}{\mathsf{Bob} : \mathsf{Ext}}\ \mathsf{Bob\text{-}I}}{\langle \mathsf{Bob}, \mathsf{Bob} \rangle : \mathsf{Em} \times \mathsf{Em}}\ \times\text{-}\mathsf{I}$$

13 Pair elimination

Elimination rules tell us how to iterate over every inhabitant of a type and construct some further value that is based on it. When it comes to pairs, any object can be constructed from that pair, simply by using the individual components of the pair to construct it.

Suppose we have an object c of type C. Suppose also that, when we constructed c, we used an object x of type A and an object b of type B. Let us represent that like this:

$$\begin{array}{c} \mathsf{x} : \mathsf{A}, \mathsf{y} : \mathsf{B} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{array}$$

Suppose now that we also have a pair, $\langle \mathsf{a}, \mathsf{b} \rangle : \mathsf{A} \times \mathsf{B}$:

$$\langle \mathsf{a}, \mathsf{b} \rangle : \mathsf{A} \times \mathsf{B} \qquad \begin{array}{c} \mathsf{x} : \mathsf{A}, \mathsf{y} : \mathsf{B} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{array}$$

We can reconstruct c by using a in place of x, and b in place of y. To denote the result of rebuilding c with the components of the pair $\langle \mathsf{a}, \mathsf{b} \rangle$, let us write this:

$$\mathsf{elim}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{c})$$

When we rebuild c, we still end up with an object of type C. So the type of this expression is C:

$$\mathsf{elim}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{c}) : \mathsf{C}$$

So, we can say that, if we have the pair $\langle \mathsf{a}, \mathsf{b}\rangle$, and if we have c, we can rebuild c using a and b:

$$\dfrac{\langle \mathsf{a}, \mathsf{b}\rangle : \mathsf{A} \times \mathsf{B} \qquad \begin{array}{c} \mathsf{x} : \mathsf{A}, \mathsf{y} : \mathsf{B} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{array}}{\mathsf{elim}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{c}) : \mathsf{C}}\ \times\text{-}\mathsf{E}$$

This is the elimination rule for pairs.

The computation rule specifies what the new value is. To get the new value, we take c, and we substitute a and b in for x and y:

$$\dfrac{\langle \mathsf{a}, \mathsf{b}\rangle : \mathsf{A} \times \mathsf{B} \qquad \begin{array}{c} \mathsf{x} : \mathsf{A}, \mathsf{y} : \mathsf{B} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{array}}{\mathsf{elim}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{c}) = \mathsf{c}[\mathsf{x} := \mathsf{a}, \mathsf{y} := \mathsf{b}] : \mathsf{C}}\ \times\text{-}\mathsf{Eq}$$

13.1 Examples.

Let us assign Alice to phone extension x11, by pairing Alice and x11:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}\ \mathsf{Em\text{-}F}}{\mathsf{Alice} : \mathsf{Em}}\ \mathsf{Alice\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x11} : \mathsf{Ext}}\ \mathsf{x11\text{-}I}}{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext}}\ \times\text{-}\mathsf{I}$$

Let us drop the labels to save space (it should be obvious which rules are used at each step):

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x11} : \mathsf{Ext}}}{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext}}$$

Now let us assign Alice and her extension to an office, by pairing the previous pair with, say, room r01:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x11} : \mathsf{Ext}}}{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \langle \mathsf{Alice}, \mathsf{x11}\rangle, \mathsf{r01}\rangle : (\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}}$$

Here we have constructed a new object — namely, $\langle\langle \mathsf{Alice}, \mathsf{x11}\rangle, \mathsf{r01}\rangle$ — which inhabits a type — $(\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}$.

Notice, though, that we constructed this object using the pair $\langle \mathsf{Alice}, \mathsf{x11}\rangle$. We could reconstruct this element, using a different pair of elements from Em and Ext. For example, suppose we have the pair $\langle \mathsf{Bob}, \mathsf{x12}\rangle$:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12} \rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x11} : \mathsf{Ext}}}{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext}} \quad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{r01} \rangle : (\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}}$$

We can apply the elimination rule here, since we are in a situation that matches the elimination rule template. In this case, a is Bob, b is $\mathsf{x12}$, x is Alice, y is $\mathsf{x11}$, c is $\langle\langle \mathsf{Alice}, \mathsf{x11}\rangle, \mathsf{r01}\rangle$, and C is $(\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}$.

Here we apply the elimination rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12} \rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x11} : \mathsf{Ext}}}{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext}} \quad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{r01} \rangle : (\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{x12} \rangle, \langle\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{r01} \rangle) : (\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}} \ \times\text{-}\mathsf{E}$$

We can use the computation rule to get new object reconstructed with Bob and $\mathsf{x12}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12} \rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x11} : \mathsf{Ext}}}{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext}} \quad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{r01} \rangle : (\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{x12} \rangle, \langle\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{r01} \rangle) = \langle\langle \mathsf{Bob}, \mathsf{x12} \rangle, \mathsf{r01} \rangle : (\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}} \ \times\text{-}\mathsf{Eq}$$

We can then use substitution to extract just the new object:

$$\dfrac{\dfrac{\vdots}{\mathsf{elim}(\ldots) = \langle\langle \mathsf{Bob}, \mathsf{x12} \rangle, \mathsf{r01} \rangle : \ldots} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12} \rangle : \mathsf{Em} \times \mathsf{Ext}} \quad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle\langle \mathsf{Bob}, \mathsf{x12} \rangle, \mathsf{r01} \rangle : (\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}}}{\langle\langle \mathsf{Bob}, \mathsf{x12} \rangle, \mathsf{r01} \rangle : (\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}} \ \mathsf{Sub}_2$$

In this example, we built a type (namely, $(\mathsf{Em} \times \mathsf{Ext}) \times \mathsf{Rm}$) that used an element from Em and Ext (namely Alice and $\mathsf{x11}$). Then we used the pair elimination rule to rebuild that object using a different pair of elements from Em and Ext (namely Bob and $\mathsf{x12}$).

However, we do not need to use elements from both Em and Ext in the elimination rule. For example, suppose we build a type that only uses Em. Suppose we pair up employees with offices, and we assign Alice to office $\mathsf{r01}$:

$$\dfrac{\dfrac{\overline{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\overline{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \mathsf{Alice}, \mathsf{r01} \rangle : \mathsf{Em} \times \mathsf{Rm}}$$

Suppose we also have the pair $\langle \mathsf{Bob}, \mathsf{x12} \rangle$ again:

$$\dfrac{\dfrac{\overline{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \qquad \dfrac{\overline{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12} \rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{\overline{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\overline{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \mathsf{Alice}, \mathsf{r01} \rangle : \mathsf{Em} \times \mathsf{Rm}}$$

We can apply the pair elimination rule in this case, if we like, since we are again in a situation that matches the elimination rule template. Here, a is Bob, b is $\mathsf{x12}$, x is Alice, y is not used, c is $\langle \mathsf{Alice}, \mathsf{r01} \rangle$, and C is $\mathsf{Em} \times \mathsf{Rm}$. Here is the elimination rule:

$$\dfrac{\dfrac{\dfrac{\overline{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \qquad \dfrac{\overline{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12} \rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{\overline{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\overline{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \mathsf{Alice}, \mathsf{r01} \rangle : \mathsf{Em} \times \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{x12} \rangle, \langle \mathsf{Alice}, \mathsf{r01} \rangle) : \mathsf{Em} \times \mathsf{Rm}} \times\text{-}\mathsf{E}$$

Notice that y from the elimination rule template is not used here to build c (which is $\langle \mathsf{Alice}, \mathsf{r01} \rangle$ in this case). We only use x (which is Alice in this case) to build c. So y from the rule template plays no role here.

We can go on and use the computation rule:

$$\dfrac{\dfrac{\dfrac{\overline{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \qquad \dfrac{\overline{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12} \rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{\overline{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\overline{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \mathsf{Alice}, \mathsf{r01} \rangle : \mathsf{Em} \times \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{x12} \rangle, \langle \mathsf{Alice}, \mathsf{r01} \rangle) = \langle \mathsf{Bob}, \mathsf{r01} \rangle : \mathsf{Em} \times \mathsf{Rm}} \times\text{-}\mathsf{Eq}$$

And then we could use the substitution rule to extract just the pair $\langle \mathsf{Bob}, \mathsf{r01} \rangle$:

$$\dfrac{\dfrac{\vdots}{\mathsf{elim}(\ldots) = \langle \mathsf{Bob}, \mathsf{r01}\rangle : \mathsf{Em} \times \mathsf{Rm}}\,\times\text{-}\mathsf{Eq} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \mathsf{Bob}, \mathsf{r01}\rangle : \mathsf{Em} \times \mathsf{Rm}}}{\langle \mathsf{Bob}, \mathsf{r01}\rangle : \mathsf{Em} \times \mathsf{Rm}}\,\mathsf{Sub}_2$$

In that example, we did not use y from the elimination rule template. But we do not need to use x either. For example, suppose that in place of c of type C from the elimination rule template, we simply introduce room $\mathsf{r02}$ of type Rm:

$$\dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r02} : \mathsf{Rm}}$$

Here we have used no elements from Em or Ext to construct $\mathsf{r02}$. Suppose, then, that we have the pair $\langle \mathsf{Bob}, \mathsf{x12}\rangle$:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12}\rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r02} : \mathsf{Rm}}$$

We can apply the pair elimination rule here too. In this case, a is Bob, b is $\mathsf{x12}$, c is $\mathsf{r01}$, and C is Rm, while x and y simply do not appear. So, let us apply the elimination rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12}\rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r02} : \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{x12}\rangle, \mathsf{r02}) : \mathsf{Rm}}\,\times\text{-}\mathsf{E}$$

We can use the computation rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12}\rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r02} : \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{x12}\rangle, \mathsf{r02}) = \mathsf{r02} : \mathsf{Rm}}\,\times\text{-}\mathsf{Eq}$$

And we can use elimination to extract the result:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \quad \dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}}{\mathsf{x12} : \mathsf{Ext}}}{\langle \mathsf{Bob}, \mathsf{x12}\rangle : \mathsf{Em} \times \mathsf{Ext}} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r02} : \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{x12}\rangle, \mathsf{r02}) = \mathsf{r02} : \mathsf{Rm}}\,\times\text{-}\mathsf{Eq} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r02} : \mathsf{Rm}}}{\mathsf{r02} : \mathsf{Rm}}\,\mathsf{Sub}_2$$

13.2 Projections.

Consider the pair elimination and computation rules again:

$$\frac{\langle a, b\rangle : A \times B \qquad \begin{array}{c} x : A, y : B \\ \vdots \\ c : C \end{array}}{elim(\langle a, b\rangle, c) : C}\ \times\text{-E}$$

$$\frac{\langle a, b\rangle : A \times B \qquad \begin{array}{c} x : A, y : B \\ \vdots \\ c : C \end{array}}{elim(\langle a, b\rangle, c) = c[x := a, y := b] : C}\ \times\text{-Eq}$$

These rules tell us how to build a further element C, from the pair $\langle a, b\rangle$. There is no reason that we cannot let C be the type of one of the pair's components, i.e., A or B.

For example, let C be A. If we substitute A in for C, the rules now look like this:

$$\frac{\langle a, b\rangle : A \times B \qquad \begin{array}{c} x : A, y : B \\ \vdots \\ c : A \end{array}}{elim(\langle a, b\rangle, c) : A}\ \times\text{-E}$$

$$\frac{\langle a, b\rangle : A \times B \qquad \begin{array}{c} x : A, y : B \\ \vdots \\ c : A \end{array}}{elim(\langle a, b\rangle, c) = c[x := a, y := b] : A}\ \times\text{-Eq}$$

The object c is now not some further object from another type. It is just the object of type A that we started with, namely x. So we can replace every occurrence of c with x:

$$\frac{\langle a, b\rangle : A \times B \qquad \begin{array}{c} x : A, y : B \\ \vdots \\ x : A \end{array}}{elim(\langle a, b\rangle, x) : A}\ \times\text{-E}$$

$$\frac{\langle a, b\rangle : A \times B \qquad \begin{array}{c} x : A, y : B \\ \vdots \\ x : A \end{array}}{elim(\langle a, b\rangle, x) = c[x := a, y := b] : A}\ \times\text{-Eq}$$

Of course, $y : B$ is not used here, so we can erase it (and the substitution of $b : B$), to make the rules simpler still:

$$\frac{\langle a, b\rangle : A \times B \qquad \begin{array}{c} x : A \\ \vdots \\ x : A \end{array}}{\mathsf{elim}(\langle a, b\rangle, x) : A}\ \times\text{-E}$$

$$\frac{\langle a, b\rangle : A \times B \qquad \begin{array}{c} x : A \\ \vdots \\ x : A \end{array}}{\mathsf{elim}(\langle a, b\rangle, x) = x[x := a] : A}\ \times\text{-Eq}$$

It is now redundant to show the vertical dots from $x : A$ to $x : A$, so we can collapse them:

$$\frac{\langle a, b\rangle : A \times B \qquad x : A}{\mathsf{elim}(\langle a, b\rangle, x) : A}\ \times\text{-E}$$

$$\frac{\langle a, b\rangle : A \times B \qquad x : A}{\mathsf{elim}(\langle a, b\rangle, x) = x[x := a] : A}\ \times\text{-Eq}$$

And it is redundant to take $x : A$ and then immediately replace it with $a : A$. So we can remove that part of the rules to make them even simpler:

$$\frac{\langle a, b\rangle : A \times B}{\mathsf{elim}(\langle a, b\rangle, a) : A}\ \times\text{-E}$$

$$\frac{\langle a, b\rangle : A \times B}{\mathsf{elim}(\langle a, b\rangle, a) = a : A}\ \times\text{-Eq}$$

Finally, since $\mathsf{elim}(\langle a, b\rangle, a)$ always selects the first item from the pair, we can just call it $\mathsf{fst}(\langle a, b\rangle)$ (and give the rules new names to indicate this):

$$\frac{\langle a, b\rangle : A \times B}{\mathsf{fst}(\langle a, b\rangle) : A}\ \times\text{-E}_1$$

$$\frac{\langle a, b\rangle : A \times B}{\mathsf{fst}(\langle a, b\rangle) = a : A}\ \times\text{-Eq}_1$$

These two derived rules are named with a subscripted numeral one, because they let us extract the first component of a pair.

We can do something similar with the type B. If we take the original elimination and computation rules and set C to B, we can build rules that let us extract the second component of a pair:

$$\frac{\langle a, b\rangle : A \times B}{\mathsf{snd}(\langle a, b\rangle) : B}\ \times\text{-E}_2$$

$$\dfrac{\langle \mathsf{a}, \mathsf{b}\rangle : \mathsf{A} \times \mathsf{B}}{\mathsf{snd}(\langle \mathsf{a}, \mathsf{b}\rangle) = \mathsf{b} : \mathsf{B}}\ \times\text{-}\mathsf{Eq}_2$$

13.3 Examples.

Take the elimination and computation rules again:

$$\dfrac{\langle \mathsf{a}, \mathsf{b}\rangle : \mathsf{A} \times \mathsf{B} \qquad \begin{matrix} \mathsf{x} : \mathsf{A}, \mathsf{y} : \mathsf{B} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{matrix}}{\mathsf{elim}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{c}) : \mathsf{C}}\ \times\text{-}\mathsf{E}$$

$$\dfrac{\langle \mathsf{a}, \mathsf{b}\rangle : \mathsf{A} \times \mathsf{B} \qquad \begin{matrix} \mathsf{x} : \mathsf{A}, \mathsf{y} : \mathsf{B} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{matrix}}{\mathsf{elim}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{c}) = \mathsf{c}[\mathsf{x} := \mathsf{a}, \mathsf{y} := \mathsf{b}] : \mathsf{C}}\ \times\text{-}\mathsf{Eq}$$

Now replace A and B with Em and Ext:

$$\dfrac{\langle \mathsf{a}, \mathsf{b}\rangle : \mathsf{Em} \times \mathsf{Ext} \qquad \begin{matrix} \mathsf{x} : \mathsf{Em}, \mathsf{y} : \mathsf{Ext} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{matrix}}{\mathsf{elim}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{c}) : \mathsf{C}}\ \times\text{-}\mathsf{E}$$

$$\dfrac{\langle \mathsf{a}, \mathsf{b}\rangle : \mathsf{Em} \times \mathsf{Ext} \qquad \begin{matrix} \mathsf{x} : \mathsf{Em}, \mathsf{y} : \mathsf{Ext} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{matrix}}{\mathsf{elim}(\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{c}) = \mathsf{c}[\mathsf{x} := \mathsf{a}, \mathsf{y} := \mathsf{b}] : \mathsf{C}}\ \times\text{-}\mathsf{Eq}$$

Now replace a and b with Alice and x11:

$$\dfrac{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext} \qquad \begin{matrix} \mathsf{x} : \mathsf{Em}, \mathsf{y} : \mathsf{Ext} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{matrix}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11}\rangle, \mathsf{c}) : \mathsf{C}}\ \times\text{-}\mathsf{E}$$

$$\dfrac{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext} \qquad \begin{matrix} \mathsf{x} : \mathsf{Em}, \mathsf{y} : \mathsf{Ext} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{matrix}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11}\rangle, \mathsf{c}) = \mathsf{c}[\mathsf{x} := \mathsf{Alice}, \mathsf{y} := \mathsf{x11}] : \mathsf{C}}\ \times\text{-}\mathsf{Eq}$$

Replace C with Em:

$$\frac{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext} \qquad \begin{matrix} \mathsf{x} : \mathsf{Em}, \mathsf{y} : \mathsf{Ext} \\ \vdots \\ \mathsf{c} : \mathsf{Em} \end{matrix}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{c}) : \mathsf{Em}} \ \times\text{-}\mathsf{E}$$

$$\frac{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext} \qquad \begin{matrix} \mathsf{x} : \mathsf{Em}, \mathsf{y} : \mathsf{Ext} \\ \vdots \\ \mathsf{c} : \mathsf{Em} \end{matrix}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{c}) = \mathsf{c}[\mathsf{x} := \mathsf{Alice}, \mathsf{y} := \mathsf{x11}] : \mathsf{Em}} \ \times\text{-}\mathsf{Eq}$$

And replace c with x:

$$\frac{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext} \qquad \begin{matrix} \mathsf{x} : \mathsf{Em}, \mathsf{y} : \mathsf{Ext} \\ \vdots \\ \mathsf{x} : \mathsf{Em} \end{matrix}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{x}) : \mathsf{Em}} \ \times\text{-}\mathsf{E}$$

$$\frac{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext} \qquad \begin{matrix} \mathsf{x} : \mathsf{Em}, \mathsf{y} : \mathsf{Ext} \\ \vdots \\ \mathsf{x} : \mathsf{Em} \end{matrix}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{x}) = \mathsf{x}[\mathsf{x} := \mathsf{Alice}, \mathsf{y} := \mathsf{x11}] : \mathsf{Em}} \ \times\text{-}\mathsf{Eq}$$

We don't need y, so we can delete references to it:

$$\frac{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext} \quad \begin{matrix} \mathsf{x} : \mathsf{Em} \\ \vdots \\ \mathsf{x} : \mathsf{Em} \end{matrix}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{x}) : \mathsf{Em}} \ \times\text{-}\mathsf{E}$$

$$\frac{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext} \quad \begin{matrix} \mathsf{x} : \mathsf{Em} \\ \vdots \\ \mathsf{x} : \mathsf{Em} \end{matrix}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{x}) = \mathsf{x}[\mathsf{x} := \mathsf{Alice}] : \mathsf{Em}} \ \times\text{-}\mathsf{Eq}$$

And we don't need the vertical ellipses, so we can drop that too:

$$\frac{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext} \quad \mathsf{x} : \mathsf{Em}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{x}) : \mathsf{Em}} \ \times\text{-}\mathsf{E}$$

$$\frac{\langle \mathsf{Alice}, \mathsf{x11} \rangle : \mathsf{Em} \times \mathsf{Ext} \quad \mathsf{x} : \mathsf{Em}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11} \rangle, \mathsf{x}) = \mathsf{x}[\mathsf{x} := \mathsf{Alice}] : \mathsf{Em}} \ \times\text{-}\mathsf{Eq}$$

Finally, we don't need to introduce x and then replace it with Alice, so we can just go directly from x to Alice:

$$\dfrac{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11}\rangle, \mathsf{Alice}) : \mathsf{Em}}\ \times\text{-}\mathsf{E}$$

$$\dfrac{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext}}{\mathsf{elim}(\langle \mathsf{Alice}, \mathsf{x11}\rangle, \mathsf{Alice}) = \mathsf{Alice} : \mathsf{Em}}\ \times\text{-}\mathsf{Eq}$$

We can replace the $\mathsf{elim}()$ expression with $\mathsf{fst}()$:

$$\dfrac{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext}}{\mathsf{fst}(\langle \mathsf{Alice}, \mathsf{x11}\rangle) : \mathsf{Em}}\ \times\text{-}\mathsf{E}_1$$

$$\dfrac{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext}}{\mathsf{fst}(\langle \mathsf{Alice}, \mathsf{x11}\rangle) = \mathsf{Alice} : \mathsf{Em}}\ \times\text{-}\mathsf{Eq}_1$$

We could do the same to build rules that let us extract the second $\mathsf{snd}()$ component of the pair:

$$\dfrac{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext}}{\mathsf{snd}(\langle \mathsf{Alice}, \mathsf{x11}\rangle) : \mathsf{Ext}}\ \times\text{-}\mathsf{E}_2$$

$$\dfrac{\langle \mathsf{Alice}, \mathsf{x11}\rangle : \mathsf{Em} \times \mathsf{Ext}}{\mathsf{snd}(\langle \mathsf{Alice}, \mathsf{x11}\rangle) = \mathsf{x11} : \mathsf{Ext}}\ \times\text{-}\mathsf{Eq}_2$$

We can use these simplified $\mathsf{fst}()$ and $\mathsf{snd}()$ rules to extract the first and second components of any pairs. For example, suppose we pair up a phone extension with a room:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x13} : \mathsf{Ext}}\ \mathsf{x13\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}\ \mathsf{Rm\text{-}F}}{\mathsf{r02} : \mathsf{Rm}}\ \mathsf{r02\text{-}I}}{\langle \mathsf{x13}, \mathsf{r02}\rangle : \mathsf{Ext} \times \mathsf{Rm}}\ \times\text{-}\mathsf{I}$$

We can extract the first component of this pair with the $\times\text{-}\mathsf{E}_1$ rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x13} : \mathsf{Ext}}\ \mathsf{x13\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}\ \mathsf{Rm\text{-}F}}{\mathsf{r02} : \mathsf{Rm}}\ \mathsf{r02\text{-}I}}{\langle \mathsf{x13}, \mathsf{r02}\rangle : \mathsf{Ext} \times \mathsf{Rm}}\ \times\text{-}\mathsf{I}}{\mathsf{fst}(\langle \mathsf{x13}, \mathsf{r02}\rangle) : \mathsf{Ext}}\ \times\text{-}\mathsf{E}_1$$

If we need to compute the value, we can use the computation rule $\times\text{-}\mathsf{Eq}_1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Ext} : \mathsf{Type}}\ \mathsf{Ext\text{-}F}}{\mathsf{x13} : \mathsf{Ext}}\ \mathsf{x13\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}\ \mathsf{Rm\text{-}F}}{\mathsf{r02} : \mathsf{Rm}}\ \mathsf{r02\text{-}I}}{\langle \mathsf{x13}, \mathsf{r02}\rangle : \mathsf{Ext} \times \mathsf{Rm}}\ \times\text{-}\mathsf{I}}{\mathsf{fst}(\langle \mathsf{x13}, \mathsf{r02}\rangle) = \mathsf{x13} : \mathsf{Ext}}\ \times\text{-}\mathsf{Eq}_1$$

And we can use substitution to extract just the value:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Ext : Type}}\,\mathsf{Ext\text{-}F}}{\mathsf{x13 : Ext}}\,\mathsf{x13\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Rm : Type}}\,\mathsf{Rm\text{-}F}}{\mathsf{r02 : Rm}}\,\mathsf{r02\text{-}I}}{\langle \mathsf{x13}, \mathsf{r02}\rangle : \mathsf{Ext} \times \mathsf{Rm}}\,\times\text{-}\mathsf{I}}{\mathsf{fst}(\langle \mathsf{x13}, \mathsf{r02}\rangle) = \mathsf{x13} : \mathsf{Ext}}\,\times\text{-}\mathsf{Eq}_1 \qquad \dfrac{\dfrac{}{\mathsf{Ext : Type}}\,\mathsf{Ext\text{-}F}}{\mathsf{x13 : Ext}}\,\mathsf{x13\text{-}I}}{\mathsf{x13 : Ext}}\,\mathsf{Sub}_2$$

Alternatively, we could also use the $\times\text{-}\mathsf{Eq}_2$ rule to extract the second component of the pair:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Ext : Type}}\,\mathsf{Ext\text{-}F}}{\mathsf{x13 : Ext}}\,\mathsf{x13\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Rm : Type}}\,\mathsf{Rm\text{-}F}}{\mathsf{r02 : Rm}}\,\mathsf{r02\text{-}I}}{\langle \mathsf{x13}, \mathsf{r02}\rangle : \mathsf{Ext} \times \mathsf{Rm}}\,\times\text{-}\mathsf{I}}{\mathsf{snd}(\langle \mathsf{x13}, \mathsf{r02}\rangle) = \mathsf{r02} : \mathsf{Rm}}\,\times\text{-}\mathsf{Eq}_2 \qquad \dfrac{\dfrac{}{\mathsf{Rm : Type}}\,\mathsf{Rm\text{-}F}}{\mathsf{r02 : Rm}}\,\mathsf{r02\text{-}I}}{\mathsf{r02 : Rm}}\,\mathsf{Sub}_2$$

14 Families of types

In CTT, when we write out the name of a type (say, in the formation rule), we are allowed to insert the names of inhabitants from other types.

Suppose we want to define a type for projects that each employees are working on. For example, we could define a type for the projects that Alice is working on, which we can call $\mathsf{Prj(Alice)}$:

$$\dfrac{\mathsf{Em : Type} \qquad \mathsf{Alice : Em}}{\mathsf{Prj(Alice) : Type}}\,\mathsf{Prj(Alice)\text{-}F}$$

This says that, so long as Em is a valid type and Alice is a valid inhabitant of Em, then $\mathsf{Prj(Alice)}$ is a valid type.

We can define a similar type for Bob:

$$\dfrac{\mathsf{Em : Type} \qquad \mathsf{Bob : Em}}{\mathsf{Prj(Bob) : Type}}\,\mathsf{Prj(Bob)\text{-}F}$$

Together, these make up a family of types, since they are exactly the same, except for the name of the employee used in their name.

To define the whole family in a concise way, we can simply use a generic placeholder, like x, to stand in place of the employee name:

$$\dfrac{\mathsf{Em : Type} \qquad \mathsf{x : Em}}{\mathsf{Prj(x) : Type}}\,\mathsf{Prj(x)\text{-}F}$$

This says that, if Em is a valid type and x is a valid inhabitant of Em (whichever inhabitant x might be), then $\mathsf{Prj(x)}$ is a valid type. This rule defines the entire type family, because it defines the type Prj(x) for any x of Em, rather than some particular inhabitant of Em.

Once we have defined the family of types, we can specify the inhabitants of each type in the family, just as we would for any other finite type.

For example, let us suppose that Alice is working on two projects: accounting (which we will call acc) and logistics (which we will call log).

$$\frac{\mathsf{Prj(Alice) : Type}}{\mathsf{acc : Prj(Alice)}}\ \mathsf{acc\text{-}I} \qquad \frac{\mathsf{Prj(Alice) : Type}}{\mathsf{log : Prj(Alice)}}\ \mathsf{log\text{-}I}$$

The rule on the left says that, if Prj(Alice) is a valid type, then acc inhabits that type. The rule on the right is similar, but for log.

Likewise, we can define inhabitants for the type Prj(Bob). Let's suppose that Bob is working on one project. marking (which we will call mrk):

$$\frac{\mathsf{Prj(Bob) : Type}}{\mathsf{mrk : Prj(Bob)}}\ \mathsf{mrk\text{-}I}$$

So each type in the family is a type just like any other, which has formation rules and introduction rules. We can also define elimination rules and computation rules for these types too, just as we can for any other type.

Type families are often called dependent types, because the types in the family depend on the inhabitants of another type. They are also sometimes called indexed types, because each type in the family is indexed by an inhabitant from another type.

15 Type families and elimination rules

Elimination rules work with type families too. If we can pair up each object in a type A with an object from the corresponding type of a type family, then we can switch over the cases.

Suppose we have a type A, which has three inhabitants a_1, a_2, and a_3. Suppose that we also have a type family B(x), where x is an inhabitant of A. Thus, in the family B(x), we have three types: $B(a_1)$, $B(a_2)$, and $B(a_3)$. Suppose finally that each of these types has one inhabitant: b_1 inhabits $B(a_1)$, b_2 inhabits $B(a_2)$, and b_3 inhabits $B(a_3)$.

We can construct a switch over A, because for each object in A, we can pair it up with an object from the corresponding type in the family B(x). Here is the switch:

```
switch on (x : A) :
    case 1 (if x is $a_1$) : return $b_1 : B(a_1)$
    case 2 (if x is $a_2$) : return $a_2 : B(a_2)$
    case 3 (if x is $a_3$) : return $a_1 : B(a_3)$
```

This switch is exactly like the other switches we discussed before, but there is something new: each return type matches the value of the case.

Consider the first case. The value of $\mathsf{x} : \mathsf{A}$ in the first case is a_1. The return type of this case is the corresponding type from the family $\mathsf{B(x)}$.

What is the corresponding type of that family? It is $\mathsf{B}(\mathsf{a}_1)$, because in the first case x is a_1. The return type for case a_1 is not just any type in the $\mathsf{B(x)}$ family. It is the type we get when we substitute the case a_1 in for x, which is $\mathsf{B}(\mathsf{a}_1)$.

The same holds for the second case. The value of $\mathsf{x} : \mathsf{A}$ in the second case is a_2, and so the return type is $\mathsf{B}(\mathsf{a}_2)$, which is the type in the family $\mathsf{B(x)}$ that we get when we substitute a_2 for x. Similarly, the return type for the third case, when x is a_3, is $\mathsf{B}(\mathsf{a}_3)$.

So, in order to pair up objects from a type A with objects from a family of types $\mathsf{B(x)}$, each case of A must be paired up with an object from the type in $\mathsf{B(x)}$ that matches its case.

We can update the elimination rule template we presented earlier, to account for this new requirement. The updated rule template looks like the following, where a_1, a_2, and so on refer to subsequent inhabitants of A:

$$\frac{\mathsf{x} : \mathsf{A} \qquad \mathsf{option\ 1} : \mathsf{B}(\mathsf{a}_1) \qquad \mathsf{option\ 2} : \mathsf{B}(\mathsf{a}_2) \qquad \ldots}{\mathsf{switch}(\mathsf{x}, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) : \mathsf{B}[\mathsf{x} := \mathsf{a}_1/\mathsf{a}_2/\ldots]}\ \mathsf{A\text{-}E}$$

This rule template is just the same as it was before, except that it specifies the output types for each case.

The computation rules also need to be updated in a similar way. Here is the first rule, updated:

$$\frac{\mathsf{a}_1 : \mathsf{A} \qquad \mathsf{option\ 1} : \mathsf{B}(\mathsf{a}_1) \qquad \mathsf{option\ 2} : \mathsf{B}(\mathsf{a}_2) \qquad \ldots}{\mathsf{switch}(\mathsf{a}_1, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) = \mathsf{option\ 1} : \mathsf{B}(\mathsf{a}_1)}\ \mathsf{A\text{-}Eq}_1$$

Here is the second computation rule, updated:

$$\frac{\mathsf{a}_2 : \mathsf{A} \qquad \mathsf{option\ 1} : \mathsf{B}(\mathsf{a}_1) \qquad \mathsf{option\ 2} : \mathsf{B}(\mathsf{a}_2) \qquad \ldots}{\mathsf{switch}(\mathsf{a}_2, \mathsf{option\ 1}, \mathsf{option\ 2}, \ldots) = \mathsf{option\ 2} : \mathsf{B}(\mathsf{a}_2)}\ \mathsf{A\text{-}Eq}_2$$

15.1 Examples.

Suppose we want to pair up each employee Em with a project $\mathsf{Prj(x)}$. Here is one possible way to do that:

```
switch on (x : Emp) :
    case 1 (if x is Alice) : return acc : Prj(Alice)
    case 2 (if x is Bob) : return mrk : Prj(Bob)
```

If the employee in question is Alice, then we can return the accounting project. If it is Bob, then we return the marketing project.

Here is that switch, as an elimination rule:

$$\frac{\mathsf{x : Em} \qquad \mathsf{acc : Prj(Alice)} \qquad \mathsf{mrk : Prj(Bob)}}{\mathsf{switch(x, acc, mrk) : Prj[x := Alice/Bob]}}\ \mathsf{Em\text{-}E}$$

And here are the computation rules:

$$\frac{\mathsf{Alice : Em} \qquad \mathsf{acc : Prj(Alice)} \qquad \mathsf{mrk : Prj(Bob)}}{\mathsf{switch(Alice, acc, mrk) = acc : Prj(Alice)}}\ \mathsf{Em\text{-}Eq}_1$$

$$\frac{\mathsf{Bob : Em} \qquad \mathsf{acc : Prj(Alice)} \qquad \mathsf{mrk : Prj(Bob)}}{\mathsf{switch(Bob, acc, mrk) = mrk : Prj(Bob)}}\ \mathsf{Em\text{-}Eq}_2$$

Since Alice is assigned to two projects (i.e., since there are two inhabitants in the type Prj(Alice)), there is another possible switch. We could pair up Alice with the other project she's assigned to, namely logistics:

```
switch on (x : Emp) :
    case 1 (if x is Alice) : return log : Prj(Alice)
    case 2 (if x is Bob) : return mrk : Prj(Bob)
```

Here is that switch, as an elimination rule:

$$\frac{\mathsf{x : Em} \qquad \mathsf{log : Prj(Alice)} \qquad \mathsf{mrk : Prj(Bob)}}{\mathsf{switch(x, log, mrk) : Prj[x := Alice/Bob]}}\ \mathsf{Em\text{-}E}$$

And here are the computation rules:

$$\frac{\mathsf{Alice : Em} \qquad \mathsf{log : Prj(Alice)} \qquad \mathsf{mrk : Prj(Bob)}}{\mathsf{switch(Alice, log, mrk) = log : Prj(Alice)}}\ \mathsf{Em\text{-}Eq}_1$$

$$\frac{\mathsf{Bob : Em} \qquad \mathsf{log : Prj(Alice)} \qquad \mathsf{mrk : Prj(Bob)}}{\mathsf{switch(Bob, log, mrk) = mrk : Prj(Bob)}}\ \mathsf{Em\text{-}Eq}_2$$

16 Families with empty members

Sometimes, we cannot pair up every member of one type A with an object from a corresponding type in a family of types B(x). This happens when at least one of the types in the family is empty (i.e., when it has no inhabitants).

Suppose we make copies of the Em and Prj(x) types, called Em^* and $\mathsf{Prj(x)}^*$. Suppose we then add another employee to Em^*, named Cary:

$$\frac{\mathsf{Em}^* : \mathsf{Type}}{\mathsf{Cary} : \mathsf{Em}^*}\ \mathsf{Cary\text{-}I}$$

Since the type $\mathsf{Prj(x)}^*$ is a family of types defined over Em^*, there is now a new type in this family, namely Cary's projects $\mathsf{Prj(Cary)}^*$:

$$\frac{\mathsf{Em}^* : \mathsf{Type} \qquad \mathsf{Cary} : \mathsf{Em}^*}{\mathsf{Prj(Cary)}^*}\ \mathsf{Prj(Cary)}^*\text{-}\mathsf{F}$$

Suppose now that Cary has not been assigned any projects yet, and so the type $\mathsf{Prj(Cary)}^*$ has no inhabitants.

Now we have a family of types $\mathsf{Prj(x)}^*$ which has three members: $\mathsf{Prj(Alice)}^*$, $\mathsf{Prj(Bob)}^*$, and $\mathsf{Prj(Cary)}^*$. The first two of these are inhabited, but the third is empty.

Can we pair up each employee with a project? That is, can we pair up each item from Em^* with an item from the corresponding type in the family $\mathsf{Prj(x)}^*$? The answer is no, because there are no projects for the third employee.

This means that we cannot switch over Em^* with respect to $\mathsf{Prj(x)}^*$. Hence, we cannot form an elimination rule (or its associated computation rules) for Em^* via $\mathsf{Prj(x)}^*$.

17 Dependent functions

Abstraction works with families of types too. For example, suppose we have this switch from earlier:

$$\frac{\mathsf{x} : \mathsf{Em} \qquad \mathsf{log} : \mathsf{Prj(Alice)} \qquad \mathsf{mrk} : \mathsf{Prj(Bob)}}{\mathsf{switch(x, log, mrk)} : \mathsf{Prj[x := Alice/Bob]}}\ \mathsf{Em\text{-}E}$$

We can use the λ-I rule and mark $\mathsf{x} : \mathsf{Em}$ as replaceable:

$$\cfrac{\cfrac{\mathsf{x} : \mathsf{Em}^1 \qquad \mathsf{log} : \mathsf{Prj(Alice)} \qquad \mathsf{mrk} : \mathsf{Prj(Bob)}}{\mathsf{switch(x, log, mrk)} : \mathsf{Prj[x := Alice/Bob]}}\ \mathsf{Em\text{-}E}}{\lambda\mathsf{x} : \mathsf{Em.switch(x, log, mrk)} : \mathsf{Em} \to \mathsf{Prj(x)}}\ \lambda\text{-}\mathsf{I}^1$$

Now we have a function from employees to their projects. We can apply it to Alice, for example:

$$\cfrac{\cfrac{\vdots}{\lambda\mathsf{x} : \mathsf{Em.switch(x, log, mrk)} : \mathsf{Em} \to \mathsf{Prj(x)}}\ \lambda\text{-}\mathsf{I}^1 \qquad \begin{matrix}\vdots \\ \mathsf{Alice} : \mathsf{Em}\end{matrix}}{\mathsf{app}(\lambda\mathsf{x} : \mathsf{Em.switch(x, log, mrk), Alice}) : \mathsf{Prj(Alice)}}\ \lambda\text{-}\mathsf{E}$$

Notice that when we apply the function to Alice, we also apply Alice to the type family $\mathsf{Prj(x)}$, to get $\mathsf{Prj(Alice)}$.

We are using the same mechanism for abstraction and application as before, except this time around, we are extending the substitution to the type families too.

In CTT, when abstraction involves type families, we use a different notation for the types. In a simple abstraction, the type is a simple function from

$\mathsf{A} \to \mathsf{B}$. But this does not give us enough information. When dependent types are involved, the abstraction is a function from each *object* xs of type A to the *family* of types $\mathsf{B(x)}$ that depend on the value of x.

To make that clear, I suppose we could encode the type like this:

$$\mathsf{x : A \to B(x)}$$

But instead, we use a different notation. We write it like this:

$$\mathsf{\Pi x : A, B(x)}$$

We call this the type of dependent functions. Its rules are the same as they are for λ, but with type families taken into account. Here is the Π formation rule:

$$\frac{\mathsf{A : Type} \qquad \mathsf{x : A} \qquad \mathsf{B(x) : Type}}{\mathsf{\Pi x : A, B(x) : Type}}\ \Pi\text{-}\mathsf{F}$$

Here is the Π introduction rule:

$$\frac{\begin{array}{c}\mathsf{x : A}^1 \\ \vdots \\ \mathsf{b : B(x)}\end{array}}{\mathsf{\lambda x : A.b : \Pi x : A, B(x)}}\ \Pi\text{-}\mathsf{I}^1$$

Here is the Π elimination rule:

$$\frac{\mathsf{\lambda x : A.b(x) : \Pi x : A, B(x)} \qquad \mathsf{a : A}}{\mathsf{app(\lambda x : A.b, a) : B(a)}}\ \Pi\text{-}\mathsf{E}$$

And here is the Π computation rule:

$$\frac{\mathsf{\lambda x : A.b(x) : \Pi x : A, B(x)} \qquad \mathsf{a : A}}{\mathsf{app(\lambda x : A.b, a) = b[x := a] : B(a)}}\ \Pi\text{-}\mathsf{Eq}$$

Now we can rewrite the above example, using the Π rules, instead of the λ rules. So, for example, take the switch we started this section with:

$$\frac{\mathsf{x : Em} \qquad \mathsf{log : Prj(Alice)} \qquad \mathsf{mrk : Prj(Bob)}}{\mathsf{switch(x, log, mrk) : Prj[x := Alice/Bob]}}\ \mathsf{Em\text{-}E}$$

Instead of using the λ-I rule, we can use the Π-I rule to mark $\mathsf{x : Em}$ as replaceable:

$$\frac{\dfrac{\mathsf{x : Em}^1 \qquad \mathsf{log : Prj(Alice)} \qquad \mathsf{mrk : Prj(Bob)}}{\mathsf{switch(x, log, mrk) : Prj[x := Alice/Bob]}}\ \mathsf{Em\text{-}E}}{\mathsf{\lambda x : Em.switch(x, log, mrk) : \Pi x : Em, Prj(x)}}\ \Pi\text{-}\mathsf{I}^1$$

Now we have a dependent function from employees to their projects. And we can apply it to Alice, just as before, but using the Π rules:

$$\dfrac{\dfrac{\vdots}{\mathsf{\lambda x : Em.switch(x, log, mrk) : \Pi x : Em, Prj(x)}}\,\Pi\text{-}\mathsf{I}^1 \qquad \begin{array}{c}\vdots\\ \mathsf{Alice : Em}\end{array}}{\mathsf{app(\lambda x : Em.switch(x, log, mrk), Alice) : Prj(Alice)}}\,\Pi\text{-}\mathsf{E}$$

17.1 Universal quantification.

If we like, we can replace the Π symbol with the universal quantifier symbol, ∀/. From the perspective of CTT, these are the same thing. If we do that, we have these extra rules (which are really just synonyms for the Π variants):

$$\dfrac{\mathsf{A : Type} \qquad \mathsf{x : A} \qquad \mathsf{B(x) : Type}}{\mathsf{\forall x : A, B(x) : Type}}\,\forall\text{-}\mathsf{F}$$

Here is the ∀ introduction rule:

$$\dfrac{\begin{array}{c}\mathsf{x : A}^1\\ \vdots\\ \mathsf{b : B(x)}\end{array}}{\mathsf{\lambda x : A.b : \forall x : A, B(x)}}\,\forall\text{-}\mathsf{I}^1$$

Here is the ∀ elimination rule:

$$\dfrac{\mathsf{\lambda x : A.b(x) : \Pi x : A, B(x)} \qquad \mathsf{a : A}}{\mathsf{app(\lambda x : A.b, a) : B(a)}}\,\forall\text{-}\mathsf{E}$$

And here is the ∀ computation rule:

$$\dfrac{\mathsf{\lambda x : A.b(x) : \forall x : A, B(x)} \qquad \mathsf{a : A}}{\mathsf{app(\lambda x : A.b, a) = b[x := a] : B(a)}}\,\forall\text{-}\mathsf{Eq}$$

17.2 Examples.

Recall from earlier that we assigned Alice to two projects (accounting and logistics), and we assigned Bob to one project (marketing).

Let us prove that for any employee, there is a project. First, let us construct a switch that pairs up Alice with one of her projects (let us pick acc) and Bob with his project (mrk):

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{x} : \mathsf{Em}} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}} \quad \dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}}}{\mathsf{Prj}(\mathsf{Alice}) : \mathsf{Type}}}{\mathsf{acc} : \mathsf{Prj}(\mathsf{Alice})} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}} \quad \dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}}}{\mathsf{Prj}(\mathsf{Bob}) : \mathsf{Type}}}{\mathsf{mrk} : \mathsf{Prj}(\mathsf{Bob})}}{\mathsf{switch}(\mathsf{x}, \mathsf{acc}, \mathsf{mrk}) : \mathsf{Prj}(\mathsf{x})[\mathsf{x} := \mathsf{Alice}/\mathsf{Bob}]}$$

Now let us mark x as replaceable:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{x} : \mathsf{Em}^1} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}} \quad \dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}}}{\mathsf{Prj}(\mathsf{Alice}) : \mathsf{Type}}}{\mathsf{acc} : \mathsf{Prj}(\mathsf{Alice})} \quad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}} \quad \dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}}}{\mathsf{Prj}(\mathsf{Bob}) : \mathsf{Type}}}{\mathsf{mrk} : \mathsf{Prj}(\mathsf{Bob})}}{\mathsf{switch}(\mathsf{x}, \mathsf{acc}, \mathsf{mrk}) : \mathsf{Prj}(\mathsf{x})[\mathsf{x} := \mathsf{Alice}/\mathsf{Bob}]}}{\lambda \mathsf{x} : \mathsf{Em}.\mathsf{switch}(\mathsf{x}, \mathsf{acc}, \mathsf{mrk}) : \Pi \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})}$$

With this last step, we constructed a function that maps each employee x to a project in Prj(x), i.e., to the projects specifically for them. To make the mapping more explicit:

$$\begin{aligned} \mathsf{Alice} &\longrightarrow \mathsf{acc} \\ \mathsf{Bob} &\longrightarrow \mathsf{mrk} \end{aligned}$$

If we add the types:

$$\begin{aligned} \mathsf{Alice} : \mathsf{Em} &\longrightarrow \mathsf{acc} : \mathsf{Prj}(\mathsf{Alice}) \\ \mathsf{Bob} : \mathsf{Em} &\longrightarrow \mathsf{mrk} : \mathsf{Prj}(\mathsf{Bob}) \end{aligned}$$

So the function $\lambda \mathsf{x} : \mathsf{Em}.\mathsf{switch}(\mathsf{x}, \mathsf{acc}, \mathsf{mrk})$ is an inhabitant of the type $\Pi \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})$. The above derivation is a proof that $\Pi \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})$ is "true." I.e., the above derivation derivation constructs an inhabitant of the type, and so it shows that for every employee, there is a project.

Notice that this function does not map each employee x to just any inhabitant of the family of types Prj(x). On the contrary, it only maps each employee x to that employee's corresponding type in the family. So, Alice is mapped to a project in Prj(Alice), not a project in Prj(Bob), while Prj(Bob) is mapped to a project in Prj(Bob), not a project in Prj(Alice).

Since Alice is assigned to two projects (i.e., since there are two inhabitants in the type Prj(Alice)), there is another function we could build. This time around, we will pair up Alice with her other project log. Let us construct that switch:

$$\dfrac{\dfrac{\dfrac{}{\textsf{Em : Type}}}{\textsf{x : Em}} \quad \dfrac{\dfrac{\dfrac{}{\textsf{Em : Type}} \quad \dfrac{\dfrac{}{\textsf{Em : Type}}}{\textsf{Alice : Em}}}{\textsf{Prj(Alice) : Type}}}{\textsf{log : Prj(Alice)}} \quad \dfrac{\dfrac{\dfrac{}{\textsf{Em : Type}} \quad \dfrac{\dfrac{}{\textsf{Em : Type}}}{\textsf{Bob : Em}}}{\textsf{Prj(Bob) : Type}}}{\textsf{mrk : Prj(Bob)}}}{\textsf{switch(x, log, mrk) : Prj(x)[x := Alice/Bob]}}$$

Now let us mark x as replaceable:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\textsf{Em : Type}}}{\textsf{x : Em}^1} \quad \dfrac{\dfrac{\dfrac{}{\textsf{Em : Type}} \quad \dfrac{\dfrac{}{\textsf{Em : Type}}}{\textsf{Alice : Em}}}{\textsf{Prj(Alice) : Type}}}{\textsf{log : Prj(Alice)}} \quad \dfrac{\dfrac{\dfrac{}{\textsf{Em : Type}} \quad \dfrac{\dfrac{}{\textsf{Em : Type}}}{\textsf{Bob : Em}}}{\textsf{Prj(Bob) : Type}}}{\textsf{mrk : Prj(Bob)}}}{\textsf{switch(x, log, mrk) : Prj(x)[x := Alice/Bob]}}}{\lambda\textsf{x : Em.switch(x, log, mrk) : }\Pi\textsf{x : Em, Prj(x)}}$$

This function looks very similar to the last function. The difference is that the switch will return log for Alice rather than acc. To make the mapping explicit:

$$\begin{aligned} \mathsf{Alice} &\longrightarrow \mathsf{log} \\ \mathsf{Bob} &\longrightarrow \mathsf{mrk} \end{aligned}$$

If we add the types:

$$\begin{aligned} \textsf{Alice : Em} &\longrightarrow \textsf{log : Prj(Alice)} \\ \textsf{Bob : Em} &\longrightarrow \textsf{mrk : Prj(Bob)} \end{aligned}$$

So we have here constructed another function, that maps an employee x to the type that corresponds to it in the family $\mathsf{Prj(x)}$. This is a second proof that each employee has a project.

If you think about it, this makes good sense. Since there are two projects assigned to Alice and one assigned to Bob, there should be two functions that pair up each employee with a project (let us call these functions f and g):

$$\begin{aligned} \mathsf{f(Alice)} &= \mathsf{acc} \\ \mathsf{f(Bob)} &= \mathsf{mrk} \end{aligned}$$

$$\begin{aligned} \mathsf{g(Alice)} &= \mathsf{log} \\ \mathsf{g(Bob)} &= \mathsf{mrk} \end{aligned}$$

And indeed, the two functions we constructed are precisely f and g.

17.3 Missing members.

If we cannot assign a project to each employee, then we cannot switch over them. For example, recall the types Em^* and $\mathsf{Prj}(\mathsf{x})^*$ above. In Em^*, there is an employee Cary who has no project assigned to her in $\mathsf{Prj}(\mathsf{Cary})^*$. So, we cannot construct a switch that pairs up each employee from Em^* with a project from $\mathsf{Prj}(\mathsf{x})^*$.

But if we cannot construct such a switch, then we also cannot us the Π-I rule to form a function that maps employees from Em^* to projects in $\mathsf{Prj}(\mathsf{x})^*$. So we cannot construct an inhabitant of the type $\mathsf{\Pi x} : \mathsf{Em}^*, \mathsf{Pr}(\mathsf{x})^*$.

In other words, it is not "true" that every employee from Em^* has a project in $\mathsf{Prj}(\mathsf{x})^*$. And that is exactly what we expect, because we know by looking at the introduction rules for Em^* and $\mathsf{Prj}(\mathsf{x})^*$ that Cary has no project. So we should not be able to construct a function/proof that every employee from Em^* has a project in $\mathsf{Prj}(\mathsf{x})^*$.

17.4 Dependent function space.

Notice that the functions we defined above are dependent functions, because their output depends on their input. Take the first of the above two functions, which maps Alice to acc and Bob to mrk. The domain of that function is the set of employees, but there is not one codomain or range for the function. Rather, there is a family of codomains or ranges, one for each employee. Which codomain or range the function will output depends on which employee we give it for input. If we give it Alice for input, then its output will range over the inhabitants of $\mathsf{Prj}(\mathsf{Alice})$. If we give it Bob for input, then its output will range over the inhabitants of $\mathsf{Prj}(\mathsf{Bob})$.

The type $\mathsf{\Pi x} : \mathsf{A}, \mathsf{B}(\mathsf{x})$ thus represents the type of all dependent functions from $\mathsf{x} : \mathsf{A}$ to $\mathsf{B}(\mathsf{x})$. Its inhabitants will be functions that map each element x in A to an element in the type that corresponds to that element in the family $\mathsf{B}(\mathsf{x})$.

So, for example, suppose that A has the elements a, b, and c:

$$\mathsf{a}, \mathsf{b}, \mathsf{c} \in \mathsf{A}$$

Then suppose that $\mathsf{B_a}$ has the elements 1 and 2, $\mathsf{B_b}$ has the element 34, and suppose that $\mathsf{C_c}$ has the elements 20, 21, and 23:

$$\begin{aligned} 1, 2 &\in \mathsf{B_a} \\ 34 &\in \mathsf{B_b} \\ 20, 21, 22 &\in \mathsf{B_c} \end{aligned}$$

Now, we can define a number functions that map each element from A to one element in the the corresponding $\mathsf{B_a}$, $\mathsf{B_b}$, or $\mathsf{B_c}$. Here are the options:

$$\begin{array}{lll}
\mathsf{f(a) = 1} & \mathsf{g(a) = 2} & \mathsf{h(a) = 1} \\
\mathsf{f(b) = 34} & \mathsf{g(b) = 34} & \mathsf{h(b) = 34} \\
\mathsf{f(c) = 20} & \mathsf{g(c) = 20} & \mathsf{h(c) = 21} \\
 & & \\
\mathsf{i(a) = 2} & \mathsf{j(a) = 1} & \mathsf{k(a) = 2} \\
\mathsf{i(b) = 34} & \mathsf{j(b) = 34} & \mathsf{k(b) = 34} \\
\mathsf{i(c) = 21} & \mathsf{j(c) = 22} & \mathsf{k(c) = 22}
\end{array}$$

This is the dependent function space from A to the family $\mathsf{B_A}$. The Π type is precisely a dependent function space, and its possible inhabitants are precisely all dependent functions like these.

17.5 Deriving λ rules

The dependent function type is a very general type. In fact, the regular λ type discussed earlier is just a special case of it. It is the case where B is not a type family. We can derive the λ rules from the Π rules, simply by setting $\mathsf{B(x)}$ in the rules to be B (without any x in it).

Consider the Π formation rule:

$$\frac{\mathsf{A : Type} \qquad \mathsf{x : A} \qquad \mathsf{B(x) : Type}}{\mathsf{\Pi x : A, B(x) : Type}}\ \Pi\text{-F}$$

Change $\mathsf{B(x)}$ to just B:

$$\frac{\mathsf{A : Type} \qquad \mathsf{x : A} \qquad \mathsf{B : Type}}{\mathsf{\Pi x : A, B : Type}}\ \Pi\text{-F}$$

The type $\mathsf{\Pi x : A, B}$ represents all mappings from each x in A to an element in the corresponding type of the family B. But B has only one member in the family, namely B. So the type $\mathsf{\Pi x : A, B}$ represents all mappings from each element in A to an element in B.

Hence, instead of writing the type as $\mathsf{\Pi x : A, B}$, we are justified in writing it with a notation that is commonly used to represent mere function types, namely $\mathsf{A \to B}$:

$$\frac{\mathsf{A : Type} \qquad \mathsf{x : A} \qquad \mathsf{B : Type}}{\mathsf{A \to B : Type}}\ \Pi\text{-F}$$

Also, since $\mathsf{x : Type}$ is not used in B, we don't need it as a hypothesis in the rule. So we can remove it, which leaves us with the same formation rule that we saw for λ:

$$\frac{\mathsf{A : Type} \qquad \mathsf{B : Type}}{\mathsf{A \to B : Type}}\ \Pi\text{-F} \equiv \lambda\text{-F}$$

Now consider the is the Π introduction rule:

$$\frac{\begin{array}{c}\mathsf{x : A}^1 \\ \vdots \\ \mathsf{b : B(x)}\end{array}}{\lambda\mathsf{x : A.b : \Pi x : A, B(x)}}\ \Pi\text{-}\mathsf{I}^1$$

Replace $\mathsf{B(x)}$ with B:

$$\frac{\begin{array}{c}\mathsf{x : A}^1 \\ \vdots \\ \mathsf{b : B}\end{array}}{\lambda\mathsf{x : A.b : \Pi x : A, B}}\ \Pi\text{-}\mathsf{I}^1$$

And replace the type $\mathsf{\Pi x : A, B}$ with $\mathsf{A} \rightarrow \mathsf{B}$, which leaves us with the same rule we saw for λ-I:

$$\frac{\begin{array}{c}\mathsf{x : A}^1 \\ \vdots \\ \mathsf{b : B}\end{array}}{\lambda\mathsf{x : A.b : A \rightarrow B}}\ \Pi\text{-}\mathsf{I}^1 \equiv \lambda\text{-}\mathsf{I}^1$$

The elimination and computation rules follow suit. We can replace $\mathsf{B(x)}$ with B and $\mathsf{\Pi x : A, B}$ with $\mathsf{A} \rightarrow \mathsf{B}$:

$$\frac{\lambda\mathsf{x : A.b : A \rightarrow B} \qquad \mathsf{a : A}}{\mathsf{app}(\lambda\mathsf{x : A.b, a) : B}}\ \Pi\text{-}\mathsf{E}$$

$$\frac{\lambda\mathsf{x : A.b(x) : A \rightarrow B} \qquad \mathsf{a : A}}{\mathsf{app}(\lambda\mathsf{x : A.b, a) = b[x := a] : B}}\ \Pi\text{-}\mathsf{Eq}$$

However, since there are no type families involved, in this case we are applying a mere function (i.e., $\mathsf{A} \rightarrow \mathsf{B}$), rather than a dependent function (i.e., $\mathsf{\Pi x : A, B(x)}$). Hence, we are justified in calling these rules λ-E and λ-Eq. And indeed, now that we have removed the type family from the rules, these rules turn out to be identical to the λ-E and λ-Eq rules we saw above:

$$\frac{\lambda\mathsf{x : A.b : A \rightarrow B} \qquad \mathsf{a : A}}{\mathsf{app}(\lambda\mathsf{x : A.b, a) : B}}\ \Pi\text{-}\mathsf{E} \equiv \lambda\text{-}\mathsf{E}$$

$$\frac{\lambda\mathsf{x : A.b(x) : A \rightarrow B} \qquad \mathsf{a : A}}{\mathsf{app}(\lambda\mathsf{x : A.b, a) = b[x := a] : B}}\ \Pi\text{-}\mathsf{Eq} \equiv \lambda\text{-}\mathsf{Eq}$$

We can show all of this with an example. Suppose we want to form a type which maps every employee to a phone extension. We can use the Π-F rule:

$$\frac{\mathsf{Em : Type} \qquad \mathsf{x : Em} \qquad \mathsf{Ext : Type}}{\mathsf{\Pi x : Em, Ext : Type}}\ \Pi\text{-}\mathsf{F}$$

To construct an inhabitant of this type, we can construct a switch:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em}}\,\mathsf{Em\text{-}F}}{\mathsf{x : Em}}\,\mathsf{x\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext}}\,\mathsf{Ext\text{-}F}}{\mathsf{Bob : Em}}\,\mathsf{x11\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext}}\,\mathsf{Ext\text{-}F}}{\mathsf{Bob : Em}}\,\mathsf{x12\text{-}I}}{\mathsf{switch(x, x11, x12) : Ext}}\,\mathsf{Em\text{-}E}$$

Then we can use the Π-I rule and mark x as replaceable:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em}}\,\mathsf{Em\text{-}F}}{\mathsf{x : Em}^1}\,\mathsf{x\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext}}\,\mathsf{Ext\text{-}F}}{\mathsf{Bob : Em}}\,\mathsf{x11\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext}}\,\mathsf{Ext\text{-}F}}{\mathsf{Bob : Em}}\,\mathsf{x12\text{-}I}}{\mathsf{switch(x, x11, x12) : Ext}}\,\mathsf{Em\text{-}E}}{\lambda\mathsf{x : Em.switch(x, x11, x12) : \Pi x : Em, Ext}}\,\Pi\text{-}\mathsf{I}^1$$

Here we constructed a function that takes an employee, and returns a phone extension. Here is the mapping:

$$\mathsf{Alice : Em} \to \mathsf{x11 : Ext}$$
$$\mathsf{Bob : Em} \to \mathsf{x12 : Ext}$$

There are no families involved here. This function simply pairs up each employee with an extension. So this is a function from employees to extensions. Hence, we are justified in changing the type from $\mathsf{\Pi x : Em, Ext}$ to $\mathsf{Em} \to \mathsf{Ext}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em}}\,\mathsf{Em\text{-}F}}{\mathsf{x : Em}^1}\,\mathsf{x\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext}}\,\mathsf{Ext\text{-}F}}{\mathsf{Bob : Em}}\,\mathsf{x11\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext}}\,\mathsf{Ext\text{-}F}}{\mathsf{Bob : Em}}\,\mathsf{x12\text{-}I}}{\mathsf{switch(x, x11, x12) : Ext}}\,\mathsf{Em\text{-}E}}{\lambda\mathsf{x : Em.switch(x, x11, x12) : Em} \to \mathsf{Ext}}\,\lambda\text{-}\mathsf{I}^1$$

Suppose now that we want to use this function to "look up" Bob's extension. We could use the Π-E or Π-Eq rule to apply this function to Bob:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em}}\,\mathsf{Em\text{-}F}}{\mathsf{x : Em}^1}\,\mathsf{x\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext}}\,\mathsf{Ext\text{-}F}}{\mathsf{Bob : Em}}\,\mathsf{x11\text{-}I} \qquad \dfrac{\dfrac{}{\mathsf{Ext}}\,\mathsf{Ext\text{-}F}}{\mathsf{Bob : Em}}\,\mathsf{x12\text{-}I}}{\mathsf{switch(x, x11, x12) : Ext}}\,\mathsf{Em\text{-}E}}{\lambda\mathsf{x : Em.switch(x, x11, x12) : Em} \to \mathsf{Ext}}\,\lambda\text{-}\mathsf{I}^1 \qquad \dfrac{\dfrac{}{\mathsf{Em}}\,\mathsf{Em\text{-}F}}{\mathsf{Bob : Em}}\,\mathsf{Bob\text{-}I}}{\mathsf{app(}\lambda\mathsf{x : Em.switch(x, x11, x12), Bob) = x12 : Ext}}\,\Pi\text{-}\mathsf{Eq}$$

But of course, here we are applying a mere function rather than a dependent function, so we are justified in using the λ-E or λ-Eq rule instead:

$$
\dfrac{
 \dfrac{
  \dfrac{
   \dfrac{\dfrac{}{\mathsf{Em}}\,\text{Em-F}}{\mathsf{x} : \mathsf{Em}^1}\,\text{x-I}
   \quad
   \dfrac{\dfrac{}{\mathsf{Ext}}\,\text{Ext-F}}{\mathsf{Bob} : \mathsf{Em}}\,\text{x11-I}
   \quad
   \dfrac{\dfrac{}{\mathsf{Ext}}\,\text{Ext-F}}{\mathsf{Bob} : \mathsf{Em}}\,\text{x12-I}
  }{\mathsf{switch}(\mathsf{x}, \mathsf{x11}, \mathsf{x12}) : \mathsf{Ext}}\,\text{Em-E}
 }{\lambda \mathsf{x} : \mathsf{Em}.\mathsf{switch}(\mathsf{x}, \mathsf{x11}, \mathsf{x12}) : \mathsf{Em} \to \mathsf{Ext}}\,\lambda\text{-I}^1
 \quad
 \dfrac{\dfrac{}{\mathsf{Em}}\,\text{Em-F}}{\mathsf{Bob} : \mathsf{Em}}\,\text{Bob-I}
}{\mathsf{app}(\lambda \mathsf{x} : \mathsf{Em}.\mathsf{switch}(\mathsf{x}, \mathsf{x11}, \mathsf{x12}), \mathsf{Bob}) = \mathsf{x12} : \mathsf{Ext}}\,\lambda\text{-Eq}
$$

18 Dependent products

As we saw earlier, products are pairs constructed from two types. We can allow the second type of a product to be a type family. When we do that, we call the type not a mere product, but rather a dependent product.

Suppose we want to form a dependent product from the type A, and the type B(x) (where x is an element of A). We could write such a dependent product type like this:

$$\mathsf{x} : \mathsf{A} \times \mathsf{B}(\mathsf{x})$$

However, in CTT, we use a different notation. We write it like this:

$$\Sigma \mathsf{x} : \mathsf{A}, \mathsf{B}(\mathsf{x})$$

This type represents the pairs we can form by taking some x from A, and some element from B(x) — and not just any member of B, but a member of the type in the family B(x) that corresponds to x).

For example, we can pair up employees and their projects. For example, Alice is an employee (i.e., Alice inhabits the type Em), and acc is one of her projects (i.e., acc inhabits the project type that depends on Prj(Alice)). So, we can put the two together to make a pair:

$$\langle \mathsf{Alice}, \mathsf{acc} \rangle$$

The type of this is the product of types x : Em and Prj(x), which we write not as x : Em × Prj(x), but rather as Σx : Em, Prj(x):

$$\langle \mathsf{Alice}, \mathsf{acc} \rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})$$

Dependent products are similar to dependent functions in that the second component depends on the first. If you like, the type Σx : Em, Prj(x) is really a family of types itself, one for each x in Em. So, there is one type of pairs for Alice × Prj(Alice), and there is another type that pairs for Bob × Prj(Bob):

$$\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x}) \equiv \mathsf{Alice} \times \mathsf{Prj}(\mathsf{Alice}), \mathsf{Bob} \times \mathsf{Prj}(\mathsf{Bob})$$

The second component of each of these pairs matches the first, because the type of the second component (namely, $\mathsf{Prj}(\mathsf{x})$) depends on the value of the first (i.e., which x that we are dealing with from Em).

Hence, we cannot pair up Alice with, say, Bob's projects. This would not be legal:

$$\langle \mathsf{Alice}, \mathsf{mrk} \rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})$$

The reason is that mrk does not inhabit the type $\mathsf{Prj}(\mathsf{Alice})$. Rather, it inhabits $\mathsf{Prj}(\mathsf{Bob})$, and the type $\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})$ requires that if the first component is Alice, the second component must be an inhabitant of $\mathsf{Prj}(\mathsf{Alice})$.

The rules for dependent products are much like the regular × product rules, but we convert B into a type family $\mathsf{B}(\mathsf{x})$. Here is the Σ formation rule:

$$\frac{\mathsf{A} : \mathsf{Type} \qquad \mathsf{x} : \mathsf{A} \qquad \mathsf{B}(\mathsf{x}) : \mathsf{Type}}{\Sigma \mathsf{x} : \mathsf{A}, \mathsf{B}(\mathsf{x}) : \mathsf{Type}}\ \Sigma\text{-}\mathsf{F}$$

This says that, so long as A is a legal type in the system, and so long as $\mathsf{B}(\mathsf{x})$ is a legal type in the system (for any x in A), then the dependent product $\Sigma \mathsf{x} : \mathsf{A}, \mathsf{B}(\mathsf{x})$ is also a legal type in the system.

To introduce an element of this type, we need an element x of A, and an element of $\mathsf{B}(\mathsf{x})$. If we have one of each, then we can put them together into a pair. Here is the Σ introduction rule:

$$\frac{\mathsf{A} : \mathsf{Type} \qquad \mathsf{x} : \mathsf{A} \qquad \mathsf{B}(\mathsf{x}) : \mathsf{Type} \qquad \mathsf{b} : \mathsf{B}(\mathsf{x})}{\langle \mathsf{x}, \mathsf{b} \rangle : \Sigma \mathsf{x} : \mathsf{A}, \mathsf{B}(\mathsf{x})}\ \Sigma\text{-}\mathsf{I}$$

19 Dependent product elimination rules

The Σ elimination rule is much like the ×-E rule, but we need to account for dependent types. Recall the pair elimination rule:

$$\frac{\langle \mathsf{a}, \mathsf{b} \rangle : \mathsf{A} \times \mathsf{B} \qquad \begin{array}{c} \mathsf{x} : \mathsf{A}, \mathsf{y} : \mathsf{B} \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{array}}{\mathsf{elim}(\langle \mathsf{a}, \mathsf{b} \rangle, \mathsf{c}) : \mathsf{C}}\ \times\text{-}\mathsf{E}$$

This says that if we can construct an object c of type C (possibly using objects from A and B), then if we have a pair $\langle \mathsf{a}, \mathsf{b} \rangle$ of type $\mathsf{A} \times \mathsf{B}$, we can rebuild $\mathsf{c} : \mathsf{C}$ using a and b instead.

Let us reformulate this rule, and add in dependent types. First, suppose we can construct an object c of type C, using x of type A and y of type $\mathsf{B}(\mathsf{x})$:

$$\begin{array}{c} \mathsf{x} : \mathsf{A}, \mathsf{y} : \mathsf{B}(\mathsf{x}) \\ \vdots \\ \mathsf{c} : \mathsf{C} \end{array}$$

But of course, the type of c might depend on the value of x now. So we need to indicate that we can substitute x into C:

$$\begin{array}{c} \mathsf{x : A, y : B(x)} \\ \vdots \\ \mathsf{c : C(x)} \end{array}$$

Now suppose that we have a pair from the type $\mathsf{\Sigma x : A, B(x)}$:

$$\begin{array}{cc} & \mathsf{x : A, y : B(x)} \\ & \vdots \\ \mathsf{\langle a, b\rangle : \Sigma x : A, B(x)} & \mathsf{c : C(x)} \end{array}$$

At this point, we can rebuild c of type $\mathsf{C(x)}$, using a and b instead of x and y. For that, we can use the $\mathsf{elim}()$ selector we used in the $\times$-E rule:

$$\dfrac{\begin{array}{cc} & \mathsf{x : A, y : B(x)} \\ & \vdots \\ \mathsf{\langle a, b\rangle : \Sigma x : A, B(x)} & \mathsf{c : C(x)} \end{array}}{\mathsf{elim(\langle a, b\rangle, c)}}$$

The type of the result will be $\mathsf{C(x)}$:

$$\dfrac{\begin{array}{cc} & \mathsf{x : A, y : B(x)} \\ & \vdots \\ \mathsf{\langle a, b\rangle : \Sigma x : A, B(x)} & \mathsf{c : C(x)} \end{array}}{\mathsf{elim(\langle a, b\rangle, c) : C(x)}}$$

This is the elimination rule for dependent products. We will call it Σ-E:

$$\dfrac{\begin{array}{cc} & \mathsf{x : A, y : B(x)} \\ & \vdots \\ \mathsf{\langle a, b\rangle : \Sigma x : A, B(x)} & \mathsf{c : C(x)} \end{array}}{\mathsf{elim(\langle a, b\rangle, c) : C(x)}}\ \Sigma\text{-}\mathsf{E}$$

The computation rule is much the same, except a and b should be substituted in for x and y:

$$\dfrac{\begin{array}{cc} & \mathsf{x : A, y : B(x)} \\ & \vdots \\ \mathsf{\langle a, b\rangle : \Sigma x : A, B(x)} & \mathsf{c : C(x)} \end{array}}{\mathsf{elim(\langle a, b\rangle, c) : C(x)[x := a, y := b]}}\ \Sigma\text{-}\mathsf{Eq}$$

19.1 Examples.

Let us construct a type that uses an employee and one of their projects to build it. To do this, we can pair up an employee and one of their projects, and then we can pair that up with a room. So first, let us pair up an employee with one of their projects, e.g., Alice and acc:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}} \quad \dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}}}{\mathsf{Prj}(\mathsf{Alice}) : \mathsf{Type}}}{\mathsf{acc} : \mathsf{Prj}(\mathsf{Alice})}}{\langle \mathsf{Alice}, \mathsf{acc} \rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})}$$

Now let us pair up that pair with, say, a room:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}} \quad \dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}}}{\mathsf{Prj}(\mathsf{Alice}) : \mathsf{Type}}}{\mathsf{acc} : \mathsf{Prj}(\mathsf{Alice})}}{\langle \mathsf{Alice}, \mathsf{acc} \rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \langle \mathsf{Alice}, \mathsf{acc} \rangle, \mathsf{r01} \rangle : (\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})) \times \mathsf{Rm}}$$

Here we have constructed a type (namely $(\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})) \times \mathsf{Rm}$), and we used an employee (namely Alice) and one of her projects (namely, acc) in the construction process.

Now suppose we have a pair of a different employee and project, for instance:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}} \quad \dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Bob} : \mathsf{Em}}}{\mathsf{Prj}(\mathsf{Bob}) : \mathsf{Type}}}{\mathsf{mrk} : \mathsf{Prj}(\mathsf{Bob})}}{\langle \mathsf{Bob}, \mathsf{mrk} \rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})} \qquad \dfrac{\dfrac{\vdots}{\langle \mathsf{Alice}, \mathsf{acc} \rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})} \qquad \dfrac{\vdots}{\mathsf{r01} : \mathsf{Rm}}}{\langle \langle \mathsf{Alice}, \mathsf{acc} \rangle, \mathsf{r01} \rangle : (\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})) \times \mathsf{Rm}}$$

The dependent product elimination rule says that we can rebuild the type on the right, using the components of the pair on the left:

$$\dfrac{\dfrac{\vdots}{\langle \mathsf{Bob}, \mathsf{mrk} \rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})} \qquad \dfrac{\vdots}{\langle \langle \mathsf{Alice}, \mathsf{acc} \rangle, \mathsf{r01} \rangle : (\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})) \times \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{mrk} \rangle, \langle \langle \mathsf{Alice}, \mathsf{acc} \rangle, \mathsf{r01} \rangle) : (\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})) \times \mathsf{Rm}} \; \Sigma\text{-E}$$

To do that, we replace Alice with Bob, and we replace $\mathsf{acc} : \mathsf{Prj}(\mathsf{Alice})$ with $\mathsf{mrk} : \mathsf{Prj}(\mathsf{Bob})$. To use the computation rule:

$$\dfrac{\dfrac{\vdots}{\langle \mathsf{Bob}, \mathsf{mrk}\rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})} \qquad \dfrac{\vdots}{\langle\langle \mathsf{Alice}, \mathsf{acc}\rangle, \mathsf{r01}\rangle : (\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})) \times \mathsf{Rm}}}{\mathsf{elim}(\ldots) = \langle\langle \mathsf{Bob}, \mathsf{mrk}\rangle, \mathsf{r01}\rangle : (\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})) \times \mathsf{Rm}}\ \Sigma\text{-}\mathsf{E}$$

In this example, c from the elimination rule template is $\langle\langle \mathsf{Alice}, \mathsf{acc}\rangle, \mathsf{r01}\rangle$, and C is the type $(\Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})) \times \mathsf{Rm}$. Also, a and x are Bob and Alice, while b and y are mrk and acc, respectively. And just as the elimination rule says, here we constructed a type C (and we used x and y to do it), and then we rebuilt that type using a and b instead.

As with regular product types, we do not need to use either or both of x and y to construct $\mathsf{c} : \mathsf{C}$. For example, suppose we pair up Alice with the room r01:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \mathsf{Alice}, \mathsf{r01}\rangle : \mathsf{Em} \times \mathsf{Rm}}$$

Now suppose we have a pair consisting of Bob and his project mrk:

$$\dfrac{\vdots}{\langle \mathsf{Bob}, \mathsf{mrk}\rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \mathsf{Alice}, \mathsf{r01}\rangle : \mathsf{Em} \times \mathsf{Rm}}$$

The object on the right is constructed using an employee (namely Alice), but not a project. Nevertheless, we can still use the elimination rule to rebuild the pair on the right with the pair on the left. It's just that mrk from the pair on the left will not be used in the pair on the right:

$$\dfrac{\dfrac{\vdots}{\langle \mathsf{Bob}, \mathsf{mrk}\rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \mathsf{Alice}, \mathsf{r01}\rangle : \mathsf{Em} \times \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{mrk}\rangle, \langle \mathsf{Alice}, \mathsf{r01}\rangle) : \mathsf{Em} \times \mathsf{Rm}}$$

Or, using the computation rule:

$$\dfrac{\dfrac{\vdots}{\langle \mathsf{Bob}, \mathsf{mrk}\rangle : \Sigma \mathsf{x} : \mathsf{Em}, \mathsf{Prj}(\mathsf{x})} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{Em} : \mathsf{Type}}}{\mathsf{Alice} : \mathsf{Em}} \qquad \dfrac{\dfrac{}{\mathsf{Rm} : \mathsf{Type}}}{\mathsf{r01} : \mathsf{Rm}}}{\langle \mathsf{Alice}, \mathsf{r01}\rangle : \mathsf{Em} \times \mathsf{Rm}}}{\mathsf{elim}(\langle \mathsf{Bob}, \mathsf{mrk}\rangle, \langle \mathsf{Alice}, \mathsf{r01}\rangle) = \langle \mathsf{Bob}, \mathsf{r01}\rangle : \mathsf{Em} \times \mathsf{Rm}}$$

This makes it clear that when we rebuild the pair, we substitute Bob for Alice, and we simply do not use mrk. The rebuilt pair is $\langle \mathsf{Bob}, \mathsf{r01}\rangle$.

19.2 Existential quantifiers.

The type $\Sigma \mathsf{x} : \mathsf{A}, \mathsf{B}(\mathsf{x})$ pairs up any object x from A with an object from the corresponding type in the family $\mathsf{B}(\mathsf{x})$. This means the only pairs we can form to inhabit this type are those where x has at least one corresponding inhabitant in $\mathsf{B}(\mathsf{x})$.

For example, suppose that there are three objects in A, called a, b, and c. Suppose also that $\mathsf{B}(\mathsf{a})$ has two inhabitants, called m and n, $\mathsf{B}(\mathsf{b})$ has one inhabitant, called k, and $\mathsf{B}(\mathsf{c})$ has no inhabitants. Here is the correspondence from each inhabitant x in A to each inhabitant y in $\mathsf{B}(\mathsf{x})$ (if any):

$$\begin{array}{lcl} \mathsf{a} : \mathsf{A} & \longrightarrow & \mathsf{m} : \mathsf{B}(\mathsf{a}) \\ \mathsf{a} : \mathsf{A} & \longrightarrow & \mathsf{n} : \mathsf{B}(\mathsf{a}) \\ & & \\ \mathsf{b} : \mathsf{A} & \longrightarrow & \mathsf{k} : \mathsf{B}(\mathsf{b}) \\ & & \\ \mathsf{c} : \mathsf{A} & \longrightarrow & \varnothing : \mathsf{B}(\mathsf{c}) \end{array}$$

If we want to form all pairs where the first element is from A and the second element is from the corresponding type in the family $\mathsf{B}(\mathsf{x})$, then we can only form the following pairs:

$$\langle \mathsf{a}, \mathsf{m} \rangle \qquad \langle \mathsf{a}, \mathsf{n} \rangle \qquad \langle \mathsf{b}, \mathsf{k} \rangle$$

Since $\mathsf{B}(\mathsf{a})$ has two inhabitants, we can pair up a from A with either of the inhabitants of $\mathsf{B}(\mathsf{a})$. Hence, we can form the pairs $\langle \mathsf{a}, \mathsf{m} \rangle$ and $\langle \mathsf{a}, \mathsf{n} \rangle$. Since $\mathsf{B}(\mathsf{b})$ has one inhabitant, we can pair up b from B with that inhabitant, hence we can form the pair $\langle \mathsf{b}, \mathsf{k} \rangle$. However, $\mathsf{B}(\mathsf{c})$ has no inhabitants at all, so we cannot pair up c with anything.

If you like, you can think of it like this: with the dependent product, you can only form *some* of all possible pairs. In particular, you can only form pairs where the first component has a corresponding second component.

Notice that the regular product is not like this. Consider $\mathsf{A} \times \mathsf{B}$. Since the second component B is not dependent, you can pair up any element from A with any element of B. Hence, you can always form every possible pair for regular products.

But this is not so for dependent products, since the second component of the product depends on the first component, and this opens the possibility that there may not be a corresponding object for the second component.

Because of this, it is natural to view the dependent product as the existential quantifier. The inhabitants of a dependent product are only those objects from the first type that have a corresponding object from the second type. That is, the only inhabitants of $\Sigma \mathsf{x} : \mathsf{A}, \mathsf{B}(\mathsf{x})$ are those where an object from A has a satisfying or corresponding object/proof from $\mathsf{B}(\mathsf{x})$.

Here are the dependent product rules written out again, but using the existential quantifier symbol instead of the sigma:

$$\frac{\mathsf{A : Type} \qquad \mathsf{x : A} \qquad \mathsf{B(x) : Type}}{\exists \mathsf{x : A, B(x) : Type}}\ \exists\text{-}\mathsf{F}$$

$$\frac{\mathsf{A : Type} \qquad \mathsf{x : A} \qquad \mathsf{B(x) : Type} \qquad \mathsf{b : B(x)}}{\langle \mathsf{x, b} \rangle : \exists \mathsf{x : A, B(x)}}\ \exists\text{-}\mathsf{I}$$

$$\frac{\langle \mathsf{a, b} \rangle : \exists \mathsf{x : A, B(x)} \qquad \begin{array}{c} \mathsf{x : A, y : B(x)} \\ \vdots \\ \mathsf{c : C(x)} \end{array}}{\mathsf{elim}(\langle \mathsf{a, b} \rangle, \mathsf{c}) : \mathsf{C(x)}}\ \exists\text{-}\mathsf{E}$$

$$\frac{\langle \mathsf{a, b} \rangle : \exists \mathsf{x : A, B(x)} \qquad \begin{array}{c} \mathsf{x : A, y : B(x)} \\ \vdots \\ \mathsf{c : C(x)} \end{array}}{\mathsf{elim}(\langle \mathsf{a, b} \rangle, \mathsf{c}) : \mathsf{C(x)}[\mathsf{x} := \mathsf{a}, \mathsf{y} := \mathsf{b}]}\ \exists\text{-}\mathsf{Eq}$$

19.3 Projections

From the dependent product elimination rules, we can derive $\mathsf{fst}()$ and $\mathsf{snd}()$ projections too, just as we can for the regular product. To define $\mathsf{fst}()$, set C in the elimination rule template to A, and to define $\mathsf{snd}()$, set C to $\mathsf{B(x)}$. That yields the following first and second projection rules:

$$\frac{\langle \mathsf{x, b} \rangle : \Sigma \mathsf{x : A, B(x) : Type}}{\mathsf{fst}(\langle \mathsf{x, b} \rangle) : \mathsf{A}}\ \Sigma\text{-}\mathsf{E}_1 \qquad \frac{\langle \mathsf{x, b} \rangle : \Sigma \mathsf{x : A, B(x) : Type}}{\mathsf{snd}(\langle \mathsf{x, b} \rangle) : \mathsf{B(x)}}\ \Sigma\text{-}\mathsf{E}_2$$

And the computation rules:

$$\frac{\langle \mathsf{x, b} \rangle : \Sigma \mathsf{x : A, B(x) : Type}}{\mathsf{fst}(\langle \mathsf{x, b} \rangle) = \mathsf{x : A}}\ \Sigma\text{-}\mathsf{Eq}_1 \qquad \frac{\langle \mathsf{x, b} \rangle : \Sigma \mathsf{x : A, B(x) : Type}}{\mathsf{snd}(\langle \mathsf{x, b} \rangle) = \mathsf{b : B(x)}}\ \Sigma\text{-}\mathsf{Eq}_2$$

20 Negation

We will say that there is a special type that represents the impossible state of affairs of being false or absurd. We will call this type "falsum" or "bottom," and write it with the traditional bottom symbol $\perp$. Here is the formation rule:

$$\frac{}{\perp : \mathsf{Type}}\ \perp\text{-}\mathsf{F}$$

There are no elements in this type, because it represents the impossible state of affairs, i.e., the absurd state of being in contradiction. And since this type cannot be inhabited, there cannot be any introduction rules either.

However, suppose that we somehow did manage to derive an element that inhabits this type. Let's call that element b:

$$\mathsf{b} : \bot$$

We can mark x (of some type A as replaceable, to form a function:

$$\frac{\mathsf{b} : \bot \qquad \mathsf{x} : \mathsf{A}^1}{\lambda \mathsf{x} : \mathsf{A}.\mathsf{b} : \mathsf{A} \to \bot} \ \lambda\text{-}\mathsf{I}^1$$

This function takes any inhabitant of A, and it returns falsum.

As an abbreviation, we can write "$\mathsf{A} \to \bot$" as "$\neg\mathsf{A}$." Hence, this is equivalent:

$$\frac{\mathsf{b} : \bot \qquad \mathsf{x} : \mathsf{A}^1}{\lambda \mathsf{x} : \mathsf{A}.\mathsf{b} : \neg\mathsf{A}} \ \lambda\text{-}\mathsf{I}^1$$