Type Theory Handbook

JT Paasch

May 10, 2024

Contents

Chapter 1

Computation

The main question we want to ask is this: what is a computation? I think we could also use the word "calculation" to ask the same thing: what is a calculation?

There are different answers to this question. We will focus on one of them. It is based on the lambda calculus and type theory. Later, we will examine the lambda calculus and type theory in detail.

But to begin, in this chapter let's think about what a "computation" is in an informal may, so that we have a rough idea of how the lambda calculus conceives of a computation.

1.1 Solving an Arithmetic Equation

Think about how you would solve a simple arithmetic equation, using pencil and paper. For instance, take this equation:

$$(1+2)*(3+(1*2)) \tag{1.1}$$

In our minds, we can break this up into parts, each of which we can solve separately:

$$(1+2) \qquad * \qquad (3 \qquad + \qquad (1*2))$$

First, we might replace $(1*2)$ with 2:

$$\begin{tikzcd}
(1+2) & * & (3 & + & (1*2)) \arrow[d] \\
(1+2) & * & (3 & + & 2)
\end{tikzcd}$$

Then we might replace $(1+2)$ with 3:

$$\begin{tikzcd}
(1+2) \arrow[d] & * & (3 & + & 2) \\
3 & * & (3 & + & 2)
\end{tikzcd}$$

Then we would replace $(3+2)$ with 5:

$$\begin{tikzcd}
3 & * & (3 & + \arrow[d] & 2) \\
3 & * & & 5 &
\end{tikzcd}$$

And finally, we would replace $3 * 5$ with 15:

$$\begin{tikzcd}
3 & * \arrow[d] & 5 \\
 & 15 &
\end{tikzcd}$$

What can we observe about this process? Here are some points to notice:

- The whole equation is made up of many parts.
- Some parts can be replaced by other parts (e.g., $(1+2)$ can be replaced with 3).
- We can perform such replacements multiple times.
- Sometimes, new replacements become possible after we perform another replacement (e.g., after we replace $(1*2)$ with 2, we then get $(3+2)$, which wasn't there before, and it can be replaced too).
- The process terminates. That is, there is a point where we cannot replace any more parts of the equation.

This is an example of computation or calculation. We compute the answer by successively replacing parts of the equation with other parts, until we cannot make any more replacements.

1.2 Calculation by Rewriting

Let us look at this same process in a slightly different way. Let us think about this process not as a process of solving parts of an equation, but rather as a process of rewriting.

To do that, think of the equation as nothing more than a string of characters: it is just a sequence of symbols (what they mean is irrelevant). We can break this string up into smaller chunks (subsequences). For instance, take the subsequence $(1 * 2)$:

$$(1 + 2) * (3 + \underline{(1 * 2)})$$

We can replace that subsequence with a new subsequence, namely 2:[1]

$$(1 + 2) * (3 + \underline{2})$$

Then we can take the subsequence $(3+2)$ and replace it with a new subsequence: 5.

$$(1 + 2) * \underline{5}$$

We can then take the subsequence $(1 + 2)$ and replace it with 3:

$$\underline{3} * 5$$

And finally, we can replace the sequence $3 * 5$ with 15:

$$15$$

This time around, we are talking about sequences and subsequences of symbols, rather than parts of an equation. Still, we can observe the same properties as before:

- The whole sequence of symbols is made up of many subsequences.
- Some subsequences can be replaced by other subsequences (e.g., $(1 + 2)$ can be replaced with 3).
- We can perform such replacements multiple times.
- Sometimes, new replacements become possible after we perform another replacement (e.g., after we replace $(1*2)$ with 2, we then get $(3+2)$, which wasn't there before, and it can be replaced too).
- The process terminates. That is, there is a point where we cannot replace any more subsequences.

Notice that we don't have to know that the above sequence of symbols is an equation. The symbols we used could have been any symbols at all, but we could still walk through the same process of taking certain subsequences and replacing them with other sequences.

[1] 2 is a sequence that contains only one symbol. In some situations, we might also replace a subsequence with an empty subsequence, i.e., a blank sequence with no symbols at all.

1.3 String Replacment

Let's pursue the above line of thought with an even more radical example. Suppose we have an arbitrary string of characters:

`ebceeb`

We could, in principle, do some kind of replacement/reduction on this too. For instance, we could say that we want to take every `e`:

$\underline{\texttt{e}}\texttt{bc}\underline{\texttt{ee}}\texttt{b}$

And then replace each one with a zero:

$\underline{\texttt{0}}\texttt{bc}\underline{\texttt{00}}\texttt{b}$

Next, we could take every `b`:

$\texttt{0}\underline{\texttt{b}}\texttt{c00}\underline{\texttt{b}}$

And replace each one with nothing (an empty sequence):

`0c00`

Finally, we might take every subsequence of `0c`:

$\underline{\texttt{0c}}\texttt{00}$

And replace it with a one:

$\underline{\texttt{1}}\texttt{00}$

This would be a kind of reduction or rewrite process too, in just the same sense as before:

- The whole sequence of symbols is made up of many subsequences.
- Some subsequences can be replaced by other subsequences (e.g., `e` can be replaced with `0`).
- We can perform such replacements multiple times.
- Sometimes, new replacements become possible after we perform other replacements (e.g., after we replace `e` with `0` and `b` with nothing, we then get `0c`, which wasn't there before, and it can be replaced too).
- The process terminates. That is, there is a point where we cannot replace any more subsequences.

So this too counts as a computation or calculation in just the way that the other examples did. But this example is more radical because there is no obvious meaning to the symbols. In this case, we are doing nothing but mindless symbol manipulation, of the sort that a computer does.

1.4 The Basic Operations of Rewriting

In every one of the above examples, we performed three basic operations. Here they are:

- Identify a sequence of symbols to replace: in each case above, we said something like this: "the sequence xyz can be replaced."
- Stipulate what to replace it with: in each case we said something like this: "replace the replaceable sequence with abc."
- Do the actual replacement: in each case, we then took the sequence to replace, we deleted it, and we replaced it with the stipulated replacement.

Think of this at the most basic level. By "basic," I mean: imagine that we can only replace one symbol at a time, so that complex replacements are accomplished by stringing together multiple one-symbol replacements. The operations are still the same:

- Identify a symbol to replace.
- Stipulate a replacement.
- Do the replacement.

From these three basic operations, we can perform any of the computations or calculations that we did above. In fact, we can model pretty much any computation at all with these three simple operations.

1.5 Computation Generally

At a very general level, we have here a process where we replace symbols with other symbols, and this is independent of whatever the meaning of those symbols might be.

This is the basic idea behind a "calculation" or "computation," as we will conceive of it here. A **computation** is a finite set of replacements on a sequence of symbols.

The **lambda calculus** is an attempt to formalize this process of calculation. In certain ways, it is very simple. Nevertheless, with the lambda calculus we can build up a remarkable number of things.

Later, we will add types to the lambda calculus, which will restrict its generality, but will give us some very useful power and precision.

Chapter 2

The Lambda Calculus (Informally)

There are three basic operations in the lambda calculus: abstraction, application, and β-reduction. These correspond to the three basic operations of rewriting that we mentioned in the last chapter. Abstraction is when you identify a symbol to replace, application is when you stipulate a replacement, and β-reduction is when you do the replacement.

2.1 Abstraction

Suppose I need to repeat some trivial calculation a lot. For instance, suppose that each day this week I need to buy 2 pounds of sand, at 5 cents per pound. I'm a forgetful person, so I write it down:

$$\text{multiply 2 by 5} \tag{2.1}$$

Suppose that next week, the price goes up to 7 cents per pound, so I cross out my note and write down:

$$\text{multiply 2 by 7} \tag{2.2}$$

The third week it is 4 cents per pound:

$$\text{multiply 2 by 4} \tag{2.3}$$

After a while, I wise up and realize I can make a formula out of this. To do that, I can use a name instead of a number for the price:

$$\text{multiply 2 by } \underline{\text{price}} \tag{2.4}$$

Then I can indicate up front that I intend for the word "price" to be replaced:

$$\text{replace “price” in (multiply 2 by price)} \tag{2.5}$$

In lamba calculus terminology, this is called an **abstraction**. We say we **abstract** “price” from the body of the original expression. What we have done here is identified the symbol (or in this case, word) that is replaceable in the original expression.

2.2 Application

Next, we can stipulate what value should be used to replace “price.” For instance, to calculate the cost if the sand is 5 cents per pound:

$$\text{replace “price” with “5” in (multiply 2 by price)} \tag{2.6}$$

We could substitute another value in for “price” too. For instance, the price could be 7:

$$\text{replace “price” with “7” in (multiply 2 by price)} \tag{2.7}$$

In lambda calculus terminology, this is called an **application**. We say that we **apply** the expression to 7 (or to 5 in the previous case). What we have done here is stipulated what the replacement is.

Note: the term “application” can be misleading. We do not actually apply anything here. All we do with an “application” is we stipulate the value that should be used to replace “price”; but we do not do any replacing here.

2.3 Beta Reduction

Consider the above application:

$$\text{replace “price” with “5” in (multiply 2 by price)} \tag{2.8}$$

We can do the actual replacement — we can put “5” in the place of “price” — to get this:

$$\text{multiply 2 by 5} \tag{2.9}$$

Or we can put “7” in the place of “price,” which comes out to this:

$$\text{multiply 2 by 7} \tag{2.10}$$

In lambda calculus terminology, when we replace the names with stipulated replacements in the body of the expression, we call this a β**-reduction** (or just “beta reduction” if you don’t want to write the Greek character).

2.4 Nesting Abstractions and Applications

You can nest abstractions and applications. Take the abstraction we made above:

$$\text{replace “price” in (multiply 2 by price)} \tag{2.11}$$

We can abstract the weight too:

$$\text{replace “weight” in (replace “price” in (multiply weight by price))} \tag{2.12}$$

Notice how the one abstraction is nested inside the other. We can nest applications too. Let us stipulate that we want to replace "weight" with "3":

$$\text{replace “weight” with “3” in (replace “price” in (multiply weight by price))} \tag{2.13}$$

When we enact the β-reduction (i.e., when we put "3" in place of "weight"), we get this:

$$\text{replace “price” in (multiply 3 by price)} \tag{2.14}$$

We can then stipulate that we want to replace "price" with "7":

$$\text{replace “price” with “7” in (multiply 3 by price)} \tag{2.15}$$

And once we actually do the β-reduction, we get:

$$\text{multiply 3 by 7} \tag{2.16}$$

Chapter 3

Replacements (Informally)

3.1 Free and Bound Names

When you formulate an abstraction, you identify a name that is replaceable. Once you identify a name as a name that can be replaced, we say that the name is **bound** by the abstraction.

In the informal formulation we have been using so far, we know a name is bound when we see the phrase "replace [name] in" up front. That tells us that "[name]" is bound. For example, consider our earlier abstraction:

$$\text{replace “price” in (multiply 2 by price)} \tag{3.1}$$

Here we have bound the word "price." The phrase

$$\text{replace “price” in } (\dots) \tag{3.2}$$

is called a **binding expression**.

Anything in the expression that is not bound is called **free**. So, the words "multiply," "2," and "by" are all free. The only word that is bound in the above abstraction is "price."

Suppose we take away the abstraction, and look at the original expression by itself:

$$\text{multiply 2 by price} \tag{3.3}$$

What about "price" now? Is it bound here? No, it is not. And we know that because there is no binding expression up front that says "replace price in." So, by itself, "price" is free in the original expression. But when it appears in the abstraction — i.e.,

$$\text{replace “price” in (multiply 2 by price)} \tag{3.4}$$

— then the word "price" is not free. Rather, it is bound.

3.2 Capturing Names

When you nest abstractions, parent abstractions can bind names that were free in a child expression. We say that the parent abstraction **captures** the free name from the child.

Consider this abstraction:

$$\text{replace “price” in (multiply weight by price)} \tag{3.5}$$

Here the word “price” is bound by the binding expression. But the word “weight” is not bound. It is free.

Now suppose we nest this whole abstraction inside another abstraction, one that binds the word “weight”:

$$\text{replace “weight” in (replace “price” in (multiply weight by price))} \tag{3.6}$$

Now the word “weight” is bound, by the outermost binding expression. That word was free before, but here it gets bound. We say that the parent binding expression captured it, and bound it.

3.3 Not Just Any Replacement

In a β-reduction, you replace a bound name with another value. If you are not careful though, you can unintentionally bind a name that isn't supposed to be bound.

Consider again this nested abstraction:

$$\text{replace “weight” in (replace “price” in (multiply weight by price))} \tag{3.7}$$

What happens if we stipulate that we want to replace the bound “weight” with the word “price”? Like this:

$$\text{replace “weight” with “price” in (replace “price” in (multiply weight by price))} \tag{3.8}$$

When we do the β-reduction, we get this:

$$\text{replace “price” in (multiply price by price)} \tag{3.9}$$

Here, we have replaced the free word “weight” with the word “price.” But “price” is bound in this expression, so now we have unintentionally bound a part of the expression that wasn't bound before.

For this reason, in the lambda calculus, you can't use any replacements that are **already bound** in the expression. In our example here, the word “price” is a forbidden replacement, since it is already bound in the expression.

Chapter 4

Equivalence (Informally)

4.1 α-conversion

Consider our earlier abstraction:

$$\text{replace “price” in (multiply 2 by price)} \tag{4.1}$$

We could replace the word “price” with something else. For instance:

$$\text{replace “}x\text{” in (multiply 2 by }x\text{)} \tag{4.2}$$

We could even replace “price” with some random word:

$$\text{replace “bargle” in (multiply 2 by bargle)} \tag{4.3}$$

When we rename the bound names in an expression like this, it is called an **α-conversion**. For instance, replacing “price” with “x” as we did above is an α-conversion.

We say that these terms are **α-variants** of each other. For instance, the “price” version and the “x” version are α-variants.

4.2 Computational Equivalence

α-variants are equivalent, in the sense that when you do a β-reduction, you get the same result. Take our earlier abstraction:

$$\text{replace “price” in (multiply 2 by price)} \tag{4.4}$$

Replace the bound name with 5:

$$\text{replace “price” with “5” in (multiply 2 by price)} \tag{4.5}$$

Do the β-reduction, and you get:

$$\text{multiply 2 by 5} \tag{4.6}$$

Now take the expression where we used x as the bound name, and do the same:

$$\text{replace “}x\text{” in (multiply 2 by }x\text{)} \tag{4.7}$$
$$\text{replace “}x\text{” with “5” in (multiply 2 by }x\text{)} \tag{4.8}$$
$$\text{multiply 2 by 5} \tag{4.9}$$

Finally, try it with the “bargle” variant:

$$\text{replace “bargle” in (multiply 2 by bargle)} \tag{4.10}$$
$$\text{replace “bargle” with “5” in (multiply 2 by bargle)} \tag{4.11}$$
$$\text{multiply 2 by 5} \tag{4.12}$$

In each case, the very same computation happens. The name we give to the bound name does not matter, because it is always replaced in the same way.

Another way to say this is that these α-variants can be **substituted** for each other. They **preserve** the same computation.

4.3 The Class of Equivalent Terms

Each of the above α-variants are equivalent. We can collect them together, and say, “these are all equivalent.”

We can add many other expressions to the class. Just put in a new bound name — change “price” to “y,” “$\circ$,” “**M**,” “slapf,” or whatever — and you have a new α-variant.

So, there is a bundle of variations that are all basically the same term, through α-equivalence. Mathematicians sometimes use the word “modulo” instead of “through”: these terms are equivalent **modulo α-equivalence**.

The crucial thing to note is that the same β-reductions apply to any variant in the class. Whatever you can compute with one variant, you can compute with the other variants.

In practice, writers often just pick a binding name, e.g., “x,” and then talk about that selected α-variant as *the* expression. But really, they mean it as a representative; there is a large collection of other variants which could be substituted in at any time.

Chapter 5

λ Variables

The untyped lambda calculus is a formal way of doing everything that we stalked about informally earlier. Instead of writing "untyped lambda calculus," I will say λ.

Let us first establish the notation — i.e., how you actually write expressions down. λ expressions are built up from what are called **terms**. Let us start with the simplest type of λ term. These are called variables.

5.1 Variables

Earlier, we used English words (and sometimes symbols like x) to build expressions. But formally, we do not need to use actual words. We only need symbols.

So, let us say that our alphabet is a collection of symbols that we can pick from. Let us say we can pick any lowercase letter:

$$a, b, c, \ldots, v, x, y, z \tag{5.1}$$

If we need to, we can attach subscripted numbers to make more. For instance:

$$v_1, v_2 \ldots, x_1, x_2, \ldots x_{56}, x_{57}, \ldots \tag{5.2}$$

In λ, these are called **variables**. Be careful here. The word "variable" means something different in, say, first-order logic, or in programming languages. In λ, it refers to any single symbol taken from the alphabet we specified here.

The simplest expression you can build in λ consists of a single variable. To construct such an expression, select a variable from the alphabet and wrap it in parentheses. For instance:

$$(x) \tag{5.3}$$

The following are all valid λ variable terms:

$$(y) \tag{5.4}$$

$$(a) \tag{5.5}$$

$$(v_9) \tag{5.6}$$

5.1.1 Dropping Parentheses

We wrap terms in parentheses in order to avoid ambiguity. One of the things that parentheses do is tell us where a term begins and where it ends.

Another thing that parentheses do is tell us how symbols should be grouped together. In a moment, we will see cases where this is important.

But for a single variable, there is no grouping. Hence, when the term is a single variable term, there is no confusion if you omit the parentheses. For instance, this:

$$(x) \tag{5.7}$$

can easily be written like this:

$$x \tag{5.8}$$

and nobody will be confused about it.

In practice, most writers omit parentheses around a single-variable term, and we will too.

However, this is just a typographical convenience. If anyone demands that we write down the term in the fully correct way, we should always be able to put the parentheses back.

Chapter 6

Abstractions

Earlier, we talked about abstractions. Abstractions occur when you select a name (or symbol) from an expression, and stipulate that it can be replaced. Our example was this:

$$\text{replace “price” in (multiply 2 by price)} \tag{6.1}$$

We can put this more generally. Let us use the symbol ϕ to stand in the place of *whatever* word or symbol we want to mark as replaceable (e.g., “price”), and let us use the symbol M to stand in place of *whatever* expression we want to make the replacements in (e.g., “multiply 2 by price”). Then we can write it like this:

$$\text{replace } \phi \text{ in } (M) \tag{6.2}$$

In λ, we can write this more compactly. The notation looks like this:

$$\lambda\phi.(M) \tag{6.3}$$

The lower case Greek λ (lambda) character indicates “replace,” and in fact this is where the lambda calculus gets its name.

Be careful when you see this λ character. Sometimes I use it to refer to the whole system that we call the untyped lambda calculus. But other times, I use it as part of an abstraction term. The context should always make it clear which I mean.

6.1 Forming the Terms

To construct an **abstraction**, you take any term, and you mark a variable value as replaceable. For instance, take this term, which consists of a single variable:

$$(x) \tag{6.4}$$

Let us suppose that we want to make x replaceable. To mark the value x as replaceable, put λx up front, followed by a dot:

$$\lambda x.(x) \tag{6.5}$$

finally wrap the whole thing in parentheses:

$$(\lambda x.(x)) \tag{6.6}$$

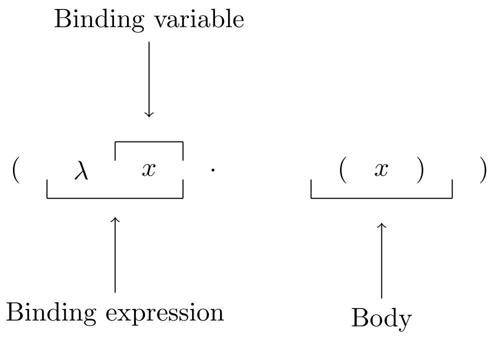
That says:

$$\text{Replace "}x\text{" in } (x) \tag{6.7}$$

Here is some terminology:

- "$\lambda x.$" is called the **binding expression**.
- The "x" in "$\lambda x.$" is called the **binding variable**.
- Everything after the binding expression is called the **body** of the abstraction.

Here is a diagram of all these parts, labeled:



6.1.1 Nested Terms

The body of an abstraction can be any term. In our last example, it was a single variable. But it could be a complex term, like another lambda term. For instance:

$$(\lambda x.(\lambda y.(x))) \tag{6.8}$$

We will discuss applications next, but note here that the body of an abstraction could also be an application.

6.1.2 Dropping Parentheses

As with variable terms, we can drop parentheses in abstractions if it leads to no ambiguity. Consider this abstraction:

$$(\lambda x.(x)) \tag{6.9}$$

We can drop the outer parentheses, and the expression is still unambiguous:

$$\lambda x.(x) \tag{6.10}$$

We can also drop the parentheses around the body of the expression:

$$\lambda x.x \tag{6.11}$$

Now consider this term:

$$(\lambda x.(\lambda x.(x))) \tag{6.12}$$

We can drop the outer parentheses without losing clarity:

$$\lambda x.(\lambda x.(x)) \tag{6.13}$$

We can also drop the rest of the parentheses without losing clarity:

$$\lambda x.\lambda x.x \tag{6.14}$$

In these cases, we can freely drop parentheses without losing clarity. But in a moment we will discuss applications, and there we will encounter cases where we need to include parentheses to avoid ambiguity.

Chapter 7

λ Applications

When we talked about λ informally above, we talked about applications. An **application** occurs when you stipulate a value to put in place of a bound name. We did this by saying what the replacement should be. For example:

$$\text{replace “price” with “5” in (multiply 2 by price)} \tag{7.1}$$

Here we say that the value “5” should be put in place of “price.” We say that value “5” is the **argument**.

In λ, we write this slightly differently. We put the argument after the formula, something like this:

$$\text{replace “price” in (multiply 2 by price) 5} \tag{7.2}$$

Of course, we can represent this more generally. We can use a symbol to stand in place of the whole formula, minus the argument. Let M stand for the formula. Then we can write:

$$M\ 5 \tag{7.3}$$

We could use another symbol to stand for the argument too. Let N stand for the argument. Then we can write:

$$MN \tag{7.4}$$

That is the basic notation we use in λ to write applications.

7.1 Forming the Terms

To construct an **application** in λ, put any two terms next to each other, and wrap the whole of them in parentheses. For instance:

$$((\lambda x.(x))(y)) \tag{7.5}$$

Here the first term is $(\lambda x.(x))$ and the second term is (y). We say that we **apply** the first term to the second term, and as we noted a moment ago, we say that the second term is the **argument**.

In this case, the first term is an abstraction. But in λ this need not be so. *Any* two terms can be put next together to form an application. Here we put two variable terms next to each other — (x) and (y):

$$((x)(y)) \tag{7.6}$$

Here are some other examples:

$$((a)(b)) \tag{7.7}$$
$$((x)(\lambda x.(y))) \tag{7.8}$$
$$((\lambda x.((a)(b)))((v_1)(v_{10}))) \tag{7.9}$$

7.1.1 Nested Applications

Notice that I said an application consists of any two *terms* next to each other. An application is itself a term. So you can make an application from other applications. For instance:

$$(((x)(y))(x)) \tag{7.10}$$

The first term of the application is $((x)(y))$ — which is itself an application of (x) to (y) — and the second term of the application is (x). So we apply $((x)(y))$ to (x).

7.1.2 Dropping Parentheses

Parentheses can be dropped if no ambiguity comes of it. Consider this:

$$((x)(y)) \tag{7.11}$$

We could drop the outer parentheses, and there is no ambiguity:

$$(x)(y) \tag{7.12}$$

And we could drop the parentheses around the variables too:

$$xy \tag{7.13}$$

But now consider this:

$$(((x)(y))(x)) \tag{7.14}$$

Suppose you drop all the parentheses:

$$xyx \tag{7.15}$$

Now it is not clear what this expression means. It could be this:

$$(xy)x \tag{7.16}$$

Or this:

$$x(yx) \tag{7.17}$$

So in this case, we need at least one set of parentheses to make it unambiguous. Here is another example, with all parentheses dropped:

$$\lambda x.xy \tag{7.18}$$

This is ambiguous. It could mean this:

$$\lambda x.(xy) \tag{7.19}$$

Or it could mean this:

$$(\lambda x.x)y \tag{7.20}$$

So here too, we need at least one set of parentheses to make the expression unambiguous.

Writers often stipulate a convention so they don't have to write so many parentheses. For instance, they often say that you should always assume the parentheses group to the left. Hence, this:

$$xyx \tag{7.21}$$

means this:

$$(xy)x \tag{7.22}$$

It is always a good idea to read an author's notes about their notation, so you can be clear about their parentheses conventions.

We will not adopt a convention here. But we will drop parentheses if it results in no ambiguity. If strict correctness is demanded, we should always be able to put the parentheses back.

Chapter 8

Variable Occurrences

Variables can occur in multiple places inside λ terms. For instance, they can occur on their own, as single variable terms. They can occur in applications and in the body of abstractions. And they can occur as binding variables. It is important to keep all these types of occurrences separate.

8.1 Free and Bound Variables

In an abstraction, there is a binding variable. In the body of the abstraction, we say that every occurrence of that value is **bound** by that binding variable. Every other variable that occurs in the body is **free**. Consider this abstraction:

$$\lambda x.(xy) \tag{8.1}$$

The body of the expression is (xy). The x that occurs there is bound by λx. The y that occurs there is free.

Note, however, that a binding variable binds *every occurrence* of a value. Consider this abstraction:

$$\lambda x.(x(xy)) \tag{8.2}$$

Here the body of the abstraction is $(x(xy))$. Now we have two occurrences of x. Both are bound by λx, while y is still free.

A bound value does not need to occur in the body of the expression. This is a valid abstraction:

$$\lambda x.y \tag{8.3}$$

But there is no occurrence of x in the body of this abstraction.

8.1.1 Free Inside, Bound Outside

Consider this abstraction:

$$\lambda x.(xy) \tag{8.4}$$

In this term, the body is (xy), and the x is bound by λx. The y is free.

Now consider the body of the abstraction all by itself, without the abstraction:

$$(xy) \tag{8.5}$$

Is x bound or free? In this case, it is free, since there is no binding expression.

Writers sometimes speak in the same sentence about how variables are both bound and free. On a first read, you might think, "how can it be bound *and* free?"

Well, the writer simply means that a variable is free "inside" the body of the expression, when we consider the body of the expression all by itself, but the variable is bound from the "outside" by an abstracting binding expression.

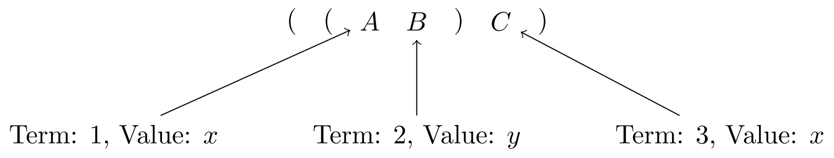
8.2 Multiple Occurrences of a Term

It is important to keep the value of a variable separate from the occurrence of the variable. Consider this term again:

$$((xy)x) \tag{8.6}$$

The same variable (x) appears twice. Technically, we say that there are multiple **occurrences** or multiple **instances** of the same **value**.

We could rewrite the above to be more explicit, something like this:



That makes it clearer that we have three separate variable terms here, and each has their own value. The value of 1 and 3 is x, and the value of 2 is y.

Chapter 9

Terms

Let us define λ terms formally.

9.1 The Format

As we saw before, there are three types of lambda terms (parentheses are dropped where it leaves no ambiguity):

- Variables — e.g., a, b, x, y, v_1, v_2, etc.
- Applications – e.g., xy, ab, $(ab)x$, $(xx)(xy)$, etc.
- Abstractions – e.g., $\lambda x.x$, $\lambda x.(xy)$, $\lambda y.(x(ab))$, etc.

Applications and abstraction terms can have other terms nested inside of them. For instance, an application has this form:

$$MN \tag{9.1}$$

where M and N are placeholders that can stand for *any* λ terms. For instance, these are all valid applications:

- xy — so $M = x$ and $N = y$.
- $(xy)y$ — so $M = (xy)$ and $N = y$.
- $x(\lambda x.(ab))$ — so $M = x$ and $N = (\lambda x.(ab))$.

An abstraction has this form:

$$\lambda\phi.M \tag{9.2}$$

where ϕ is a placeholder for any value of a variable term, and M is a placeholder for any lambda term. These are all valid abstractions:

- $\lambda x.(x)$ — so $\phi = x$ and $M = (x)$.
- $\lambda y.(ab)$ — so $\phi = y$ and $M = (ab)$.
- $\lambda a.((\lambda x.x)a)$ — so $\phi = a$ and $M = ((\lambda x.x)a)$.

9.2 Definition

With this information at hand, let us formulate a definition of lambda terms. Let us say that a lambda term is any string of characters that are built by using one of the following rules:

- Take the set of possible variables $a, b, c, \ldots$ (we call this the alphabet we can choose from), pick any one of them, and surround it with parentheses, e.g., (a) or (b).
- Take any two terms, call them M and N, put them next to each other, and surround them with parentheses, e.g., (ab) or $((bc)a)$.
- Take the value of any variable, call it ϕ, and take any term, call it M, and put them together like this: $(\lambda\phi.M)$, e.g., $(\lambda a.(a))$ or $(\lambda b.(ab))$.

Here it is, more exactly:

Definition 1 (The set Λ of all lambda terms)**.** *Let V be an infinite set of symbols, possibly with subscripts: $V = a, b, c, \ldots x, y, z, \ldots v_1, v_2, \ldots$. Then Λ is the set of all terms that satisfy the following conditions:*

- *If $\phi \in V$, then $(\phi) \in \Lambda$ (call this type of term a **variable**).*
- *If $M, N \in \Lambda$, then $(MN) \in \Lambda$ (call this type of term an **application**).*
- *If $\phi \in V$ and $M \in \Lambda$, then $(\lambda\phi.M) \in \Lambda$ (call this type of term an **abstraction**).*

9.3 Explanation

V is the set of all symbols in our alphabet, and the capital Greek letter Λ (Lambda) is the set of all λ terms. According to this definition, all the terms in Λ are of three types.

- First, there are variable terms, which are formed by taking a symbol ϕ from V (in set theory notation, we write: $\phi \in V$, where $\in$ means "in"), and putting parentheses around it, like this: (ϕ). Any term formed that way is in Λ. (In set theory notation: $(\phi) \in \Lambda$.)
- Second, there are application terms, which are formed by taking any two terms M and N from Λ (in set theory notation: $M, N \in \Lambda$), then putting them next to each other, and wrapping them in parentheses, like his: (MN).

- Third, there are abstraction terms, which are formed by taking the value of any symbol ϕ in V (in set theory notation: any $\phi \in V$) and taking any term M in Λ (in set theory notation: $M \in \Lambda$), and putting them together like this: $(\lambda\phi.M)$.

Notice that there are an infinite number of terms in Λ:

- Variables can be formed by taking symbols out of V. But there are an infinite number of symbols in the alphabet V. So there are an infinite number of variable terms in Λ.
- Applications and abstractions can be formed by taking any other terms from Λ. So, you can form terms out of terms out of terms out of terms, infinitely.

Of course, as we have said, we can drop parentheses when it leads to no ambiguity, but if absolute strictness is ever demanded of us, we should always be able to put the parentheses back in.

Chapter 10

Subterms

Application terms and abstraction terms can be built from other λ terms. So, terms can have subterms. Let us define subterms formally.

10.1 The Terms in a Variable

The subterms of a λ term will include all the terms it is built from, plus itself. Consider a variable term first:

$$(x) \tag{10.1}$$

There is just one term here, namely the variable term (x). So the list of subterms is this:

- (x)

10.2 The Terms in an Application

Consider an application term like this one:

$$(xx) \tag{10.2}$$

This is an application term, but it is composed of two variable terms: x and x. The total list of subterms in this term are these:

- x
- x
- (xx)

There are three subterms here. There are two occurrences of the variable x, and there is the larger whole, which is the application (xx). We can diagram this, in a tree format:

$$\begin{tikzcd}
& \text{appl.} \arrow[dl] \arrow[dr] & \\
x & & x
\end{tikzcd}$$

You can tell how many subterms are in this term by counting the nodes in the tree. In this case, there are three nodes: the top node (the application), plus the two branches (the two variable terms).

10.3 The Terms in an Abstraction

Consider this term:

$$\lambda x.(xy) \tag{10.3}$$

This is an abstraction term, but the body of the abstraction is an application composed of two variable terms. So the total list of subterms is this:

- x
- y
- (xy)
- $\lambda x.(xy)$

We can diagram that like this:

$$\begin{tikzcd}
& \lambda x \arrow[d] & \\
& \text{appl.} \arrow[dl] \arrow[dr] & \\
x & & y
\end{tikzcd}$$

Again, you can determine how many subterms there are here by counting the number of nodes. There are four: the top node (the abstraction), its child (the application), plus the two branches (the two variables).

10.4 Sets or Multisets?

On a first pass, you might think we could say the subterms of a term is the *set* of terms that the term is built from. However, a set cannot have any repetitions. Consider this term again:

$$(xx) \tag{10.4}$$

When we list the subterms here, we get this list:

- x
- x
- (xx)

Now suppose we put those into a set:

$$\{(x), (x), (xx)\} \tag{10.5}$$

Of course, repetitions don't matter in a set, so we remove the duplicate (x) to get this:

$$\{(x), (xx)\} \tag{10.6}$$

Unfortunately, that does not represent all the subterms. It's missing one!

To allow for repetitions, we need to use a multiset. A **multiset** is a set that allows multiple occurrences of the same item. (That is why a multipset is called a *multi*set — the name is short for a *mult*iple membership *set*.)

So, we can say that the subterms are the *multiset*:

$$\{(x), (x), (xx)\} \tag{10.7}$$

10.5 Definition

With the above information in mind, let us define subterms formally. Let us say that the subterms of a term is the multiset of terms it is built from. There are three types of terms a term can be built from, so we need to enumerate each case:

- If the term is a variable (x), then the multiset will contain just that variable.
- If the term is an application (MN), then the multiset will contain all the subterms contained in M, and all the subterms contained in N, plus the application (MN) itself.
- If the term is an abstraction $(\lambda\phi.M)$, then the multiset will contain all the subterms contained in M, plus the abstraction $(\lambda\phi.M)$ itself.

We can unite multiple multisets into one big multiset. We call this the **union** of the multisets. The union of two multisets will include every element in each set. We write it like this: the union of multisets A and B is notated as $A \cup B$.

Here is our definition:

Definition 2 (The multiset Sub of subterms in a λ term M)**.** *Let M be a lambda term, and let V be the set of variables defined above. The multiset Sub of subterms in M (notated as $Sub(M)$) is the smallest multiset that satisfies the following conditions:*

- *If M is a variable $x \in V$, then $Sub(x)$ is $\{x\}$.*
- *If M is an application of the form MN, then $Sub(MN)$ is the union of the multisets $Sub(M)$, $Sub(N)$, and $\{(MN)\}$, i.e., $Sub(M) \cup Sub(N) \cup \{(MN)\}$.*
- *If M is an abstraction of the form $\lambda\phi.M$, then $Sub(\lambda\phi.M)$ is the union of the multisets $Sub(M)$ and $\{(\lambda\phi.M)\}$, i.e., $Sub(M) \cup \{(\lambda\phi.M)\}$.*

10.6 Proper Subterms

The subterms of a term include the larger term itself. If we want to talk about just the subterms, without the larger term itself, we can call those terms the **proper subterms** of the term. We can define that like this:

Definition 3 (Proper subterms)**.** *Let M be a lambda term. For any subterm $N \in Sub(M)$, N is a proper subterm of M if $N \neq M$.*

So, take any subterm in a term M, and that subterm is a proper subterm if it is not identical to M.

Chapter 11

Free Variables

The purpose of an abstraction term is to stipulate a value that can be replaced in another term. It has this form:

$$\lambda x.M \tag{11.1}$$

This is a short hand notation for this: "replace 'x' in M." An abstraction like this has three parts:

- The **binding expression**, which is λx.
- The **binding variable**, which is x.
- The **body** of the expression, which is M.

Any occurrence of x inside M is **bound** (by the binding expression λx). Any other variable that occurs in M is said to be **free**. Let us define bound and free variables formally.

11.1 Finding all Free Variables

The free variables in a λ term will include every variable that is not bound. Let us refer to thihs set with the name FV, and we will say that the set of free variables in a term M is $FV(M)$.

Notation 1. *$FV(M)$ stands for the set of free variables FV in a λ term M.*

There are three different types of terms: variables, applications, and abstractions. Let us consider how we gather the free variables for each case.

11.1.1 Variable Terms

A variable term has this form:

$$(\phi) \tag{11.2}$$

What are the free variables in this expression? Well, it is the variable itself, x. And this will be true for any variable term at all. If a term is just a variable, that variable will be free.

Let ϕ be a placeholder that stands for any variable. Then we can say this: take any variable term of this form:

$$(\phi) \tag{11.3}$$

The set of free variables in this expression will consist only of ϕ:

$$FV(\phi) = \{\phi\} \tag{11.4}$$

11.1.2 Application Terms

An application has this form:

$$(xy) \tag{11.5}$$

What are the free variables in this expression? Well, there are two: x and y. So the set of free variables in this case are these:

$$FV((xy)) = \{x, y\} \tag{11.6}$$

But let's think more generally about this. An application is really just two terms, next to each other. So this is really the set of free variables in the first term, plus the set of free variables in the second term:

$$FV((xy)) = FV(x) \text{ plus } FV(y) \tag{11.7}$$

In set theory terminology, we can say that the free variables in the application are the *union* of the free variables of the first term and the second term:

$$FV((xy)) = FV(x) \cup FV(y) \tag{11.8}$$

In this case, the two terms in the application are x and y. But the same thing holds no matter the terms are. Suppose we have an application like this:

$$((xy)y) \tag{11.9}$$

Now the first term is itself an application — (xy) — while the second term is a variable y. Nevertheless, the free variables in this expression will be the set of free variables in the first term, plus the set of free variables in the second term:

$$FV(((xy)y)) = FV((x, y)) \cup FV(y) \tag{11.10}$$

And of course, we could then reduce this further, and get the free variables for each of x and y in the innermost application:

$$FV(((xy)y)) = \tag{11.11}$$
$$FV((x,y)) \cup FV(y) = \tag{11.12}$$
$$(FV(x) \cup FV(y)) \cup FV(y) \tag{11.13}$$

We can keep going down into nested application like this, to figure out what the free variables are in an application.

To formulate this in a general way, let us say that M and N are placeholders that can stand for any λ terms. Then, take any application of this form:

$$(MN) \tag{11.14}$$

The set of free variables in this expression will be the set of free variables in M, plus the set of free variables in N:

$$FV((MN)) = FV(M) \cup FV(N) \tag{11.15}$$

11.1.3 Abstraction Terms

An abstraction has this form:

$$\lambda\phi.M \tag{11.16}$$

where ϕ is a placeholder for any variable value and M is a placeholder for any λ term.

The free variables in any such expression will be all the free variables in M, except for the one that is bound — ϕ. Like this:

$$FV(\lambda\phi.M) = FV(M) \text{ minus } \{\phi\} \tag{11.17}$$

In set theory, when we want to talk about one set, call it A, minus another set, call it B, we name that **set difference**. We usually notate it with a slash: $A \setminus B$, but sometimes you'll see it notated with a minus: $A - B$. However it is notated, the intention is this: the set we form from $A \setminus B$ is the set we get by taking every element that is in A, and which is *not* in B. We can use that here. Instead of saying "minus," we can use set difference:

$$FV(\lambda\phi.M) = FV(M) \setminus \{\phi\} \tag{11.18}$$

Consider an example. Here is an abstraction:

$$\lambda x.y \tag{11.19}$$

What are the free variables here? First, take the free variables in the body of the expression. The body of the expression is y, so $FV(y)$. Second, subtract from that the binding variable, which is x:

$$FV(\lambda x.y) = FV(y) \setminus \{x\} \tag{11.20}$$

We can reduce this further. The free variables in y is just the set $\{y\}$. So:

$$FV(\lambda x.y) = \{y\} \setminus \{x\} \tag{11.21}$$

We can reduce this even further. The set difference of $\{y\} \setminus \{x\}$ is just $\{y\}$. After all, if we take every element in the first set that is not in the second set, we get $\{y\}$. There is only one element in the first set, namely y, and it is not in the second set.

$$FV(\lambda x.y) = \{y\} \tag{11.22}$$

Here is another example:

$$\lambda x.x \tag{11.23}$$

The set of free variables here will again be the set of free variables in the body of the expression — $FV(x)$ — minus the binding variable — $\{x\}$. So:

$$FV(\lambda x.x) = FV(x) \setminus \{x\} \tag{11.24}$$

The set of free variables in x is just $\{x\}$, so:

$$FV(\lambda x.x) = \{x\} \setminus \{x\} \tag{11.25}$$

What is the set difference of $\{x\} \setminus \{x\}$? What are the elements in the first set that are not in the second set? There are none. The only element in the first set is x, and it is in the second set. So $\{x\} \setminus \{x\}$ is the empty set $\{\}$.

$$FV(\lambda x.x) = \{\} \tag{11.26}$$

Here is one more example:

$$\lambda x.(xy) \tag{11.27}$$

To find the free variables here, we take the free variables in the body of the expression, and we minus the binding variable:

$$FV(\lambda x.(xy)) = FV((xy)) \setminus \{x\} \tag{11.28}$$

To find the free variables of the application (xy), we use the rule from above: find the free variables in the first term, and join those with the free variables in the second term: $FV(x) \cup FV(y)$.

$$FV(\lambda x.(xy)) = (FV(x) \cup FV(y)) \setminus \{x\} \tag{11.29}$$

Of course, the free variables in x is just x, and the free variables in y is just y:

$$FV(\lambda x.(xy)) = (\{x\} \cup \{y\}) \setminus \{x\} \tag{11.30}$$

So if we do the union, we get:

$$FV(\lambda x.(xy)) = \{x, y\} \setminus \{x\} \tag{11.31}$$

Now we can do the set difference. Take every element in the first set that is not in the second set. The element x is in the second set, so we can't take that one. But y is in the first set and it is not in the second set. So we are left with: $\{y\}$.

$$FV(\lambda x.(xy)) = \{y\} \tag{11.32}$$

11.2 Definition

With the above information at hand, let us define the free variables in a lambda expression formally. Here it is:

Definition 4 (The free variables FV in a term M)**.** *Let V be the alphabet of symbols defined above, and let Λ be the set of lambda terms defined above. The free variables FV of any lambda term M (notated as $FV(M)$) is the set of variables that satisfy the following conditions:*

- *If $\phi \in V$, then $FV(\phi) = \{\phi\}$.*
- *If $M, N \in \Lambda$, then $FV(MN) = FV(M) \cup FV(N)$.*
- *If $\phi \in V$ and $M \in \Lambda$, then $FV(\lambda\phi.M) = FV(M) \setminus \{\phi\}$.*

Chapter 12

Bound Variables

We can define bound variables in a similar way. That is, we can consider all three types of lambda terms — variable terms, application terms, and abstraction terms — and identify how we find the bound variables for each case.

12.1 Finding all Bound Variables

Let us notate the bound variables of a term like this: $BV(M)$ is short hand for the bound variables in the term M.

Notation 2. *$BV(M)$ stands for the set of bound variables BV in a λ term M.*

12.1.1 Variable Terms

Let's look at variable terms first. For a variable term, there will be no bound variables. Consider this term:

$$(x) \tag{12.1}$$

What are the bound variables here? There aren't any. There is only a free variable, namely x.

The same goes for any variable ϕ. Whatever variable ϕ might stand for, if you take an expression of this form:

$$(\phi) \tag{12.2}$$

the bound variables for that term will be the empty set $\{\}$:

$$BV(\phi) = \{\} \tag{12.3}$$

12.1.2 Application Terms

Now consider application terms. An application is just two terms next to each other, so the bound variables in an application will be all the bound variables we can find in the first term plus all the bound variables we can find in the second term.

Let M and N be placeholders that can stand for any λ term. Take any application of this form:

$$(MN) \tag{12.4}$$

The bound variables will be the union of the bound variables in M, with the bound variables in N:

$$BV((MN)) = BV(M) \cup BV(N) \tag{12.5}$$

12.1.3 Abstraction Terms

Finally, consider abstraction terms, which have this form (where ϕ is a placeholder for any variable and M is a placeholder for any λ term):

$$\lambda\phi.M \tag{12.6}$$

What are the bound variables here? Well, to begin with, we have the binding variable. So we can start with that: $\{\phi\}$.

But we also have the body of the expression, M. And the set of all bound variables should include any bound variables we find there too. So we also want $BV(M)$.

Hence, the set of bound variables in an abstraction will include the binding variable itself, plus any bound variables we find in the body of the expression:

$$BV(\lambda\phi.M) = \{\phi\} \cup BV(M) \tag{12.7}$$

12.2 Definition

If we put that all together, we can define bound variables formally. Here it is:

Definition 5 (The bound variables BV in a term M)**.** *Let V be the alphabet of symbols defined above, and let Λ be the set of lambda terms defined above. The bound variables BV of any lambda term M (notated as $BV(M)$) is the set of variables that satisfy the following conditions:*

- *If $\phi \in V$, then $BV(\phi) = \{\}$.*
- *If $M, N \in \Lambda$, then $BV(MN) = BV(M) \cup BV(N)$.*
- *If $\phi \in V$ and $M \in \Lambda$, then $BV(\lambda\phi.M) = \{\phi\} \cup BV(M)$.*

Chapter 13

Substitution

A common operation in λ is that you take a term, and you replace every occurrence of a free variable in that term with another term. We call this **substitution**.

13.1 Notation

Let us establish the notation we will use to write about substitution. Let us say that if you have a term M, and you want to replace every free occurrence of a variable ϕ in it with another term N, we will write that like this: $M[\phi := N]$.

Notation 3 (Substitution)**.** *Let M and N be placeholders that stand for any λ terms, and let ϕ be a placeholder that can stand for any symbol from the alphabet V. Then $M[\phi := N]$ denotes the result of replacing every free occurrence of ϕ in M with N.*

13.2 Examples

Here is an example of a substitution. Let M be the term (xy), and let N be the term (z):

$$M = (xy) \tag{13.1}$$
$$N = (z) \tag{13.2}$$

Let us suppose that we want to replace x with N. We would write that like this: $M[x := N]$. If you fill in the values of M and N, you get this: $(xy)[x := (z)]$.

$$M[x := N] = (xy)[x := (z)] \tag{13.3}$$

And then of course, if you proceed to do the actual replacement, i.e., if you replace x with (z), you get this: $((z)y)$.

$$(xy)[x := (z)] = ((z)y) \tag{13.4}$$

We can remove the parentheses that aren't needed to make it a little easier to read:

$$(xy)[x := z] = (zy) \tag{13.5}$$

Here is another example. Let $M = (y)$, and let $N = (xy)$:

$$M = (y) \tag{13.6}$$

$$N = (xy) \tag{13.7}$$

Suppose we want to replace y with N. We would write that like this: $M[y := N]$. If you fill in the values of M and N, you get this: $(y)[y := (xy)]$.

$$M[y := N] = (y)[y := (xy)] \tag{13.8}$$

And if you do the actual replacement, i.e., if you replace (y) with (xy), you get this: (xy).

$$(y)[y := (xy)] = (xy) \tag{13.9}$$

13.3 Strategy for Defining Substitution

To be completely systematic, let us think about all the different cases where we can substitute a new term in this manner.

There are three types of λ terms: variable terms, application terms, and abstraction terms. Let us look at each in turn, starting with variable terms.

Chapter 14

Substitution with Variable Terms

A variable term consists of a single symbol from the alphabet V. Let ϕ be a placeholder that can stand for any symbol in V. So, a variable term has this form:

$$(\phi) \tag{14.1}$$

To replace ϕ in this term with another term, you simply swap this out for the new term. Let N be a placeholder that stands for the new term. After the replacement, you have this:

$$N \tag{14.2}$$

Let us put this in terms of the general formula. The general formula for replacement is this:

$$M[\phi := N] \tag{14.3}$$

This says that we take M, and then we replace every occurrence of ϕ with N. In our case, we can see that, since M is just a single occurrence of ϕ, the result of replacing every ϕ with N is that we replace the only ϕ here. So the result is just N. Let us put that like this:

$$\text{if } M \text{ is the variable term } (\phi)\text{, then } M[\phi := N] = N \tag{14.4}$$

14.1 Examples

Here is an example. Suppose M is the term (x) and N is (y):

$$M = (x) \tag{14.5}$$
$$N = (y) \tag{14.6}$$

Suppose we want to replace every free occurrence of x in M with N:

$$M[x := N] = (x)[x := (y)] \tag{14.7}$$

And the result is just (y):

$$(x)[x := (y)] = (y) \tag{14.8}$$

Here is another example. Suppose M is (z) and N is $((xy)x)$:

$$M = (z) \tag{14.9}$$
$$N = ((xy)x) \tag{14.10}$$

Suppose we want to replace every occurrence of z in M with N:

$$M[z := N] = (z)[z := ((xy)x)] \tag{14.11}$$

After you do the replacement, you get $((xy)x)$:

$$(z)[z := ((xy)x)] = ((xy)x) \tag{14.12}$$

14.2 When There Is Nothing to Replace

Of course, if the variable term you want to replace never occurs in the original term, then the replacement will have no effect.

To see this, let ψ be a placeholder for any symbol from V other than ϕ (so $\psi \neq \phi$). Then M will be (ψ):

$$M = (\psi) \tag{14.13}$$

What is the result now of saying that we want to replace every ϕ with N?

$$M[\phi := N] \tag{14.14}$$

Well, in this case, there is no ϕ that occurs in (ψ). So, the result is that (ψ) goes unchanged:

$$(\psi) \tag{14.15}$$

So, when M is a variable term that doesn't match the value you want to replace, the replacement does nothing. Let us put that like this:

$$\text{if } M \text{ is the variable term } (\psi) \text{ and } \psi \neq \phi, \text{ then } M[\phi := N] = M \tag{14.16}$$

Here is an example. Suppose M is (x) and N is (y):

$$M = (x) \tag{14.17}$$
$$N = (y) \tag{14.18}$$

Suppose now that we want to replace every z in M with N:

$$M[z := N] = (x)[z := (y)] \tag{14.19}$$

No z occurs in (x) here, so we cannot replace anything. The result, then, is that (x) goes unchanged:

$$(x)[z := (y)] = (x) \tag{14.20}$$

14.3 The Two Cases

There is an important point to notice here. For variable terms, if you want to do a substitution, you have two cases:

- If the variable matches what you want to replace, then you replace it.
- If the variable doesn't match what you want to replace, then you don't replace anything.

Chapter 15

Substitution in Application Terms

An application consists of two terms next to each other. Let P and Q be placeholders that can stand for any λ terms. Then an application term has this form:

$$(PQ) \tag{15.1}$$

Suppose now that we want to replace every free occurrence of a variable with another term. To do that, we simply replace every occurrence of the variable in P, and we replace every occurrence of the variable in Q.

We can write this using our notation. Let ϕ be a placeholder that can stand for any symbol from the alphabet V, and let N be a placeholder that can stand for any λ term.

Then, to say that we want to replace every occurrence of ϕ with N, we can say this:

$$(PQ)[\phi := N] \tag{15.2}$$

To do the actual replacement, we simply need to make the replacement in each of P and Q, like this:

$$P[\phi := N] \text{ and } Q[\phi := N] \tag{15.3}$$

Think about the tree of the term:

$$\begin{tikzcd}
 & \mathit{appl.} \arrow[dl] \arrow[dr] & \\
P & & Q
\end{tikzcd}$$

An application breaks into two branches. To replace all the free occurrences of a variable, we need to do it on both branches:

$$\begin{tikzcd}
& \mathit{appl.} \arrow[dl] \arrow[dr] & \\
P[\phi := N] & & Q[\phi := N]
\end{tikzcd}$$

So, we can say that if M is an application of the form (PQ), and we want to replace every ϕ in M with N, we simply replace every ϕ with N in P, and then we also do it in Q:

$$\text{if } M = (PQ), \text{ then } M[\phi := N] = ((P[\phi := N])(Q[\phi := N])) \tag{15.4}$$

15.1 An Example

Here is an example. Suppose M is (xy), and N is (z):

$$M = (xy) \tag{15.5}$$
$$N = (z) \tag{15.6}$$

Now suppose we want to replace every occurrence of x in M with N:

$$M[x := N] = (xy)[x := (z)] \tag{15.7}$$

Here is the tree:

$$\begin{tikzcd}
& \mathit{appl.} \arrow[dl] \arrow[dr] & \\
x & & y
\end{tikzcd}$$

To do the replacement, we need to do the replacement on both branches:

$$\begin{tikzcd}
& \mathit{appl.} \arrow[dl] \arrow[dr] & \\
x[x := (z)] & & y[x := (z)]
\end{tikzcd}$$

Let's do the first branch. Here we have a case of variable substitution. As we saw above, when the term we want to replace matches the variable we want to replace, we can just do the replacement. So the replacement on the first term from the original application ends up being (z):

$$(x)[x := (z)] = (z) \tag{15.8}$$

On the tree:

$$\begin{tikzcd} & appl. \arrow[dl] \arrow[dr] & \\ (z) & & y[x := (z)] \end{tikzcd}$$

Now for the second branch. Here too we have a simple variable substitution. But this time, the replacement does not match, so we do nothing. The result is just y:

$$(y)[x := (z)] = y \tag{15.9}$$

On the tree:

$$\begin{tikzcd} & appl. \arrow[dl] \arrow[dr] & \\ z & & y \end{tikzcd}$$

So the final result is the two together:

$$(zy) \tag{15.10}$$

We started with an application term (xy), and we said we wanted to replace every occurrence of x with (z). So, we replaced every occurrence of x in the first term x, which gave us z, and then we replaced every occurrence of x in the second term y, which gave us y. To put it all in one line:

$$(xy)[x := (z)] = (((x)[x := (z)])((y)[x := (z)])) = (zy) \tag{15.11}$$

15.2 Nested Applications

Here is another example. Suppose M is $(xy)x$ and N is (zz):

$$M = (xy)x \tag{15.12}$$
$$N = (zz) \tag{15.13}$$

Suppose now that we want to replace every occurrence of y in M with N:

$$M[y := N] = ((xy)x)[y := (zz)] \tag{15.14}$$

Here is the tree of the original term:

$$\begin{tikzcd}
 & & \mathit{appl.} \arrow[dl] \arrow[dr] & \\
 & \mathit{appl.} \arrow[dl] \arrow[dr] & & x \\
x & & y &
\end{tikzcd}$$

To do the replacement, we need to do the replacement on each branch. So, we'll do this on the first level of the split:

$$\begin{tikzcd}
 & & \mathit{appl.} \arrow[dl] \arrow[dr] & \\
 & \mathit{appl.}[y := (zz)] \arrow[dl] \arrow[dr] & & x[y := (zz)] \\
x & & y &
\end{tikzcd}$$

But of course, on the left side, we have another application, that's nested. So we want to apply the same rule here: do the replacement on each of *its* branches:

$$\begin{tikzcd}
 & & \mathit{appl.} \arrow[dl] \arrow[dr] & \\
 & \mathit{appl.} \arrow[dl] \arrow[dr] & & x[y := (zz)] \\
x[y := (zz)] & & y[y := (zz)] &
\end{tikzcd}$$

Take the far left branch. There is no y to replace, so stays the same:

$$\begin{tikzcd}
 & & \mathit{appl.} \arrow[dl] \arrow[dr] & \\
 & \mathit{appl.} \arrow[dl] \arrow[dr] & & x[y := (zz)] \\
x & & y[y := (zz)] &
\end{tikzcd}$$

Take the middle branch. This one does have a y to replace, so we substitute in (zz):

$$\begin{tikzcd}
 & & \mathit{appl.} \arrow[dl] \arrow[dr] & \\
 & \mathit{appl.} \arrow[dl] \arrow[dr] & & x[y := (zz)] \\
x & & (zz) &
\end{tikzcd}$$

Now we have completed the substitution on the inner, nested application. We can do the replacement for the final branch, on the right. There is no y to replace, so it stays the same:

$$\begin{tikzcd}
 & & \mathit{appl.} \arrow[dl] \arrow[dr] & \\
 & \mathit{appl.} \arrow[dl] \arrow[dr] & & x \\
x & & (zz) &
\end{tikzcd}$$

And that completes our substitution. To put it all together:

$$((xy)x)[y := (zz)] = \tag{15.15}$$
$$((xy)[y := (zz)])(x[y := (zz)]) = \tag{15.16}$$
$$((x[y := (zz))(y[y := (zz)))x = \tag{15.17}$$
$$(x(zz))x \tag{15.18}$$

Chapter 16

Naive Substitutions in Abstractions

An abstraction is term with a binding expression. Let ϕ be a placeholder that can stand for any symbol in the alphabet V, and let P be a placeholder that can stand for any λ term. Then, an abstraction has this form:

$$\lambda\phi.P \tag{16.1}$$

This has a bound variable, namely ϕ. So ϕ does not occur free in $\lambda\phi.P$. (Think about that point: if there were a ϕ somewhere in P, it would be bound by the binding expression $\lambda\phi$. So there cannot be any ϕs that are free in $\lambda\phi.P$.)

Consequently, we cannot replace any free ϕs in $\lambda\phi.P$, since there aren't any free ϕs in $\lambda\phi.P$ to replace.

But we can replace other free variables in $\lambda\phi.P$. Let ψ be a placeholder that can stand for any symbol in V other than ϕ (so $\phi \neq \psi$) and let N be a placeholder that can stand for any λ term.

Then, to replace every occurrence of ψ with N in $\lambda\phi.P$, you simply need to do the replacement in P:

$$P[\psi := N] \tag{16.2}$$

We can write this with our notation, like this:

$$\text{if } M = \lambda\phi.P \text{ and } \psi \neq \phi,\ M[\psi := N] = \lambda\phi.(P[\psi := N]) \tag{16.3}$$

16.1 An Example

Here is an example. Suppose M is $\lambda x.y$ and N is z:

$$M = \lambda x.y \tag{16.4}$$
$$N = z \tag{16.5}$$

Suppose we want to replace every occurrence of y with N:

$$M[y := N] = (\lambda x.y)[y := z] \tag{16.6}$$

To do that, we do the replacement in the body of the abstraction. Here is a tree of the original expression:

$$\begin{tikzcd} \lambda x. \arrow[d] \\ y \end{tikzcd}$$

The replacement occurs in the body:

$$\begin{tikzcd} \lambda x. \arrow[d] \\ y[y := z] \end{tikzcd}$$

When we do the replacement, y is replaced with z:

$$\begin{tikzcd} \lambda x. \arrow[d] \\ z \end{tikzcd}$$

So the final result is:

$$\lambda x.z \tag{16.7}$$

To put it all in one line:

$$M[y := N] = (\lambda x.y)[y := z] = \lambda x.(y[y := z]) = \lambda x.z \tag{16.8}$$

16.2 Nested Replacements

Here is another example. Let M be $\lambda x.(xy)$, and let N be $\lambda z.z$:

$$M = \lambda x.(xy) \tag{16.9}$$
$$N = \lambda z.z \tag{16.10}$$

Suppose we want to replace every y in M with N:

$$M[y := N] = (\lambda x.(xy))[y := (\lambda z.z)] \tag{16.11}$$

Here is a tree of the original expression:

$$\begin{tikzcd}
& \lambda x. \arrow[d] & \\
& \mathit{appl.} \arrow[dl] \arrow[dr] & \\
x & & y
\end{tikzcd}$$

To replace y with N, we do it on the body of abstraction. The body is an application:

$$\begin{tikzcd}
& \lambda x. \arrow[d] & \\
& \mathit{appl.}[y := (\lambda z.z)] \arrow[dl] \arrow[dr] & \\
x & & y
\end{tikzcd}$$

To perform a substitution on an application, we know from above that we need to do the replacement on each of its child branches:

$$\begin{tikzcd}
& \lambda x. \arrow[d] & \\
& \mathit{appl.} \arrow[dl] \arrow[dr] & \\
x[y := (\lambda z.z)] & & y[y := (\lambda z.z)]
\end{tikzcd}$$

On the left branch, there is no y to replace, but on the right branch there is. So the left branch doesn't change, and the right branch gets the replacement:

$$\begin{tikzcd}
& \lambda x. \arrow[d] & \\
& appl. \arrow[dl] \arrow[dr] & \\
x & & \lambda z.z
\end{tikzcd}$$

The result is this:

$$\lambda x.(x(\lambda z.z)) \tag{16.12}$$

To put it all together:

$$M[y := N] = \tag{16.13}$$
$$(\lambda x.(xy))[y := (\lambda z.z)] = \tag{16.14}$$
$$\lambda x.((xy)[y := (\lambda z.z)]) = \tag{16.15}$$
$$\lambda x.((x[y := (\lambda z.z)])(y[y := (\lambda z.z)])) = \tag{16.16}$$
$$\lambda x.(x(\lambda z.z)) \tag{16.17}$$

16.3 Naive Substitutions

A **variable conflict** occurs when you do a replacement, and the replacement has (free) variables in it that are already used (bound) in the parent term.

In the foregoing examples of abstraction substitutions, we had no variable conflicts. If you look back to each of the examples, you'll see that the original term used xs and ys, while the replacements used zs.

Variable conflicts can cause problems, so they need to be dealt with. Since our examples had no conflicts, we were able to **naively** substitute in the replacements, without worrying about conflicts.

Next, we will turn to the question of how to handle variable conflicts.

Chapter 17

Variable Capture

When you perform a substitution in an abstraction, you take an abstraction M, and you end up grafting a new term N onto various places on the tree.

But remember that the original term M is an abstraction, and so it has a binding variable. That binding variable will bind any occurrences of itself on the tree.

So, if N has any free occurrences of that variable, when you graft N onto the tree, those variables will be **captured** (bound) by the parent's binding expression.

Variable capturing is not allowed in substitutions, but it is important to understand how it happens, and what exactly it results in.

17.1 A Constant Abstraction

Suppose we have a term M, which is $\lambda x.y$:

$$M = \lambda x.y \tag{17.1}$$

Here x is the binding variable, but there is no x in the body of the expression. Instead, the body of the expression is just y.

Think about what this computes. If you apply an argument to this and do the β-reduction, it will *always* reduce to y. For example, suppose we want to replace x with z:

$$\lambda x.y(z) \tag{17.2}$$

What happens when you do the β-reduction? Well, we take the argument, which is z, and we put in the place of every occurrence of x in the body of the abstraction. There are no xs in the body, so we replace nothing. The result is thus:

$$y \tag{17.3}$$

Now suppose we want to replace x with, say, 5. Strictly speaking, this is not a pure lambda term, because numbers are not part of our alphabet.[1] But let us use our imaginations for the moment, just to see the point:

$$\lambda x.y(5) \tag{17.4}$$

If we do the β-reduction, we get the same result. We take the argument, which is 5, and we put it in the place of every occurrence of x in the body of the abstraction. There are no xs in the body, so we replace nothing. The result is thus:

$$y \tag{17.5}$$

You can see that no matter what we provide as an argument for x, when we try to reduce this expression, it will always reduce to y.

We can see even more clearly this by looking at the tree of this expression:

$$\begin{tikzcd}\lambda x \arrow[d] \\ y\end{tikzcd}$$

Let us think about the binding and bound variables as slots instead of symbols. For instance, replace x with square brackets:

$$\begin{tikzcd}\lambda[\ \ ] \arrow[d] \\ y\end{tikzcd}$$

We can see that whatever we put into the slot, there is no slot in the body to fill in. The body will always be y. So again, this abstraction will always reduce to y.

Sometimes we use a different way of speaking to make this same point: we say this abstraction always **returns** y. Also, because it always returns (or reduces down to) the same value, we say it returns a **constant**.

17.2 A Naive Substitution

Consider the constant abstraction we just discussed:

[1]Of course, we can build up a number system using just the alphabet we have stipulated here, but that is not something we will do at this point. That happens when you start using the λ calculus to build up more complex data structures like numbers, lists, and so on. You can do all of these things with the λ calculus, but we will not do that at this point.

$$M = \lambda x.y \tag{17.6}$$

Here is a tree of it:

$$\begin{tikzcd}
\lambda x \arrow[d] \\
y
\end{tikzcd}$$

Now suppose that we have another term N, which is just x:

$$N = x \tag{17.7}$$

Suppose that we want to take M from before, and replace every occurrence of y with N:

$$M[y := N] \tag{17.8}$$

Substitute M and N in, and you get this:

$$(\lambda x.y)[y := x] \tag{17.9}$$

To do this replacement in a naive way, we do the replacement in the body of the expression:

$$\begin{tikzcd}
\lambda x \arrow[d] \\
y[y := x]
\end{tikzcd}$$

And if you do the replacement, you get this:

$$\begin{tikzcd}
\lambda x \arrow[d] \\
x
\end{tikzcd}$$

So the final result is:

$$\lambda x.x \tag{17.10}$$

To put it on one line:

$$(\lambda x.y)[y := x] = \lambda x.x \tag{17.11}$$

17.3 The New Abstraction

The new abstraction we ended up with is this:

$$\lambda x.x \tag{17.12}$$

Think about what this term computes. The binding variable is x, but the body is x too. So, if you provide an argument and do the β-reduction, you will replace the body with whatever value the argument is.

For example, suppose we want to replace x with z:

$$\lambda x.x(z) \tag{17.13}$$

If we then do the β-reduction, we take z and put in the place of the bound x, which gives us:

$$z \tag{17.14}$$

Let us imagine that we can replace x with 5 (again, this is not a pure λ term, since numbers are not in our alphabet, but will use our imaginations).

$$\lambda x.x(5) \tag{17.15}$$

If we then do the β-reduction and replace x in the body of the expression with 5, we get:

$$5 \tag{17.16}$$

No matter what value we provide as an argument, when we do the β-reduction, we will always return that same argument. This abstraction will always return the argument you give it.

We can see this even more clearly by looking at the tree:

$$\begin{tikzcd}\lambda x \arrow[d] \\ x\end{tikzcd}$$

If we think about slots instead of symbols, we can see the structure of this abstraction:

$$\begin{tikzcd}\lambda[\ \ ] \arrow[d] \\ [\ \ ]\end{tikzcd}$$

You can see from this that whatever value you put in the binding slot, you put that same value into the body of the expression.

So, this abstraction is **not constant**. The previous abstraction always returned y, but this one always returns whatever argument you give it.

17.4 Variable Capture

The new abstraction, $\lambda x.x$, has very different behavior from the previous one, $\lambda x.y$. As we saw, the previous abstraction is constant, but the new one is not.

So this replacement changed the behavior of the original in a quite substantial way.

The problem is that the value we swapped in got captured by the binding expression. In the first form of the expression, $\lambda x.y$, the y was free. Again, look at the tree:

$$\begin{tikzcd}\lambda x \arrow[d] \\ y\end{tikzcd}$$

The body is y, which is not bound by the binding variable x.

But then, we replaced y with x:

$$\begin{tikzcd}\lambda x \arrow[d] \\ x\end{tikzcd}$$

And at that point, the new value we put into the body of the expression got **captured** by the parent binding variable.

17.5 Capture at Depth

Capturing can happen at any level. A replacement might have any number of nested terms in it, and a variable that lives deep in a nested child might be the one that gets captured.

Consider again our initial expression:

$$M = \lambda x.y \tag{17.17}$$

Here is the tree:

$$\begin{tikzcd}\lambda x \arrow[d] \\ y\end{tikzcd}$$

Suppose that N is a complex term, for instance:

$$N = (y(z(ux))) \tag{17.18}$$

Here is a tree of N:

$$\begin{tikzcd}
 & \mathit{appl.} \arrow[dl] \arrow[dr] & & & \\
y & & \text{appl.} \arrow[dl] \arrow[dr] & & \\
 & z & & \text{appl.} \arrow[dl] \arrow[dr] & \\
 & & u & & x
\end{tikzcd}$$

Suppose we want to replace y in the original expression with N:

$$M[y := N] = (\lambda x.y)[y := (y(z(ux)))] \tag{17.19}$$

The replacement happens in the body of the original expression, on the y:

$$\begin{tikzcd}
\lambda x \arrow[d] \\
y[y := N]
\end{tikzcd}$$

So, if we replace y with N, we graft the tree of N on at that point:

$$\begin{tikzcd}
 & \lambda x \arrow[d] & & & \\
 & \mathit{appl.} \arrow[dl] \arrow[dr] & & & \\
y & & \text{appl.} \arrow[dl] \arrow[dr] & & \\
 & z & & \text{appl.} \arrow[dl] \arrow[dr] & \\
 & & u & & x
\end{tikzcd}$$

But notice that, at the far right child branch, we have an x. It was free before (when N was all by itself), but now that it is grafted onto the tree for M, it ends up getting captured by the binding variable x way at the top of the tree.

Chapter 18

Renaming

In a substitution, you take a term M, and you replace every free occurrence of a variable ϕ in it with another term N. We write that like this:

$$M[\phi := N] \tag{18.1}$$

However, we cannot naively perform the substitution if any of the free variables in N will get captured after the replacement. Before we do the replacement, we must change some variable names to avoid the conflict.

We can do this by replacing all the bound variables in the original term M with fresh symbols that do not occur in N. After that, we can proceed with our naive substitition.

18.1 Fresh Symbols

When you rename the bound variables, you must pick fresh symbols that do not occur in the term. We can define a fresh symbol as any symbol from our alphabet that is not among the bound variables of the term, nor among the free variables of the term.

Definition 6 (Fresh symbols)**.** *Let M be a placeholder that can stand for any λ term, and let ϕ be a placeholder that can stand for any symbol from the alphabet V. Then ϕ is fresh in M (notated as $Fr(M)$) just in case the following conditions are satisfied:*

- $\phi \notin BV(M)$
- $\phi \notin FV(M)$

For example, suppose we have this term:

$$\lambda x.(xy) \tag{18.2}$$

Here are the variables we find in it:

- Bound variables: $\{x\}$
- Free variables: $\{y\}$

A fresh symbol will be any symbol from V that is not in these two sets — so any symbol from V that is neither x nor y. Here are some fresh symbols:

$$a, b, c, u, v, z, \ldots \tag{18.3}$$

Remark 1 (The Barendregt Convention)**.** *Henk Barendregt wrote an important book in the early 1980s called The Lambda Calculus: Its Syntax and Semantics. There Barendregt suggested that all distinct bound variables in a λ term should be fresh. That way, there are no variable conflicts. This is now often referred to as the* ***Barendregt convention****.*

18.2 Renaming Procedure

An abstraction has this form:

$$\lambda\phi.P \tag{18.4}$$

To rename the bound variable ϕ (and all occurrences of it in P), you proceed in three steps:

1. Select a symbol that is fresh in P, call it ψ.
2. In P, replace all free occurrences of ϕ with ψ.
3. Replace $\lambda\phi$ with $\lambda\psi$.

Think about each of these steps.

In step (1), we pick a symbol that is fresh in P. That is, we pick a symbol from $Fr(P)$ — a symbol that does not occur in P.

In step (2), we do a substitution of the sort we have already discussed. To use our notation, we do this:

$$P[\phi := \psi] \tag{18.5}$$

In step (3), we strip $\lambda\phi$ from the front of P, and we put $\lambda\psi$ in its place.

To put this all together, we start with this:

$$\lambda\phi.P \tag{18.6}$$

And we end up with this:

$$\lambda\psi.(P[\phi := \psi]),\ \text{where } \phi \in Fr(P) \tag{18.7}$$

18.3 α-conversion

When we follow the procedure we just outlined, and rename the bound variables in an abstraction term, we call that an **α-conversion** (or just "alpha-conversion" if you do not want to write the Greek letter α).

We can also say that an α-converted term is **α-congruent** with the original term, or that it is an **α-variant** of the original term.

Definition 7 (α-conversion)**.** *Let M be an abstraction of the form $\lambda\phi.P$, and let N be a placeholder for any λ term. Then:*

- *N is an α-conversion of M just in case $\lambda\psi.(P[\phi := \psi])$, where $\psi \in Fr(P)$.*
- *If N is an α-conversion of M, then N and M are α-congruent.*
- *If N is an α-conversion of M, then N is an α-variant of M.*

18.4 A Simple Example

Suppose we have an abstraction M, like this:

$$M = \lambda x.y \tag{18.8}$$

Here is the tree of M:

$$\begin{tikzcd}\lambda x \arrow[d] \\ y\end{tikzcd}$$

Let us rename the bound variables in M (i.e., let us perform an α-conversion on M).

If we follow our procedure, we first select a fresh symbol. The bound and free variables in M are these:

$$BV(M) = \{x\}, FV(M) = \{y\} \tag{18.9}$$

So a fresh symbol $Fr(P)$ will be any symbol that is not x or y. Let us pick z:

$$z \in Fr(P) \tag{18.10}$$

The second step is to replace every occurrence of x with z in the body of M. The body is y, so:

$$y[x := z] \tag{18.11}$$

Here it is on the tree:

$$\begin{tikzcd}\lambda x \arrow[d] \\ y[x := z]\end{tikzcd}$$

Since there are no free occurrences of x in y, y remains unchanged:

$$\begin{tikzcd}\lambda x \arrow[d] \\ y\end{tikzcd}$$

Finally, we remove λx and replace it with λz:

$$\begin{tikzcd}\lambda z \arrow[d] \\ y\end{tikzcd}$$

So, the final result is:

$$\lambda z.y \tag{18.12}$$

18.5 A Nested Example

Note that step (2) involves a substitution, which we perform just as we discussed earlier. And of course, substitutions can involve nested substitutions. In such cases, you need to make sure to carry the substitution through to the nested children terms as well.

For example, suppose we have an abstraction M, like this:

$$M = \lambda x.(xy) \tag{18.13}$$

Here is the tree of M:

$$\begin{tikzcd} & \lambda x \arrow[d] & \\ & \text{appl.} \arrow[dl] \arrow[dr] & \\ x & & y\end{tikzcd}$$

Let us rename the bound variable x. First, we pick a fresh variable. Let us pick z again:

$$z \in Fr(M) \tag{18.14}$$

Now let us replace every free occurrence of x with z in the body of M. That is:

$$\lambda x.((xy)[x := z]) \tag{18.15}$$

Here it is on the tree:

$$\begin{tikzcd}
& \lambda x \arrow[d] & \\
& \text{appl.}[x := z] \arrow[dl] \arrow[dr] & \\
x & & y
\end{tikzcd}$$

To do a substitution on an application, we do the replacement on each of the child branches:

$$\begin{tikzcd}
& \lambda x \arrow[d] & \\
& \text{appl.} \arrow[dl] \arrow[dr] & \\
x[x := z] & & y[x := z]
\end{tikzcd}$$

On the left branch, x gets replaced with z. On the right branch, there are no xs, so nothing happens:

$$\begin{tikzcd}
& \lambda x \arrow[d] & \\
& \text{appl.} \arrow[dl] \arrow[dr] & \\
z & & y
\end{tikzcd}$$

Finally, we remove λx and replace it with λz:

$$\begin{tikzcd}
& \lambda z \arrow[d] & \\
& \text{appl.} \arrow[dl] \arrow[dr] & \\
z & & y
\end{tikzcd}$$

So, the final result is:

$$\lambda z.(zy) \tag{18.16}$$

Chapter 19

Multiple Renames

An abstraction can have children abstractions nested in its subterms. These can be renamed too, simply be repeating the renaming procedure for each abstraction.

To do multiple renames, always start from the innermost abstraction, and do the rename there first. Then work your way outwards and rename each successive outer abstraction as you come to them.

Suppose we have an abstraction M, that looks like this:

$$M = \lambda x.(x(\lambda y.(xz))) \tag{19.1}$$

Here is the tree of M:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda y \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& x & & z
\end{tikzcd}$$

The bound and free variables in this tree are these:

$$BV(M) = \{x, y\}, FV(M) = \{x, z\} \tag{19.2}$$

19.1 The First Pass

To begin, let us rename the bound variable y. First, we select a fresh symbol. We cannot select z, since z is among the free variables of M. Let us select v_1:

$$v_1 \in Fr(M) \tag{19.3}$$

Next, let us substitute v_1 for every free occurrence of y in the body of the abstraction:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda y \arrow[d] & \\
& & \text{appl.}[y := v_1] \arrow[dl] \arrow[dr] & \\
& x & & z
\end{tikzcd}$$

To perform a substitution on an application, we push the substitution down to each branch:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda y \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& x[y := v_1] & & z[y := v_1]
\end{tikzcd}$$

There are no free ys on either branch, so the bottom branches remain unchanged:

$$\begin{tikzcd}
 & \lambda x \arrow[d] & & \\
 & \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda y \arrow[d] & \\
 & & \text{appl.} \arrow[dl] \arrow[dr] & \\
 & x & & z
\end{tikzcd}$$

The last step in the rename is to replace λy with λv_1:

$$\begin{tikzcd}
 & \lambda x \arrow[d] & & \\
 & \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda v_1 \arrow[d] & \\
 & & \text{appl.} \arrow[dl] \arrow[dr] & \\
 & x & & z
\end{tikzcd}$$

The result is:

$$\lambda x.(x(\lambda v_1.(xz))) \tag{19.4}$$

Let us call this new term N:

$$N = \lambda x.(x(\lambda v_1.(xz))) \tag{19.5}$$

19.2 The Second Pass

At this point, we can turn to the outer abstraction. First, we pick a fresh symbol. Let us select v_2:

$$v_2 \in Fr(N) \tag{19.6}$$

Now we want to replace every free occurrence of x with v_2 in the body of N:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.}[x := v_2] \arrow[dl] \arrow[dr] & & \\
x & & \lambda v_1 \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& x & & z
\end{tikzcd}$$

To do a substitution on an application, we push the replacement to each branch:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
x[x := v_2] & & \lambda v_1[x := v_2] \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& x & & z
\end{tikzcd}$$

On the left branch, we can replace x with v_2:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
v_2 & & \lambda v_1[x := v_2] \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& x & & z
\end{tikzcd}$$

On the right branch, we have an abstraction, so we push the substitution to the body:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
v_2 & & \lambda v_1 \arrow[d] & \\
& & \text{appl.}[x := v_2] \arrow[dl] \arrow[dr] & \\
& x & & z
\end{tikzcd}$$

Now we have an application, so we push the replacement to its branches:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
v_2 & & \lambda v_1 \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& x[x := v_2] & & z[x := v_2]
\end{tikzcd}$$

On the left, we can replace x with v_2, and on the right, there is nothing to replace, so it remains unchanged:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
v_2 & & \lambda v_1 \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& v_2 & & z
\end{tikzcd}$$

The last step is we replace λx with λv_2:

$$\begin{tikzcd}
 & \lambda v_2 \arrow[d] & & \\
 & \text{appl.} \arrow[dl] \arrow[dr] & & \\
v_2 & & \lambda v_1 \arrow[d] & \\
 & & \text{appl.} \arrow[dl] \arrow[dr] & \\
 & v_2 & & z
\end{tikzcd}$$

The final result is:

$$\lambda v_2.(v_2(\lambda v_1.(v_2 z))) \tag{19.7}$$

Now we have completely renamed all the bound variables in the original term M, so that they are fresh. Neither of the bound variables x or y from before are present in this term.

Chapter 20

Non-Naive Substitution in Abstractions

Now that we have defined α-conversions, we can reformulate how substitution works in abstractions, in a non-naive way. As we mentioned before: we first rename all bound variables in the term, and then we do the naive substitution.

Let us formulate this with our notation. An abstraction term M has this form:

$$M = \lambda\phi.P \tag{20.1}$$

Before, we defined naive substitution as follows. Let N be another term, and let ψ be any symbol other than ϕ (so $\psi \neq \phi$). What we want to do is replace every occurrence of ψ with N in M, which we write like this:

$$M[\psi := N] \tag{20.2}$$

To do that naively, we simply do the replacement in the body of the expression, namely in P:

$$M[\psi := N] = \lambda\phi.(P[\psi := N]) \tag{20.3}$$

However, we have now seen that we must first rename the bound variables in P, so that they are fresh. The bound variable here is ϕ, so that is the one we want to replace.

If we follow the procedure we outlined above, we first select a fresh symbol. Let ξ be a placeholder that can stand for any symbol in V that is fresh (so $\xi \neq \psi$ and $\xi \neq \phi$):

$$\xi \in FV(M) \tag{20.4}$$

Second, we replace every free occurrence of ϕ in P with ξ:

$$P[\phi := \xi] \tag{20.5}$$

And third, we remove $\lambda\phi$ from the front and replace it with $\lambda\xi$:

$$\lambda\xi.(P[\phi := \xi]) \tag{20.6}$$

The result of that will be an α-variant of the original term M.

Now we can do the naive substitution, where we replace every ψ with N. To write that, we simply tack $[\psi := N]$ onto the end:

$$(\lambda\phi.P)[\psi := N] = \lambda\xi.((P[\phi := \xi])[\psi := N]) \tag{20.7}$$

The parentheses make it clear which replacements must happen first. First, we rename the bound ϕs to ξs, and then we replace the ψs with Ns.

Here is the whole rule:

if $M = \lambda\phi.P$, $\psi \neq \phi$, and $\xi \in Fr(M)$, then:

$$\begin{aligned} M[\psi := N] = (\lambda\phi.P)[\psi := N] = \\ \lambda\xi.((P[\phi := \xi])[\psi := N]) \end{aligned} \tag{20.8}$$

The notation here is cumbersome, but the basic point is simple: give the bound variables fresh names first, then do the naive substitution.

20.1 An Example

Suppose M is $\lambda x.y$ and N is x:

$$M = \lambda x.y \tag{20.9}$$
$$N = x \tag{20.10}$$

Suppose we want to replace every free occurrence of y with N:

$$M[y := N] \tag{20.11}$$

Swap in M and N, and you get:

$$(\lambda x.y)[y := x] \tag{20.12}$$

Here is the tree of M:

$$\begin{tikzcd} \lambda x \arrow[d] \\ y \end{tikzcd}$$

If we were to do this substitution naively, we would replace y with x, and the new x would get captured by the binding expression λx:

$$\lambda x.x \tag{20.13}$$

So before we do the substitution, we must rename the bound variable x. Let us pick a fresh symbol, v_1:

$$v_1 \in Fr(M) \tag{20.14}$$

Now we want to replace every x in the body with v_1:

$$\begin{tikzcd} \lambda x \arrow[d] \\ y[x := v_1] \end{tikzcd}$$

There is nothing to replace here, so the body of the abstraction remains unchanged.

$$\begin{tikzcd} \lambda x \arrow[d] \\ y \end{tikzcd}$$

The last step of the rename procedure is to remove λx and replace it with λv_1:

$$\begin{tikzcd} \lambda v_1 \arrow[d] \\ y \end{tikzcd}$$

So, after the rename, the abstraction is this:

$$\lambda v_1.y \tag{20.15}$$

Now we can do our original substitution naively. We want to replace every y with x:

$$(\lambda v_1.y)[y := x] \tag{20.16}$$

We do the substitution on the body the abstraction:

$$\begin{tikzcd} \lambda v_1 \arrow[d] \\ y[y := x] \end{tikzcd}$$

And once we do the replacement, we get:

$$\begin{tikzcd}\lambda v_1 \arrow[d] \\ x\end{tikzcd}$$

The final result is:

$$\lambda v_1.x \tag{20.17}$$

Now we have replaced y in the body of the abstraction with x, but because we renamed the bound x to v_1, there is no conflict. The x we put into the body does not get captured.

Chapter 21

Multiple Non-Naive Substitutions

Terms can be nested in other terms. So when we perform substitutions, we need to be careful to carry the substitutions out correctly, all the way down the branches, as many times as needed.

Suppose M is $\lambda x.(xy)$ and N is $\lambda x.z$:

$$M = \lambda x.(x(\lambda x.(xy))) \tag{21.1}$$
$$N = \lambda x.z \tag{21.2}$$

Suppose we want to replace every free occurrence of y in M with N:

$$M[y := N] \tag{21.3}$$

Swap in M and N, and you get:

$$(\lambda x.(x(\lambda x.(xy))))[y := (\lambda x.z)] \tag{21.4}$$

Here is the tree of M:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda x \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& & x & y
\end{tikzcd}$$

Notice that there is a nested abstraction here, and both the child and parent abstractions bind x. But each has a different scope. The lowest x is bound by the lower λx, and the higher x is bound by the higher λx.

We can see this clearly if we think of slots in terms of variables. For the innermost x, I'll use square brackets [ ], and for the outermost x, I'll use round braces ( ). The tree looks like this:

$$\begin{tikzcd}
& \lambda(\;\;) \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
(\;\;) & & \lambda[\;\;] \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& & [\;\;] & y
\end{tikzcd}$$

This makes it clear which slots are bound where. The bottom slot [ ] is bound by the bottom λ[ ], and the top left slot ( ) is bound by the top λ( ).

Before we do our substitution, we need to rename the bound variables. We always start renaming from the innermost abstraction, and we work our way outwards. So let us rename the bottom abstraction first. Let us pick a fresh symbol, v_1:

$$v_1 \in Fr(M) \tag{21.5}$$

Then we replace every free x in the body of the abstraction with v_1:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda x \arrow[d] & \\
& & \text{appl.}[x := v_1] \arrow[dl] \arrow[dr] & \\
& x & & y
\end{tikzcd}$$

This is an application, so we push the substitution to its branches:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda x \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& x[x := v_1] & & y[x := v_1]
\end{tikzcd}$$

On the left, we can replace x, and on the right, there is no change:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda x \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& v_1 & & y
\end{tikzcd}$$

The last step of the rename is to replace the bottom λx with λv_1:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
x & & \lambda v_1 \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& v_1 & & y
\end{tikzcd}$$

Now we do a rename for the outer abstraction. Let us select a fresh variable v_2, and let us replace every free x in the body of the abstraction with v_2:

$$\begin{tikzcd}
 & \lambda x \arrow[d] & & \\
 & \text{appl.}[x := v_2] \arrow[dl] \arrow[dr] & & \\
x & & \lambda v_1 \arrow[d] & \\
 & & \text{appl.} \arrow[dl] \arrow[dr] & \\
 & v_1 & & y
\end{tikzcd}$$

We can fast-forward a few steps, and push the replacement all the way down to all the branches:

$$\begin{tikzcd}
 & \lambda x \arrow[d] & & \\
 & \text{appl.} \arrow[dl] \arrow[dr] & & \\
x[x := v_2] & & \lambda v_1 \arrow[d] & \\
 & & \text{appl.} \arrow[dl] \arrow[dr] & \\
 & v_1[x := v_2] & & y[x := v_2]
\end{tikzcd}$$

On the upper left branch, we can replace x, and on the bottom two branches there is no change:

$$\begin{tikzcd}
& \lambda x \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
v_2 & & \lambda v_1 \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& v_1 & & y
\end{tikzcd}$$

The last step of the rename is to remove λx and put λv_2 in its place:

$$\begin{tikzcd}
& \lambda v_2 \arrow[d] & & \\
& \text{appl.} \arrow[dl] \arrow[dr] & & \\
v_2 & & \lambda v_1 \arrow[d] & \\
& & \text{appl.} \arrow[dl] \arrow[dr] & \\
& v_1 & & y
\end{tikzcd}$$

So the final term after all bound variables have been renamed is this:

$$\lambda v_2.(v_2(\lambda v_1.(v_1 y))) \tag{21.6}$$

Notice how the rename has made the expression a little easier to understand. There can be no confusion here about which x is bound by which λx. Here we have different (fresh) symbols for each binding, and so the tree looks a lot more like the tree we drew above with different slots.

At this point, we can do our naive substitution. Recall that N is $\lambda x.z$, and that we want to replace every free y with N:

$$\begin{tikzcd}
& \lambda v_2[y := (\lambda x.z)] \arrow[d] & & \\
& \text{appl.} \arrow[ld] \arrow[rd] & & \\
v_2 & & \lambda v_1 \arrow[d] & \\
& & \text{appl.} \arrow[ld] \arrow[rd] & \\
& v_1 & & y
\end{tikzcd}$$

We have no variable conflicts (since we replaced all bound variables with fresh symbols), so we can proceed naively. We can fast-forward a few steps and push the substitution to all the branches:

$$\begin{tikzcd}
& \lambda v_2 \arrow[d] & & \\
& \text{appl.} \arrow[ld] \arrow[rd] & & \\
v_2[y := (\lambda x.z)] & & \lambda v_1 \arrow[d] & \\
& & \text{appl.} \arrow[ld] \arrow[rd] & \\
& v_1[y := (\lambda x.z)] & & y[y := (\lambda x.z)]
\end{tikzcd}$$

On the bottom right branch, we can remove y and graft on N, but the rest of the replacements result in no change:

$$\begin{tikzcd}
 & \lambda v_2 \arrow[d] & & \\
 & \text{appl.} \arrow[dl] \arrow[dr] & & \\
v_2 & & \lambda v_1 \arrow[d] & \\
 & & \text{appl.} \arrow[dl] \arrow[dr] & \\
 & v_1 & & \lambda x \arrow[d] \\
 & & & z
\end{tikzcd}$$

Chapter 22

Substitution (Formally)

With the above information at hand, we can define substitution formally. The format for any substitution is:

$$M[\phi := N] \tag{22.1}$$

where M and N are placeholders that can stand for any λ term, and ϕ is a placeholder that can stand for any symbol in the alphabet V.

There are three types of λ terms: variable terms, application terms, and abstraction terms. Substitution works differently for each:

- **Variables** — If the variable is just ϕ, you replace it with N, otherwise you make no change. So if $M = \phi$, you replace that with N. If M is some other variable, ψ, where $\psi \neq \phi$, then you leave it as ψ.
- **Applications** — You push the substitution into each of the child terms. If M is (PQ), you do the replacement on each: $(P[\phi := N])(Q[\phi := N])$.
- **Abstractions** — This is the complicated one. M has the form $\lambda\psi.P$, and you proceed in two stages. First, you replace all the bound variables with fresh symbols. So, you replace ψ with a fresh symbol, ξ (i.e., $\xi \in Fr(M)$). That is to say, $\lambda\psi.P$ gets converted to $\lambda\xi.(P[\psi := \xi])$. And of course, P itself might have further nested abstractions, and you need to walk down the tree and re-apply this rule: replace all bound variables with fresh symbols. Then you can move to the second stage: you do a naive substitution and replace ϕ in the body of the expression with N — i.e., $\lambda\xi.(P[\psi := \xi]([\phi := N]))$.

Here is the full definition:

Definition 8 (Substitution)**.** *Let M, N, P, and Q be placeholders that can stand for any λ term, and let ϕ, ψ, and ξ be placeholders that can stand for any symbol in the alphabet V. Then substitution is defined as the result of taking a term M, and replacing every free occurrence of ϕ in it with another term N.*

This is notated as $M[\phi := N]$*. The replacement must proceed according to the following rules:*

- *If* M *is* (ϕ)*, then* $M[\phi := N] = N$*.*
- *If* M *is* (ψ) *and* $\psi \neq \phi$*, then* $M[\phi := N] = M$*.*
- *If* M *is* (PQ)*, then* $M[\phi := N] = ((P[\phi := N])(Q[\phi := N]))$*.*
- *If* M *is* $\lambda\psi.P$*, then* $M[\phi := N] = \lambda\xi.(P[\psi := \xi]([\phi := N]))$*, where* $\xi \in Fr(M)$*.*

Chapter 23

Equivalence

23.1 Reflexivity, Symmetry, and Transitivity

In mathematics and logic, we can compare two objects to each other in various ways. There are three properties of relationships that we often look at:

- Some comparisons are **symmetric**, which means it goes both ways. If John is the brother of Thomas, it goes the other way too: Thomas is the brother is John. Not all comparisons are symmetric. If John is taller than Thomas, it does not follow that Thomas is taller than John.
- Some comparisons are **reflexive**, which means the objects can be compared to themselves too. If John is as smart as Thomas, we can say that John is as smart as himself too. Of course, we may not normally say things like "John is as smart as himself," but it is a true statement (in fact, it seems like it couldn't be false). Not all comparisons are reflexive. If Harold is a parent of John, it does not follow that Harold is a parent of Harold.
- Some comparisons are **transitive**, which means they can be chained. If John is taller than Thomas and Thomas is taller than Sally, then John is taller than Sally. Not all comparisons are transitive. If my left arm is attached directly to my body and my body is attached directly to my right arm, it does not follow that my left arm is attached directly to my right arm (on the contrary, my body is in between them).

We can put these statements more formally, by describing them as general properties of relations.

Definition 9 (Reflexivity, Symmetry, Transitivity)**.** *Let R be a relation, and let x, y, and z be placeholders that can stand for any items related by R. Then:*

- *R is reflexive: if x is related to y by R, then x is related to x by R.*

- *R is symmetric: if x is related to y by R, then y is related to x by R.*
- *R is transitive: if x is related to y by R and y is related to z by R, then x is related to z by R.*

23.2 Definition of Equivalence

When objects are compared in such a way that the comparison is all of the above — that is, it is reflexive, symmetric, and transitive — then we say the objects are **equivalent**. The definition of equivalence is precisely any relationship that is reflexive, symmetric, and transitive.

Definition 10 (Equivalence)**.** *For any relation R, R is an equivalence relation just in case R is reflexive, symmetric, and transitive.*

23.3 Equivalence and Identity

Equivalence is not the same as identity. To say that two objects are equivalent is not the same as saying that they are identical.

- The morning star and the evening star are the very same object, namely the planet Venus. So they are identical.
- To prove that you can work in the United States, you need to show one of two things: (i) a passport, or (ii) a birth certificate and a state-issued driver's license. These two options are not identical, but they are equivalent, for the purposes of proving that you are allowed to work in the United States.

Mathematicians and logicians sometimes use an equals sign (=) ambiguously. Sometimes they use it to mean identity. For instance, in arithmetic, we might see this:

$$7 = 7 \tag{23.1}$$

This is identity. The number 7 is the same object as the number 7, or the value 7 is identical to the value 7.

But the following is equivalence, not identity:

$$5 + 2 = 7 \tag{23.2}$$

These are computationally equivalent, because if you solve on the left side of the equation you get an identical value. But $5 + 2$ is obviously not identical to 7.

To help keep things clear, we will reserve an equals sign (=) for identity, and we will use a three-line equals sign ($\equiv$) for equivalence. So:

$$5 + 2 \equiv 7 \tag{23.3}$$

And:

$$7 = 7 \tag{23.4}$$

Chapter 24

Equivalent α-Variants (Informally)

In the λ calculus, α-variants are equivalent. We can see this informally by looking at how they compute the same results, and how they have the same structure.

24.1 Computationally Equivalent

Any α-variants will compute the same result. For instance, consider this term:

$$\lambda x.x \tag{24.1}$$

Computationally, this will return whatever argument we provide to it. For instance, suppose we apply it to y:

$$(\lambda x.x)y \tag{24.2}$$

If you do the β-reduction, you get y:

$$(\lambda x.x)y = y \tag{24.3}$$

If you apply it to a different argument, say z, you get z:

$$(\lambda x.x)z = z \tag{24.4}$$

Now suppose we replace the bound variables with fresh symbols. For instance, let's replace x with a:

$$\lambda a.a \tag{24.5}$$

This new term behaves the same way. It always returns the argument. We can do the same two computations as above, and the result is the same:

$$(\lambda a.a)y = y \tag{24.6}$$
$$(\lambda a.a)z = z \tag{24.7}$$

Let's rename it again. Let's pick v_{10}:

$$(\lambda v_{10}.v_{10}) \tag{24.8}$$

Again, this computes the same results:

$$(\lambda v_{10}.v_{10})y = y \tag{24.9}$$
$$(\lambda v_{10}.v_{10})z = z \tag{24.10}$$

Here is another example. Take this term:

$$\lambda x.(xy) \tag{24.11}$$

Let's apply this to a, and then to b:

$$(\lambda x.(xy))a = (ay) \tag{24.12}$$
$$(\lambda x.(xy))b = (by) \tag{24.13}$$

When we compute the results here, we can see that this abstraction always returns the argument with y.

Now rename the bound variables to, say, v_1. We get the same results:

$$(\lambda v_1.(v_1y))a = (ay) \tag{24.14}$$
$$(\lambda v_1.(v_1y))b = (by) \tag{24.15}$$

Try more renamings. We still get the same results:

$$(\lambda v_2.(v_2y))a = (ay) \tag{24.16}$$
$$(\lambda v_2.(v_2y))b = (by) \tag{24.17}$$
$$\ldots \tag{24.18}$$
$$(\lambda v_4.(v_4y))a = (ay) \tag{24.19}$$
$$(\lambda v_4.(v_4y))b = (by) \tag{24.20}$$
$$\ldots \tag{24.21}$$

24.2 Same Structure

If we think of the bound variables as slots instead of symbols, it is easy to see that α-variants have the same structure. Take the abstraction we had above:

$$\lambda x.(xy) \tag{24.22}$$

Here it is as a tree:

$$\begin{tikzcd}
& \lambda x \arrow[d] & \\
& \text{appl.} \arrow[dl] \arrow[dr] & \\
x & & y
\end{tikzcd}$$

If we think of the xs as slots, we can see the structure of this expression very clearly:

$$\begin{tikzcd}
& \lambda[\ \ ] \arrow[d] & \\
& \text{appl.} \arrow[dl] \arrow[dr] & \\
[\ \ ] & & y
\end{tikzcd}$$

No matter which fresh symbols we put in the place of the bound xs, the result will have this same structure.

This is also true when there are multiple, nested bindings. Consider the following λ term:

$$\lambda x.(\lambda y.(xy)) \tag{24.23}$$

Here it is as a tree:

$$\begin{tikzcd}
& \lambda x \arrow[d] & \\
& \lambda y \arrow[d] & \\
& \text{appl.} \arrow[dl] \arrow[dr] & \\
x & & y
\end{tikzcd}$$

If we use square brackets for the x slots and round braces for the y slots, we can see the structure of this term:

$$\begin{tikzcd}
& \lambda[\ \ ] \arrow[d] & \\
& \lambda(\ \ ) \arrow[d] & \\
& \text{appl.} \arrow[dl] \arrow[dr] & \\
{[\ \ ]} & & (\ \ )
\end{tikzcd}$$

Here too, we can see that no matter which fresh symbols we put in the appropriate slots, the structure remains the same.

Chapter 25

Specifying Types

Type systems provide a way of stipulating a variety of things:

- You can stipulate what types of **values** are available for use in computations: e.g., numbers, prices, phone numbers.
- You can stipulate which types of **arguments** are valid inputs to an abstraction.
- You can stipulate which types of **results** an expression can compute.

25.1 Types

Not all values are the same type of value. Compare the following values:

5
Sally
(765) 546-2378
$2.50

The first item is a number (the number 5), the second item is a name (the name Sally), the third item is a domestic US phone number, and the fourth item is a unit of money, in USD.

For computational purposes, we can organize values into types, and give the types names. For instance, we could stipulate a type called *USD-unit*, and say that it is the collection of all values that are units of money in USD. Then we can say that $2.50 is a value of this type precisely because it is contained in the *USD-unit* collection.

Similarly, we could stipulate a type called *phone-numbers*, and say that it contains all domestic US phone-numbers. The value (765) 546-2378 is a value of this type precisely because it is contained in this collection.

25.2 Restricting Inputs

Once we have declared a set of types that are available for computations, we can stipulate which types computations can work with.

Suppose you have an abstraction that looks something like this:

$$\text{Replace “price” in (multiply 2 by price)} \tag{25.1}$$

As we know, we can stipulate a value to put in place of “price,” for example “5”:

$$\text{Replace “price” with “5” in (multiply 2 by price)} \tag{25.2}$$

When you do the replacement, you get this:

$$\text{multiply 2 by 5} \tag{25.3}$$

But what if you replace “price” with, say, “Tom”?

$$\text{Replace “price” with “Tom” in (multiply 2 by price)} \tag{25.4}$$

When you do that replacement, you get this:

$$\text{multiply 2 by Tom} \tag{25.5}$$

The untyped λ calculus allows this, because as far as it is concerned, it just manipulates symbols.

However, we might want to restrict the types of values you can put in the place of “price.” We might want to say something like this: “you can replace ‘price’ only with a unit in USD.”

To do that, we could write this out explicitly:

$$\text{Replace “price” with a USD-unit in (multiply 2 by price)} \tag{25.6}$$

Now it is clear that “Tom” doesn’t fit in the place of “price”. If you try it, you get this:

$$\text{Replace “price” with the USD-unit “Tom” in (multiply 2 by price)} \tag{25.7}$$

And that makes no sense, because “Tom” is not a unit of USD. However, if you select a unit of USD, then your expression does make sense:

$$\text{Replace “price” with the USD-unit “\$2.50” in (multiply 2 by price)} \tag{25.8}$$

25.3 Specifying Outputs

We can specify the type of the result (or output) of a calculation too.

Suppose we want to calculate the total cost of x pounds of sand. We know that each pound costs \$2.50. To do that calculation, we would multiply the number of pounds by \$2.50.

So, suppose we encode this with an abstraction. Like this:

$$\text{replace “lbs” in (multiply lbs by \$2.50)} \tag{25.9}$$

Obviously it would make no sense to put something like “Tom” in the place of “lbs”:

$$\text{replace “lbs” with “Tom” in (multiply lbs by \$2.50)} \tag{25.10}$$

So, we stipulate that the type of the replacement should be a unit of pounds:

$$\text{replace “lbs” with a lbs-unit in (multiply lbs by \$2.50)} \tag{25.11}$$

Now we can only put in lbs units. For instance, “2 lbs”:

$$\text{replace “lbs” with the lbs-unit “2 lbs” in (multiply lbs by \$2.50)} \tag{25.12}$$

If we do that replacement, we get:

$$\text{multiply 2 lbs by \$2.50} \tag{25.13}$$

Suppose that we go on to perform the calculation here — we multiply 2 (lbs) by 2.50 (USD). That gives us the total cost of the sand:

$$\$5.00 \tag{25.14}$$

This value is a unit of USD. So, the result of this calculation has a type too: it is a USD unit.

To specify this, we could explicitly say it in our expression:

$$\text{replace “lbs” with a lbs-unit in (multiply lbs by \$2.50) to get a USD-unit} \tag{25.15}$$

Now it is clear which type of input goes in, and which type of output comes out.

25.4 Arrows

We can specify input-output pair of types with an arrow. For instance, for the last expression, we can say this:

$$\textit{lbs-units} \to \textit{USD-units} \tag{25.16}$$

This is a way to specify the type of the expression. Hence:

- Collections are ways to specify the type of **values**.
- Arrows are ways to specify the type of **expressions**.

Let us use this vocabulary. Take the expression we formulated a moment ago:

$$\text{replace “lbs” with a lbs-unit in (multiply lbs by \$2.50) to get a USD-unit} \tag{25.17}$$

This is of the following type: *lbs-units* $\rightarrow$ *USD-units*, which means it is the type of expression that takes lbs-units for input, and it produces USD-units as output.

Many computations implicitly assume some sort of arrow typing like this. Consider the sort of basic arithmetic you might have done in school. For instance:

$$x + 5 \tag{25.18}$$

It is often not stated explicitly, but the unwritten assumption is that you can only put a number in the place of x, and that the result will also be a number. So this expression has the type:

$$\textit{number} \rightarrow \textit{number} \tag{25.19}$$

25.5 Collections and Arrows

In the foregoing, we did two things when it comes to types.

- We stipulated that there are certain types of values, i.e., collections of certain values. For instance, we said that there are USD-units, or lbs-units.
- We stipulated that expressions take inputs of a certain type, and they produce outputs of a certain type. For instance, we said that our “multiply lbs by \$2.50” expression takes lbs-units for input, and it produces USD-units as output.

We have names for these.

- We will call collections of values **collection** types, because each type is a collection of particular values. For instance, “USD-unit” is a collection type.
- We will call the input-output pairs **arrow** types, because they show how you go from one type of value to another. For instance, *lbs-unit* $\rightarrow$ *USD-units* is an arrow type.

Simple type theory provides a way for us to specify these sorts of type restrictions (though of course in a formal way, building on top of the λ calculus).

Chapter 26

Simple Types (Informally)

Let us think more about the two sorts of types we encountered before: collection types and arrow types.

26.1 Collection Types

What are collection types? Here is some terminology:

- A collection type is an exactly specified collection of items.
- We say that the items in the collection are the **inhabitants** of the type.

Some obvious examples are the different number types we learn about in school:

- The set of natural numbers, written as: $\mathbb{N}$. This is an exactly specified collection: it consists of all and only the natural numbers $0, 1, 2, \ldots$ It does not include any negative numbers, fractions, or decimals.
- The set of integers, written as $\mathbb{Z}$. This includes the natural numbers plus all negative whole numbers: $\ldots, -3, -2, -1, 0, 1, 2, 3 \ldots$
- The set of all fractions, which are called the "rational numbers" (from "ratio") because they can be represented as a quotient. Hence, the symbol for the rational numbers is $\mathbb{Q}$ (for "quotient").

You can create your own collection type, for any purpose whatever, simply by specifying a collection. The only requirement is that it must be exactly specified, and by that, I mean you must specify exactly which items belong in the collection. Here are some examples:

- The colors of the rainbow: red, orange, yellow, green, blue, and purple.
- The 2018 Porsche models: the 911, the Boxster, the Cayman, the Cayenne, the Macan, and the Panamera.

Collection types may have only a few inhabitants, like the two collections just mentioned. It is easy to specify such collections: you just list their inhabitants.

But some collection types have an infinite number of inhabitants — e.g., any of the number types we just mentioned, like integers or natural numbers.

For those that have an infinite number of inhabitants, you obviously cannot list them all. In that case, to define a collection type, you need to specify an exact way to **construct** each and every inhabitant of the type.

For example, any natural number can be constructed with a zero and a successor operation. To construct the first number after zero, you take the successor of zero. To construct the next number you take the successor of the successor of zero. To construct the next number you take the successor of the successor of the successor of zero, and so on.

26.2 Arrow Types

The second kind of type is the arrow. The inhabitants of an **arrow** type are mappings from one type to another.

Suppose we have two collection types: one is a collection of *employees*, and the other is a collection of *phone-numbers*. What we want to do is connect employees to phone numbers.

To do that, we might take a piece of paper, write all the employees on the left, and all the phone numbers on the right. Then we could draw an arrow from each employee to a phone number. Something like this:

$$\begin{tikzcd}
\text{Sally Farnston} \arrow[dr] & \text{567-2876} \\
\text{Harold Moore} \arrow[dr] & \text{227-9600} \\
\text{Jennifer Kowana} \arrow[uur] & \text{545-0428} \\
\cdots & \cdots
\end{tikzcd}$$

This is a mapping from employees to phone numbers. In set theory, we call this a **function**. It is essentially a look-up table — to each item on the left, you assign one item on the right — so that, for any given an employee, you can look up their phone number.

Of course, we could make a variety of different mappings between these two collections. We could draw the lines differently so that, say, Sally's phone number is 567-2876 instead of 227-9600.

But all such mappings have one thing in common: they map *employees* to *phone-numbers*. If we collect together all mappings from employees to phone numbers, we have an arrow type.

To specify this type, we write it like this:

$$employees \to phone\text{-}numbers \tag{26.1}$$

Chapter 27

Constructing Types

Let us begin to formulate the simple typed λ calculus. Instead of writing "simply typed λ," I will write "$\lambda \to$."

27.1 An Alphabet for Type Variables

A collection type is an exactly specified collection of values. We can give names to collection types. In our earlier example, we named the collection of USD values *USD-unit*.

We do not need to work with long names. We can just use single-character symbols. Let us establish an alphabet of symbols we can use to construct type names. We will call these **type variables**.

Definition 11 (Alphabet of Type Variables)**.** *Let $\mathbb{V}$ be an infinite set of lowercase Greek letter symbols, possibly with subscripted numbers: $\{\alpha, \beta, \gamma, \ldots, \kappa_1, \kappa_2, \ldots, \mu_{51}, \mu_{52}, \ldots\}$.*

As with the "variables" we used in the untyped λ, the word "variable" here has nothing to do with variables that we find in programming languages, first-order logic, and so on. In this context, it refers simply to the symbols from $\mathbb{V}$.

27.2 Building Type Variables

The simplest type you can construct is a collection. To pick a name for a collection, you simply need to select a symbol from $\mathbb{V}$, and wrap it in parentheses.

For example, suppose we want to name our USD-units. To do that, we select a symbol – say, τ — and wrap it in parentheses:

$$(\tau) \tag{27.1}$$

You can drop the parentheses if it results in no ambiguity. For example:

$$\tau \tag{27.2}$$

One thing to note about these names is this: after you select a name for a collection, you must use that name for that one collection only. Each name must be unique, so that it names one and only one collection.

27.3 Building Arrow Types

Once you have built some collection type variables, you can build up arrow types from them. To do that, you take two type variables, you put an arrow between them, and you wrap the whole thing in parentheses.

For instance, if (τ) is the name for USD-units, and (σ) is the name for lbs-units, then the arrow type from lbs-units to USD-units is this:

$$((\sigma) \to (\tau)) \tag{27.3}$$

We can drop extra parentheses if it results in no ambiguity. For example, we could write the above like this:

$$(\sigma) \to (\tau) \tag{27.4}$$

Or even this:

$$\sigma \to \tau \tag{27.5}$$

27.4 Nesting Arrow Types

You can also build further arrow types from other collection and arrow types. To do that, you take two types you have already built — they could be collection types or arrow types — then you put an arrow between them, and then you wrap the whole thing in parentheses.

For example, if we have a collection type (μ), and an arrow type $((\sigma) \to (\tau))$, we could put an arrow between them to make a new arrow type:

$$((\mu) \to ((\sigma) \to (\tau))) \tag{27.6}$$

We can drop the extra parentheses if it results in no ambiguity. For instance, the last type could be just this:

$$\mu \to (\sigma \to \tau) \tag{27.7}$$

But we cannot drop that last set of parentheses. For suppose we did:

$$\mu \to \sigma \to \tau \tag{27.8}$$

That is ambiguous. It could mean this:

$$\mu \to (\sigma \to \tau) \tag{27.9}$$

Or this:

$$(\mu \to \sigma) \to \tau \tag{27.10}$$

And those are different types.

27.5 Defining All Types

We can form all types using the two mechanisms I just described: you can select a symbol from $\mathbb{V}$ to form a type variable, or you can put two types together with an arrow. The types you put together to form an arrow can themselves be complex types, built from other types.

Definition 12 (Simple Types)**.** *The set of simple types $\mathbb{T}$ is the set of all strings of symbols that satisfy the following conditions:*

- *If $\alpha \in \mathbb{V}$, then $(\alpha) \in \mathbb{T}$.*
- *if $\sigma, \tau \in \mathbb{V}$, then $((\sigma) \to (\tau)) \in \mathbb{T}$.*

Parentheses can be dropped if it results in no ambiguity.

Chapter 28

Typed Terms

Suppose we have a term, call it M:

$$M \tag{28.1}$$

We can stipulate that this term has a particular type, say σ, like this:

$$M : \sigma \tag{28.2}$$

This is a **typed term**. The colon and the type variable is a **type decoration**, while the M (by itself, without the type decoration) is a **pretyped term**. In $\lambda \rightarrow$, we construct typed terms by adding type decorations to pretyped terms.

Pretyped terms look very similar to untyped λ terms, but there is a subtle difference. So let us first define pretyped terms. Then we can define typed terms.

28.1 Variables

First, we can select variables from V, just as we could before. To form a pretyped variable term, we take any symbol from V, and we wrap it in parentheses. For instance, we can take x and make a variable term like this:

$$(x) \tag{28.3}$$

And of course, we can drop parentheses if it leads to no ambiguity, e.g.:

$$x \tag{28.4}$$

28.2 Abstractions

Pretyped abstractions are formed by taking another pretytped term, and binding a typed variable in it. Let M be any pretyped term, let x be any variable symbol from V, and let σ be any type name from $\mathbb{V}$. Then a pretyped abstraction has this form:

$$(\lambda x : \sigma.M) \tag{28.5}$$

Parentheses can again be dropped if it leads to no ambiguity. For instance, we can do this:

$$\lambda x : \sigma.M \tag{28.6}$$

Note that there cannot be any untyped binding expressions in $\lambda x \rightarrow$. This is not a valid pretyped term:

$$\lambda x.M \tag{28.7}$$

28.3 Applications

Pretyped applications are formed by taking two pretyped terms, and putting them next to each other, in parentheses. Let P and Q stand for any pretyped terms. Then an application looks like this:

$$(PQ) \tag{28.8}$$

We can drop parentheses if it leads to no ambiguity, for instance:

$$PQ \tag{28.9}$$

Even though the (PQ) formulation looks very much like an application in the untyped λ, it is not the same. Here, P and Q must be selected from *pretyped* λ terms, not untyped λ terms.

Remember: P or Q can themselves be complex terms. And in $\lambda \rightarrow$, every nested term must be a pretyped term, not an untyped term. So, this is an invalid pretyped application term:

$$(\lambda x.x)y \tag{28.10}$$

It includes an untyped abstraction $(\lambda x.x)$, which is not valid in $\lambda \rightarrow$. But this is a valid application:

$$(\lambda x : \sigma.x)y \tag{28.11}$$

28.4 Definition of Pretyped Terms

With the foregoing information at hand, let us put together a definition of pretyped terms for $\lambda \rightarrow$. Here it is:

Definition 13 (Pretyped $\lambda \to$ terms)**.** *Let V be the alphabet of variable symbols defined before, and let $\mathbb{V}$ be the alphabet of type variables defined before. $\Lambda_{\mathbb{T}}$ is the set of all pretyped $\lambda \to$ terms that satisfy the following conditions:*

- *If $x \in V$, then $(x) \in \Lambda_{\mathbb{T}}$.*
- *If $M, N \in \Lambda_{\mathbb{T}}$, then $(MN) \in \Lambda_{\mathbb{T}}$.*
- *If $M \in \Lambda_{\mathbb{T}}$, $x \in V$, and $\sigma \in \mathbb{V}$, then $(\lambda x : \sigma.M) \in \Lambda_{\mathbb{T}}$.*

Parentheses can be dropped if it leads to no ambiguity.

28.5 Typed Terms

Now that we have defined the pretyped terms, we can see that a typed term will be any pretyped term followed by a type name.

Definition 14 (Typed Term)**.** *If $M \in \Lambda_{\mathbb{T}}$, and $\sigma \in \mathbb{V}$, then $M : \sigma$ is a typed term.*

Chapter 29

Typing Terms

One of the things we do a lot in $\lambda\rightarrow$ is declare that a term has a certain type. The notation we use for this has the following form:

$$M : \sigma \tag{29.1}$$

Here, M can be any pretyped term, and σ can be any type name from $\mathbb{V}$. We call this a **typing statement** (or just a **statement** for short). The term M is called the **subject** of the statement, and σ is the **type**.

Definition 15 (Typing statements)**.** *Let M stand for any pretyped term and let σ stand for any type variable from $\mathbb{V}$. Then $M : \sigma$ denotes that the term M has the type σ. The expression $M : \sigma$ is called a typing stetement, M is called the subject of the statement, and σ is called the type of the subject.*

29.1 Typing Free Variables

In $\lambda\rightarrow$, we begin by declaring the types of free variables.

To declare a type for a free variable, attach a type name (a symbol from $\mathbb{V}$) to the end. For instance, to say that a free variable x has the type σ, we write this:

$$x : \sigma \tag{29.2}$$

When we stipulate the type of a free variable like this, we call it a **declaration**, because we are declaring that this free variable has a certain type.

Note that a declaration is just a typing statement, where the subject is a free variable.

Definition 16 (Declaration)**.** *Let M stand for any pretyped term and let σ stand for any type variable from $\mathbb{V}$. The typing statement $M : \sigma$ is a declaration when the subject is a free variable from V.*

29.2 Contexts

When we evaluate expressions in $\lambda \rightarrow$, we do it in a context. A **context** is simply a list of free variables and their types. A context tells us what the types of the free variables are.

For example, suppose we have an expression with three free variables in it: x, y, and z. Suppose we want to evaluate the expression in a context where x and z have the type σ, while y has the type τ. To declare this context, we write this:

$$x : \sigma, y : \tau, z : \sigma \tag{29.3}$$

We typically use the capital Greek letter Γ (Gamma) to denote the context, so we could re-write the above like this:

$$\Gamma = x : \sigma, y : \tau, z : \sigma \tag{29.4}$$

Definition 17 (Contexts)**.** *A context Γ is a list with zero or more items D_1, D_2, …, D_n, where every D_i from $i = 1$ to n is a declaration for a fresh variable in V.*

Note that a context can have zero items in it. We call this an **empty context**.

29.3 Unique Types

Note that we said each declaration in a context is for a **fresh** variable.

This is because each free variable can have one and only one type in a context. If you assign σ to x, then x will have the type σ and only the type σ in that context.

If you ever see another type τ assigned to x, this can mean x is being attributed two types, which we would declare as "untypable" or impossible in $\lambda \rightarrow$.

However, there may be contexts where this does not necessarily mean that x has two different types. In those contexts, it can mean that τ and σ are the same type (they are different names for the same type).

The context usually makes it clear if σ and τ are supposed to show us that x is untypable, or if they are supposed to just be different names for the same type.

29.4 Typing Binding Variables

Variable symbols also occur as binding variables in abstractions. Consider this abstraction (where M can be any pretyped term):

$$\lambda x : \sigma.M \tag{29.5}$$

Binding variables must have types in $\lambda \rightarrow$. To declare the type of a binding variable, attach a type name to it, in the binding expression, as we have done here.

What is the meaning of the type decoration on the binding expression? It restricts the type of input you can apply the abstraction to. It says: whatever value we might want to substitute in for x, it *must* be a value with type σ.

29.5 Typing the Body of an Abstraction

The body of an abstraction must have a type too. To declare it, attach a type name to it.

For example, take the abstraction we looked at a moment ago:

$$\lambda x : \sigma.M \tag{29.6}$$

To declare that the term M has the type τ, you could write this:

$$\lambda x : \sigma.(M : \tau) \tag{29.7}$$

This says that the input (marked by the binding variable x) must have the type σ, while the body of the expression (marked by M) must have the type τ.

29.6 Typing the Result of an Abstraction

An abstraction of the form $\lambda x : \sigma.(M : \tau)$ says: "you can put in a value of type σ and get out a term M of type τ."

So, an abstraction has an arrow type. In this case, it goes from σ to τ.

$$\sigma \rightarrow \tau \tag{29.8}$$

We can attach that to the end of the whole abstraction to be fully explicit:

$$(\lambda x : \sigma.(M : \tau)) : \sigma \rightarrow \tau \tag{29.9}$$

This says that the whole expression takes σ as input, and if you reduce it, it gives you τ as output.

We do not need to explicitly write out the type of M. We can normally just write this:

$$(\lambda x : \sigma.M) : \sigma \rightarrow \tau \tag{29.10}$$

It is obvious from the arrow that M must have type τ.

29.7 Typing Applications

An application has the following form (where P and Q can be any pretyped terms):

$$(PQ) \tag{29.11}$$

Recall that an application stipulates a value that can be put into an abstraction. So P is meant to be an abstraction and Q is meant to be an argument for it.

Well, an abstraction has an arrow type. For instance, in our earlier example of an abstraction, it had the type $\sigma \to \tau$.

Hence, in an application (PQ), P will have an arrow type. Like this (where σ and τ can be any type variable from $\mathbb{V}$):

$$((P : \sigma \to \tau)Q) \tag{29.12}$$

But then, Q must be the right input type too, in order to be a value that can be substituted into the abstraction. So Q must have the type σ:

$$((P : \sigma \to \tau)(Q : \sigma)) \tag{29.13}$$

This is the basic rule for types in applications. The first term in the application must be an arrow type of the form $A \to B$, and the second term must match the input type A:

$$((P : A \to B)(Q : A)) \tag{29.14}$$

Provided that the two terms P and Q have types that match that pattern, then we can assign a type to the application.

So which type do we assign to an application, assuming that we can? Well, if you take the application and do the reduction, you get the final result of the arrow: B. That is, if $P : A \to B$ and $Q : A$, then:

$$(PQ) : B \tag{29.15}$$

29.8 Summary

The typing rules for variables, abstractions, and applications, are as follows:

- Variables — A typed variable can be constructed by taking a variable x from V and a type variable A from $\mathbb{V}$, and putting them together like this: $x : A$. The variable x has the type A.

- Abstractions — A typed abstraction can be constructed by taking a variable x from V, a type variable A from $\mathbb{V}$, and a typed term $M : B$, and putting them together like this: $\lambda x : A.M : A \to B$. This abstraction has the type $A \to B$.

- Applications — A typed application can be constructed by taking two typed terms $M : A \to B$ and $N : A$, and putting them together like this: $MN : B$. This term has the type B. Applications can *only* be typed if M has an arrow type and N has the same type as the first item in M's arrow.

Chapter 30

Judging Types

An interesting fact about $\lambda\rightarrow$ is this: once you know the types of the free and bound variables, you can work out the other types.

30.1 Judgments

When you try to work out the type of an expression, you end up saying that you think "this term M has the particular type σ, in this particular context Γ." We call this a **judgment** (you might call it a "type judgment"). We have a special way to write down such type judgments, which runs as follows.

Let Γ stand for any context, let M stand for any pretyped term, and let σ stand for any type variable from $\mathbb{V}$. Then we can say that "M has type σ in the context Γ" like this:

$$\Gamma \vdash M : \sigma \tag{30.1}$$

The right turnstile character ($\vdash$) is a special symbol that we can call the **derivable symbol**. You can read the above expression like this: "$M : \sigma$ is derivable in/from Γ," which means that, given the context Γ, we can work out that M has the type σ.

Definition 18 (Judgments). *Let Γ stand for any context, let M stand for any pretyped term, and let σ stand for any type variable from $\mathbb{V}$. Then $\Gamma \vdash M : \sigma$ is a judgment that M has type σ in context Γ"*

If we want to say that a term having a certain type is *not* derivable in a given context, we can draw a slash through the derivability symbol:

$$\Gamma \nvdash M : \sigma \tag{30.2}$$

30.2 An Example

Suppose we have this term:

$$(\lambda x : \sigma.z)(y) \tag{30.3}$$

Let us make a judgment and claim that, in a context where y has the type σ and z has the type τ, the result of this expression will have the type τ. Here is our judgment:

$$y : \sigma, z : \tau \vdash ((\lambda x : \sigma.z)(y)) : \tau \tag{30.4}$$

This judgment is correct. We will discuss how to derive this formally soon. But for the moment, let us walk through how this works informally, step by step.

First, we have our context Γ, which assigns to y the type σ and z the type τ:

$$\Gamma = y : \sigma, z : \tau \tag{30.5}$$

Next, we have the term M we want to assign a type to:

$$M = (\lambda x : \sigma.z)(y) \tag{30.6}$$

This is an application of the form:

$$M = (PQ) \tag{30.7}$$

Let's work out the types for P and Q independently. Then we can put it all together. Let's take P first.

What is the arrow type of P? This term is an abstraction:

$$P = \lambda x : \sigma.z \tag{30.8}$$

It takes an input of type σ, and it returns the term z (which in this context has the type τ). The type of this abstraction is thus:

$$P = (\lambda x : \sigma z) : \sigma \to \tau \tag{30.9}$$

What about the type of Q? In this context, y has the type σ:

$$Q = y : \sigma \tag{30.10}$$

So our application PQ has this type:

$$M = ((P : \sigma \to \tau)(Q : \sigma)) \tag{30.11}$$

Recall that an application must have a particular set of types. The first term P must have an arrow type of the form $A \to B$, and the second term Q must have the type A. And if that is so, then the whole application has the type B.

In our case, M does have the right pattern of types, so we can assign it the term τ:

$$M : \tau \tag{30.12}$$

So our original term has the type τ:

$$M = ((\lambda x : \sigma.z)(y)) : \tau \tag{30.13}$$

Given the context Γ. Our initial judgment is thus correct:

$$y : \sigma, z : \tau \vdash ((\lambda x : \sigma.z)(y)) : \tau \tag{30.14}$$

30.3 An Incorrect Judgment

Suppose now that we have a different context. For instance, suppose we assign y a type μ, but keep z's type τ:

$$\Gamma = y : \mu, z : \tau \tag{30.15}$$

Would it then be the case that our original expression has the type τ?

$$((\lambda x : \sigma.z)(y)) : \tau \tag{30.16}$$

That is, could we make this judgment?

$$y : \mu, z : \tau \vdash ((\lambda x : \sigma.z)(y)) : \tau \tag{30.17}$$

The answer here is no. As we saw above, the abstraction requires an input of type σ, but in this context, y does not have the type σ. So, y cannot be substituted into the abstraction. Hence this judgment is not correct, which we write like this:

$$y : \mu, z : \tau \nvdash ((\lambda x : \sigma.z)(y)) : \tau \tag{30.18}$$

30.4 Untypable Terms

In $\lambda \rightarrow$, not every term can be assigned a type. Consider this term:

$$M = (xy) \tag{30.19}$$

Recall that this can only be typed if x has an arrow type of the form $A \rightarrow B$ and y has the type A.

So, let us suppose that we have a context like this:

$$\Gamma = x : \sigma \rightarrow \tau, y : \sigma \tag{30.20}$$

In this context, x and y have the relevant sort of types, so (xy) can be assigned a type, namely τ:

$$x : \sigma \rightarrow \tau, y : \sigma \vdash (xy) : \tau \tag{30.21}$$

But if y does not have the right type, it cannot serve as an input for the first term. For instance, take this context:

$$\Gamma = x : \sigma \to \tau, y : \tau \tag{30.22}$$

In this context, x and y do not have the right sort of types, so (xy) cannot be given a type. Hence:

$$x : \sigma \to \tau, y : \tau \nvdash (xy) : \tau \tag{30.23}$$

Here is another example. Consider this term:

$$M = (xx) \tag{30.24}$$

Since this is an application, the first term must have types of the form $A \to B$, and the second term must have the type A:

$$((x : A \to B)(x : A)) \tag{30.25}$$

As we said above, variables are assigned unique types. They cannot have two types at the same time.

But here, in order for the term (xx) to get a type, then x must have the type $A \to B$, *and* it must have the type A, which is impossible.

So (xx) cannot receive a type, in any context.

30.5 Typable Terms

Since some terms can be typed and some cannot, let us say that some terms are typable. A term is **typable** if it can be assigned a type (in some context).

Definition 19 (Typable Terms)**.** *Let Γ stand for any context, let M stand for any pretyped term, and let σ stand for any type variable from $\mathbb{V}$. M is typable if there is a context Γ and a type σ such that $M : \sigma$ in Γ.*

Chapter 31

About Derivations

A judgment that a pretyped term M has type σ in context Γ is written like this:

$$\Gamma \vdash M : \sigma \tag{31.1}$$

There is a formal way to derive this judgment (or to show that it cannot be derived, if there is no way to derive it). This system is called a derivation system.

A **derivation** system is a set of rules that tell you how you can formally derive judgments. You construct a derivation by chaining derivations: you start by using a rule to derive an intermediate judgment, then you use another rule to derive a further intermediate judgment, and so on until you reach your goal.

You can think of a derivation as a board game. The derivation rules tell you exactly which moves you can make to win the game. You win the game if you chain together a sequence of moves that get you to your goal.

31.1 Specifying Derivation Rules

We write derivation rules in this form:

$$\text{Rule Name}\ \frac{\text{judgment } A \qquad \text{judgment } B \qquad \ldots}{\text{judgment } C}$$

This says that if you already have judgment A, B, and so on, then you can infer judgment C. Of course, this is just a template for a rule. In a real rule:

- The words "judgment A," "judgment B," …, "judgment C" will be replaced by particular types of judgments, of the form "$\Gamma \vdash M : \sigma$."
- The words "Rule Name" will be replaced by the name for the rule.

We call the judgments above the line the **premises** of the derivation, and we call the judgment underneath the line the **conclusion**.

Some derivation rules require no premises. They are written like this:

$$\text{Rule Name}\ \frac{\varnothing}{\text{judgment } A}$$

A rule of this form says: you do not need any judgments to be already established. You can simply assert judgment A at any time in the game.

31.2 Examples of Rules

Let us introduce a few rules as examples. Here is one rule:

$$\text{Variable Rule (var)}\ \frac{\varnothing}{a : A \vdash a : A}$$

This rule says: if a variable a has type A in the context, then you can copy that variable (with its type) into the judgment and say: a has type A.

Here is another rule:

$$\text{Application Rule (appl)}\ \frac{\Gamma \vdash P : A \to B \qquad \Gamma \vdash Q : A}{\Gamma \vdash PQ : B}$$

This rule says that if you established that a term P has type $A \to B$ and you also have established that a term Q has type A, then you can infer that the application PQ has type B.

31.3 Derivations (Trees)

You construct a derivation by chaining moves together. To make each move, you use one of the rules.

Each derivation rule tells you how to derive a certain judgment. So, after each move, you will have a judgment. That judgment can then become a premise for the next move.

Let us look at an example. Suppose we have a context where x has type $\sigma \to \tau$ and y has type σ:

$$\Gamma = x : \sigma \to \tau, y : \sigma \tag{31.2}$$

What we want to derive is that the application xy has the type τ, in this context:

$$x : \sigma \to \tau, y : \sigma \vdash xy : \tau \tag{31.3}$$

To begin, let us use the *var* rule to assert that x has type $\sigma \to \tau$. To begin, we start by writing a horizontal line:

$$\frac{}{}$$

Then we write the premises we use above the line, and the conclusion we derive below the line.

The *var* rule has no premises, so we can write that:

$$\frac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau}$$

Then we write the conclusion we derive underneath the line:

$$\frac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau}$$

And finally, make a note to say which rule we used by writing it to the left of the line:

$$\text{var}\ \frac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau}$$

This shows that we used the *var* rule to derive the judgment $x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau$.

Next, let us us the *var* rule again to derive that y has type σ. Over to the right, draw a horizontal line, and note that there are no premises:

$$\text{var}\ \frac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau} \qquad \frac{\varnothing}{\qquad\qquad\qquad}$$

Now fill in the judgment underneath the line:

$$\text{var}\ \frac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau} \qquad \frac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash y : \sigma}$$

Make a note that we have used the *var* rule again, by writing *var* next to the line:

$$\text{var}\ \frac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau} \qquad \frac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash y : \sigma}\ \text{var}$$

At this point, we have derived two judgments, side by side. One says that x has type $\sigma \to \tau$ and the other says that y has type σ.

These are exactly the two premises we need to apply the *appl* rule. So, let's make those two judgments into premises for an *appl* derivation.

Draw a line underneath the two judgments, to say that they are now going to be the premises of a new derivation:

$$\frac{\text{var}\ \dfrac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau} \qquad \dfrac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash y : \sigma}\ \text{var}}{\qquad}$$

And then insert the judgment that xy has type τ:

$$\frac{\text{var}\ \dfrac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau} \qquad \dfrac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash y : \sigma}\ \text{var}}{x : \sigma \to \tau, y : \sigma \vdash xy : \tau}$$

Make a note that we used the *appl* rule by writing it to the left of the line:

$$\text{appl}\;\dfrac{\text{var}\;\dfrac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau} \qquad \dfrac{\varnothing}{x : \sigma \to \tau, y : \sigma \vdash y : \sigma}\;\text{var}}{x : \sigma \to \tau, y : \sigma \vdash xy : \tau}$$

At this point, we have reached our goal. We wanted to show that xy has type τ, given the context $x : \sigma \to \tau, y : \sigma$. And we did that by chaining derivations: we use the *var* rule twice to generate two intermediate judgments, and then we used those judgments as premises for our final derivation.

31.4 Derivation Tree Structure

Notice that the derivation we just constructed has a tree structure. To make this explicit, we can draw it like this:

$$\begin{tikzcd}
\overset{x : \sigma \to \tau, y : \sigma \vdash y : \sigma}{\bullet} \arrow[dr] & & \overset{x : \sigma \to \tau, y : \sigma \vdash xy : \tau}{\bullet} \arrow[dl] \\
& \underset{x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau}{\bullet} &
\end{tikzcd}$$

On the top left branch, we have the first judgment we established with the *var* rule. On the top right branch, we have the second judgment we established with the *var* rule.

Then, those two branches together join to form the bottom node, where we have the final judgment we established with the *appl* rule.

Chapter 32

Derivations in a Linear Format

Complex derivations are too big to fit on a page. To make it easier to encode these derivations, we can write the same information down in a linear format.

32.1 Linear Format

In the linear format, we write each step down on its own line. Let us repeat the derivation we constructed above, in the linear format.

To begin, start with a line, numbered as 1:

$$1 \quad \ldots$$

Next, we want to write down our inital premise, which is nothing:

$$1 \quad \varnothing$$

Then, we want to do the first *var* judgment. Write that down on line 2:

$$\begin{array}{ll} 1 & \varnothing \\ 2 & x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau \end{array}$$

To the right of that line, we want to indicate which rule we used to get the judgment. We can do that by writing *var*, 1:

$$\begin{array}{lll} 1 & \varnothing & \\ 2 & x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau & \text{var, 1} \end{array}$$

This says that we started with nothing (line 1), then we derived the judgment we see on line 2. We got line 2 by using the *var* rule, using as a premise line 1 (which is nothing).

Next, write down our second judgment, which was also derived using the *var* rule:

1	$\varnothing$	
2	$x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau$	var, 1
3	$x : \sigma \to \tau, y : \sigma \vdash y : \sigma$	var, 1

Finally, write down the the application:

1	$\varnothing$	
2	$x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau$	var, 1
3	$x : \sigma \to \tau, y : \sigma \vdash y : \sigma$	var, 1
4	$x : \sigma \to \tau, y : \sigma \vdash xy : \tau$	

The rule we used to get this is the *appl* rule, and the premises are lines 2 and 3, so we write that to the right of line 4:

1	$\varnothing$	
2	$x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau$	var, 1
3	$x : \sigma \to \tau, y : \sigma \vdash y : \sigma$	var, 1
4	$x : \sigma \to \tau, y : \sigma \vdash xy : \tau$	appl, 2, 3

And with that, we have completed the derivation.

This version encodes the same information as the tree format. But each step is on its own line, so it is easier to fit it onto one or more pages. Also, the justification for each step is written to the right of each line.

Some of the information is unnecessary. For instance, we don't need the empty premise at the beginning. We all know that the *var* rule requires no premises. So, we can write the above like this:

1	$x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau$	var
2	$x : \sigma \to \tau, y : \sigma \vdash y : \sigma$	var
3	$x : \sigma \to \tau, y : \sigma \vdash xy : \tau$	appl, 1, 2

32.2 Context

It can be tedious to write the context over and over again in these derivations. It is also a source for human error.

We can make some changes to how we write the linear format to make this easier. To begin, put the first declaration from the context ($x : \sigma \to \tau$) on its own line:

$$1 \quad x : \sigma \to \tau$$

Now, underline it, and draw a vertical line down the side of it, like this:

$$\begin{array}{r|l} 1 & \underline{\;x : \sigma \to \tau\;\;} \\ & \\ & \end{array}$$

We can now put other steps in the derivation underneath this declaration, indented some. Like this:

```
1 │ x : σ → τ
  ├──────────
  │
  │ 2   ...
  │ 3   ...
  │ ⋮
```

The declaration on line 1 now provides a context that applies to all the lines that are nested inside it.

In this example, the declaration on line 1 would be the context for lines 2 and 3. Another way to put it: lines 2 and 3 inherit the context of their parent.

We can add more declarations to the context by nesting them. Let's add the second declaration from our context: $y : \sigma$.

```
1 │ x : σ → τ
  ├────────
  │
  │ 2 │ y : σ
  │   ├──────────
  │   │
  │   │ 3   ...
  │   │ 4   ...
  │   │ ⋮
  │
```

Now we have two declarations to provide context. Lines 3 and 4 inherit the context of their parent declarations, so lines 3 and 4 have as their context both $y : \sigma$ and $x : \sigma \to \tau$. That is to say, their context is:

$$\Gamma = x : \sigma \to \tau, y : \sigma \tag{32.1}$$

Now that we have the context, we can continue with our derivation. First, let us use the *var* rule to assert that x has type $x : \sigma \to \tau$:

```
1 │ x : σ → τ
  ├────────
  │
  │ 2 │ y : σ
  │   ├──────────────
  │   │
  │   │ 3   x : σ → τ
  │   │ ⋮
  │
```

And to the right of that line, write down the rule we used to get that judgment:

$$
\begin{array}{r|l}
1 & x : \sigma \to \tau \\ \hline
 & \begin{array}{r|lll}
 2 & y : \sigma & & \\ \hline
 & 3 & x : \sigma \to \tau & \text{var, } 1 \\
 & \vdots & &
 \end{array} \\
 &
\end{array}
$$

This line offers a rather condensed form of a judgment. The judgment itself is that x has the type $\sigma \to \tau$:

$$\Gamma \vdash x : \sigma \to \tau \tag{32.2}$$

But the context (Γ) is not stated explicitly on line 3. Instead, it is inherited from the parent lines 1 and 2. If we fill that in, we can see that line 3 is saying this:

$$x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau \tag{32.3}$$

The last important piece of information on line 3 is the justification that is written to the right of the line: we used the *var* rule to derive this judgment, and in particular we refer back to line 1, where x is declared to have the type $\sigma \to \tau$. So, the whole judgment in the derivation that we have encoded on line 3 is this:

$$x : \sigma \to \tau, y : \sigma \vdash x : \sigma \to \tau \qquad \text{var} \tag{32.4}$$

Next, we can use the *var* rule a second time to derive the judgment that y has the type σ:

$$
\begin{array}{r|l}
1 & x : \sigma \to \tau \\ \hline
 & \begin{array}{r|lll}
 2 & y : \sigma & & \\ \hline
 & 3 & x : \sigma \to \tau & \text{var, } 1 \\
 & 4 & y : \sigma & \text{var, } 2 \\
 & \vdots & &
 \end{array} \\
 &
\end{array}
$$

And finally, we can make the judgment that the application xy has the type τ, using the *appl* rule with lines 3 and 4:

$$\begin{array}{r|l}
1 & x : \sigma \to \tau \\ \hline
 & \begin{array}{r|lll}
2 & y : \sigma & & \\ \hline
 & 3 & x : \sigma \to \tau & \text{var, 1} \\
 & 4 & y : \sigma & \text{var, 2} \\
 & 5 & xy : \tau & \text{appl, 3, 4}
\end{array}
\end{array}$$

That completes our derivation.

This is equivalent to the derivations we did earlier, in the tree format, and in the linear format where we wrote out the full context on each line.

This nested way to write derivations is more compact, and because of the vertical lines and indentation, the governing context is marked visually.

Chapter 33

Derivation Rules

Let us examine the derivation rules for $\lambda \rightarrow$.

33.1 The Variable Rule

We already saw the variable rule. It says: if you have a variable with a type in a context, you can copy that variable into the judgment. Formally, the rule is defined as follows.

Definition 20 (The Variable Derivation Rule)**.** *Let* Γ *stand for any context, let* x *stand for any free variable from* V*, and let* A *stand for any type variable from* $\mathbb{V}$*. Then the variable derivation rule is defined as follows:*

$$\textit{Variable Rule (var)}\,\frac{\varnothing}{\Gamma, x : A \vdash x : A}$$

Note the following about this rule:

- There are no required premises. So you do not need to establish any other judgments first, before you can use this rule. You can make a judgment of this sort at any point in a derivation.
- The context is written here as "$\Gamma, x : A$." This means that the context includes whatever might be in the context (we represent that simply by writing Γ), plus $x : A$.
- No matter what else may be in Γ, so long as $x : A$ is in the context, then you can copy $x : A$ from the context to the judgment.

33.2 The Meaning of the Rule

The variable rule may seem to be almost too obvious to write down. If you declare a free variable in the context, then *of course* that variable will have that type.

But this is only obvious because our minds are able to make this inference so quickly and easily. A computer or machine does not know how to do this though.

Suppose you have a context like this:

$$\Gamma = x : \sigma, y : \tau \tag{33.1}$$

Is the following judgment correct, given that context?

$$\Gamma \vdash x : \sigma \tag{33.2}$$

Yes, it is correct, because x is assigned the type σ in the given context.

Consider a second judgment. Is the following correct?

$$\Gamma \vdash y : \tau \tag{33.3}$$

This too is correct, because y is assigned the type τ in the given context.

Now consider a third judgment. Is this one correct?

$$\Gamma \vdash x : \tau \tag{33.4}$$

No, this is not correct, because x is not assigned the type τ in the given context. On the contrary, x is assigned the type σ.

Consider a fourth and final judgment. Is this one correct?

$$\Gamma \vdash z : \mu \tag{33.5}$$

No, this is not correct, because z is not assigned any type in the context.

How would a computer be able to determine that the last two are not correct, while the first two are correct?

We would have to tell it the rule: the only judgments about variables that are correct are those that copy a variable and its type directly out of the context. And that is exactly what the the variable derivation rule says.

If you think about it, the rule makes perfect sense. If you are evaluating an expression in a particular context, what types will any of the variables you encounter have? They will only have the types they have *in that particular context.* In a different context, they could have different types.

33.3 The Application Derivation Rule

We have already seen the application derivation rule too. It says the following. Suppose you have already established two judgments: one says that a term P has the type $A \to B$. The other says another term Q has the type A. If you have both of those judgments, you can then say that the application PQ has the type B, because Q has the right input type for P.

Definition 21 (The Application Derivation Rule)**.** *Let Γ stand for any context, let P and Q stand for any pretyped terms in $\Lambda_{\mathbb{T}}$, and let A and B stand for any type variables from $\mathbb{V}$. Then the application derivation rule is defined as follows:*

$$\textit{Application Rule (appl)}\ \frac{\Gamma \vdash P : A \to B \qquad \Gamma \vdash Q : A}{\Gamma \vdash PQ : B}$$

Note the following about this rule:

- The first term P has an arrow type: it goes from A to B. This means it can only take one sort of input: terms with type A.
- The second term Q has the correct input type: A.
- So, together, the two can form an application.
- This rule has two premises, which are judgments. So, this rule requires that both judgments have already been derived. This rule cannot be used until both of the requisite judgments have been derived by different derivations.

33.4 The Meaning of the Application Rule

The application rule captures our intuitions about functions or computations. If the first term P is a function that computes some output, then that means two things:

- It takes a certain type of input (which we label as being of type A).
- It computes a certain type of output (which we label as being of type B).

As an informal example, recall this equation:

$$x + 5 \tag{33.6}$$

This takes a particular type of input. It takes only numbers. It makes no sense to fill in x with a name, or a phone-number, or anything else that cannot be added to 5.

Likewise, assuming that a suitable value has been provided for x, this equation produces a particular type of output. It produces a number, not a name, or a phone-number, or anything else that is not numeric.

So this equation has an arrow type:

$$\text{number} \to \text{number} \tag{33.7}$$

In an application, the first term P must be like this. It must be an arrow-type term. A non-arrow typed term is not susceptible of application.

Similarly, the second term Q must be a value from the correct type. As we noted, we cannot fill in x with a name, a phone-number, or whatever. It must be the right type of value.

But if we get both P and Q having these sorts of types, then the application makes sense: then P with type $A \to B$ can take the input Q with type A, and it will compute the output with type B.

So the application PQ will have the type B exactly when P has the type $A \to B$ and Q has the type A. It cannot coherently be given a type under any other circumstances.

(Informally, we might think of this rule like the rule that classical logicians call *modus ponens*: if you have A, and you also know that if A then B, then you can infer B.)

33.5 Abstractions

Finally, let us define the abstraction derivation rule.

Suppose you have a free variable x with the type A. We can encode this by saying that $x : A$ is in the context Γ:

$$\Gamma, x : A \tag{33.8}$$

Suppose also that you have already derived a judgment which says that a term M has the type B:

$$\Gamma, x : A \vdash M : B \tag{33.9}$$

Now suppose you want to form an abstraction from these, with x as the binding variable, and M as the body:

$$\lambda x : A.M \tag{33.10}$$

You can make the judgment that the type of this abstraction will be $A \to B$, because it takes as input a value of type A, and it produces as output a value of type B.

Definition 22 (The Abstraction Derivation Rule)**.** *Let Γ stand for any context, let x stand for an variable from V, let M stand for any pretyped term in $\Lambda_{\mathbb{T}}$, and let A and B stand for any type variables from $\mathbb{V}$. Then the abstraction derivation rule is defined as follows:*

$$\textit{Abstraction Rule (abstr)}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}$$

Note the following about this rule:

- This judgment has a premise, which is to say that it requires that you've already derived a judgment of the form $\Gamma, x : A \vdash M : B$.
- The free variable $x : A$ is included in the context, but note that that variable may occur free in M.
- The judgment you derive here takes the free variable $x : A$, and it binds it in M.

33.6 The Meaning of the Abstraction Rule

Think about the premise of the abstraction rule:

$$\Gamma, x : A \vdash M : B \tag{33.11}$$

Here we have a term M, which has the type B. In the context, there is a free variable x.

This variable — x — might be used in M, or it might not be. Remember that M can itself be a complex term, made of other subterms. And there is nothing in the abstraction derivation rule that prevents x from appearing as a free subterm in M.

Similarly, there is nothing about the rule that says x must appear in M. It could just as well not be a part of M.

Now think about the judgment that we derive from the abstraction rule:

$$\Gamma \vdash \lambda x : A.M : A \to B \tag{33.12}$$

Now the x gets bound. So, $x : A$ no longer appears in the context. It has been removed from the context (because the context includes only free variables). It is now bound in the term M.

This means that if x was free in M before, now it is bound.

The type of an abstraction is an arrow. An abstraction takes an input for x, and it replaces every occurrence of x with that input in the term M.

So, you start with x, and you get M. The type of the abstraction is thus an arrow from the type of the input x (which we have labeled A), to the type of the term M (which we have labeled B).

33.7 Legal Terms

The three derivation rules we discussed provide a way to construct every possible judgment that can be derived from any context. You can take any context, and systematically apply the rules, and in this way you could put together every possible derivation.

Definition 23 ($\lambda \to$ Judgments). *Let M stand for any pretyped term in $\Lambda_{\mathbb{T}}$, let A stand for any type variable from $\mathbb{V}$, and let Γ stand for any context. $\mathbb{J}$ is the smallest set of all judgments $\Gamma \vdash M : A$ derived by repeated application of the variable, application, or abstraction rules. We can also call $\mathbb{J}$ the set of all derivations, as a synonym.*

We say that a term is **legal** if there is a context in which it can be derived. More exactly, there needs to be a context and a type, so that in that context, the term can be derived with that type.

Definition 24 (Legal Terms). *Let M stand for any pretyped term in $\Lambda_{\mathbb{T}}$, let A stand for any type variable from $\mathbb{V}$, and let Γ stand for any context. M is a legal term iff there is a context Γ and a type A such that $\Gamma \vdash M : A$.*

Chapter 34

Structural Induction

To prove things in $\lambda \rightarrow$ (and other forms of λ), we need to use a technique we call **structural induction**.

34.1 Recursive Constructions

One way to define a collection is to list out every element in the collection. Another way is to specify how you construct the collection. To do that, you must specify three things:

- **Base cases** — You must specify which items we start with. The collection can have zero or more initial elements. Each of these initial elements is called a base case.
- **Constructors** — You must specify one or more rules that tell you how to build further items from the elements already in the collection. Each of the rules is called a constructor.
- **Closure** — You must specify that the collection is closed, i.e., that there are no elements beyond the base cases and the ones we build through the constructors. A matter of terminology: sometimes we say the collection is **closed** by the base cases and the constructor, or we say the collection is the **smallest set** that we can build from the base cases and the constructors, or we say that the collection is the **closure** over/of the base cases and constructors. However we say it, the point is just that the collection is a private club: members only (where membership is defined to include only the founding members (the base cases) and any other member who is brought into the club by a constructor). In most logic and mathematics books, when you see a definition of this sort, the closure condition is omitted, because it is assumed to be obvious.

An important fact to notice about definitions like this is that the constructors can be applied again and again, to repeatedly construct newer members of the collection.

Because of this, we call such constructions **recursive** (look up the definition of that word). So, recursive definitions are definitions that stipulate base cases and constructors, with the (explicit or implicit) assumption of closure.

34.2 An Example

Consider the natural numbers (denoted as $\mathbb{N}$). The natural numbers are all the positive whole numbers:

$$\mathbb{N} = \{0, 1, 2, 3, \ldots\} \tag{34.1}$$

We obviously cannot define this collection by listing out each and every natural number. There are infinitely many, so if we sat down and started writing them out, we would never finish (even if someone took our places after we died, and someone took their places, and someone took their places, and so on).

But we can define $\mathbb{N}$ recursively. First we need to start with an initial set of items — the base cases. For $\mathbb{N}$, we have one base case: the number zero. So we say that 0 (zero) is in $\mathbb{N}$. In set theory symbols, we say $0 \in \mathbb{N}$:

$$\text{Base case: } 0$$

Next, we need to specify how you build other items. For this, we can introduce a constructor that will construct the next natural number. Let's call this S (short for the "successor"), and let's define it like this: it can take any x from $\mathbb{N}$, and produce $x + 1$. We can write $S(x)$ to mean "the successor of x:

$$\text{Constructor: } S(x) = x + 1$$

If we put these together, we can define the natural numbers, recursively:

Definition 25 (The natural numbers)**.** *Let the set of natural numbers $\mathbb{N}$ be the set that satisfies the following conditions:*

- *Base case:* $0 \in \mathbb{N}$.
- *Constructor: if $x \in \mathbb{N}$, then $S(x) \in \mathbb{N}$.*
- *Closure: there are no elements in $\mathbb{N}$ besides the base case and elements built from S.*

We could also define the natural numbers using "smallest set" terminology, instead of "closure" terminology:

Definition 26 (The natural numbers, alternative definition)**.** *Let the set of natural numbers $\mathbb{N}$ be the smallest set that satisfies the following conditions:*

- *Base case:* $0 \in \mathbb{N}$.

- *Constructor: if $x \in \mathbb{N}$, then $S(x) \in \mathbb{N}$.*

This definition (and the one before it) is a recursive definition in just the way we discussed above. It does not list every element in $\mathbb{N}$. Rather, it tells us how to construct $\mathbb{N}$.

It does this by first telling us about the set we start with (the one base case):

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

Then it tells us how to construct more items, by using the constructor S. So, if we start with the base case, there is only one item in the set, namely zero. So we apply the successor to that, to build a new element:

$$\mathbb{N} = \{0, S(0)\}$$

Of course, we can *write* the second element as "1" (one), but that is just a matter of notation. We could also write it in binary (01), or in hexadecimal (0x000001), and so on. These are all just different notations for the same number. Underneath it all, the natural number "1" is just the item you get when you apply the constructor S to 0.

Now we have two items in the set. We can apply S again to build a new item. Of course, we might try to apply S to 0 again, which would give us:

$$\mathbb{N} = \{0, S(0), S(0)\}$$

But in sets, repeated elements are ignored (dropped), so that puts us back to where we were:

$$\mathbb{N} = \{0, S(0)\}$$

However, we can still apply the constructor to the second element in the set, namely "$S(0)$". And that gives us "$S(S(0))$":

$$\mathbb{N} = \{0, S(0), S(S(0))\}$$

Now we have a new number, beyond "0" and "$S(0)$." Again, we could *write* this new number as 2, but we could also write it in binary (10), hexadecimal, or whatever. Underneath it all, the number 2 is just the item we get by applying the constructor S to 0, and then applying S again to that result.

We could go on like this, to construct more and more natural numbers, but we needn't do that. The point is clear enough: our recursive definition provides us with a complete description of the set. The set of natural numbers is precisely the smallest set of items you get by starting with 0, and then applying the successor constructor over and over to generate new elements.

34.3 Structural Induction (Informally)

Now that we know how to construct collections recursively, we can turn to how we prove things about these collections.

Let us suppose that we want to prove something about every item in the collection. Let us say that P is the property we are interested in, and we want to prove that P holds for every single item in the collection. To prove this, we need to do two things:

- We must prove that P holds for each base case.
- We must prove that each constructor preserves P — that is, if the constructor starts with an item x that P holds for, after it constructs a new element y, P holds for y too.

If we can do both of these things, then we can conclude that P holds for *every* element of the collection. This type of proof is called a **structural induction** proof.

34.4 Example

As an example, let us prove that every natural number is divisible by one. To do this, we will put together a structural induction proof.

First, we want to show that each base case is divisible by one. The base case of $\mathbb{N}$ is zero, and zero is in fact divisible by one. Indeed, $\frac{0}{1}$ is a real fraction. So, we can just write that:

$$\text{Base case: 0 is divisible by 1, since } \tfrac{0}{1} = 0.$$

Next, we want to show that the constructor preserves the property of being divisible by one.

To do this, let's pick some arbitrary number, and call it k. We don't actually need to know which number k stands for. The point is just that it is some arbitrary member of $\mathbb{N}$.

Then, we apply the successor to it, which will give us $k + 1$. So, $S(k)$ is the same as $k + 1$:

$$S(k) = k + 1 \tag{34.2}$$

Now we want to show that the successor preserves the property of being divisible by one. That is, we want to show that, if k is divisible by one, then when we apply the successor to produce a new number, $k + 1$, that is divisible by one too.

This is, of course, obvious, since any number divided by one is just itself, regardless of whether the number in question is k, $k + 1$, $k + 2$, or whatever. So we can just write that:

> Constructor case: Let k by an arbitrary element from $\mathbb{N}$. The successor of k is $S(k)$, which is $k+1$. Both k and $k+1$ are divisible by one: k is divisible by one, since $\frac{k}{1} = k$, and $k+1$ is divisible by one, since $\frac{k+1}{1} = k+1$.

That covers all the base cases, and all the constructor cases. So our proof is done.

34.5 Proof Template

We can use a general template to write out structural induction proofs, and then fill in the blanks. Here is a template:

Theorem. *Let A be the collection of ______, and let P be the property of ______. $\forall x \in A, P(x)$.*

Proof. By structural induction.

- Base case 1. ______.
- Base case 2. ______.
- $\vdots$
- Constructor 1. ______.
- Constructor 2. ______.
- $\vdots$

Therefore, $\forall x \in A, P(x)$. □

We then fill in the blanks. In our case, A is the set of natural numbers $\mathbb{N}$, and P is being divisible by one. So we can fill in the top:

Theorem. *Let A be the collection of natural numbers $\mathbb{N}$, and let P be the property of being divisible by one. $\forall x \in A, P(x)$.*

Proof. By structural induction.

- Base case 1. ______.
- Base case 2. ______.
- $\vdots$

- Constructor 1. ______.
- Constructor 2. ______.
- $\vdots$

Therefore, $\forall x \in A, P(x)$. □

We only have one base case and one constructor, so we can cut out the extras:

Theorem. *Let A be the collection of natural numbers $\mathbb{N}$, and let P be the property of being divisible by one. $\forall x \in A, P(x)$.*

Proof. By structural induction.

- Base case. ______.
- Constructor. ______.

Therefore, $\forall x \in A, P(x)$. □

Next, we can fill in the base case:

Theorem. *Let A be the collection of natural numbers $\mathbb{N}$, and let P be the property of being divisible by one. $\forall x \in A, P(x)$.*

Proof. By structural induction.

- Base case. 0 is divisible by 1, since $\frac{0}{1} = 0$.
- Constructor. ______.

Therefore, $\forall x \in A, P(x)$. □

And finally, we can fill in the constructor:

Theorem. *Let A be the collection of natural numbers $\mathbb{N}$, and let P be the property of being divisible by one. $\forall x \in A, P(x)$.*

Proof. By structural induction.

- Base case. 0 is divisible by 1, since $\frac{0}{1} = 0$.

- Constructor. Let k by an arbitrary element from $\mathbb{N}$. The successor of k is $S(k)$, which is $k+1$. Both k and $k+1$ are divisible by one: k is divisible by one, since $\frac{k}{1} = k$, and $k+1$ is divisible by one, since $\frac{k+1}{1} = k+1$.

Therefore, $\forall x \in A, P(x)$. $\square$

34.6 Examples from λ

Proofs by structural induction have a definite structure: they should mimic exactly the recursive definition of the collection you are trying to prove. For each base case in the recursive definition, you should have a line in your proof. And for each constructor in the recursive definition, you should have a line in your proof.

Almost all of the definitions we have seen so far for λ and $\lambda \rightarrow$ have been recursive definitions. For example:

- Λ — the terms of the untyped λ. The base case is any variable from V. There are two constructors: application, and abstraction.
- FV — the free variables of λ. The base case is a variable term: the free variables is just the variable in the term. Then there are two constructors: for application you find the free variables of the two sub-terms, and for abstraction you find the free variables of the body minus the binding variable.
- $\mathbb{T}$ — type variables for $\lambda \rightarrow$. The base case is any variable from $\mathbb{V}$. There is one constructor: build an arrow type by taking two other items in $\mathbb{T}$ and joining them with an arrow.
- $\Lambda_{\mathbb{T}}$ — the pretyped terms of $\lambda \rightarrow$. The base case is any variable from V. Then there are two constructors: application, and (typed) abstraction.
- $\mathbb{J}$ — the set of all judgments. There is no base case for $\mathbb{J}$, but there are three constructor cases: the variable derivation rule, the application derivation rule, and the abstraction derivation rule.
- Subterms, substition, and so on — all of these are recursive definitions.

So, to prove any properties of these, we can use structural induction.

Chapter 35

Some Helpful Definitions for $\lambda \to$

Here are a few definitions that are useful when analyzing and proving properties about $\lambda \to$.

35.1 Domain

The **domain** of a context is the list of the variables in it. This is just a list of the variable **names**. To put this list together, we take a context, and we pull out each variable name (without any type decorations).

Definition 27 (Domain)**.** *If Γ is a context of the form $x_1 : \sigma_1, \ldots, x_n : \sigma_n$, then the domain of Γ (denoted as the dom(Γ)), is the list $(x_1, \ldots, x_n)$.*

35.2 Subcontext

The **subcontext** of a context is a subsequence of that context. I use the word *subsequence* for a reason. In a sequence, the order matters. So, if a context has a list of items in a certain order, a subcontext will be a portion of that list, with the items in the very same order.

A subcontext might be a smaller portion of a context, but it can also encompass the whole portion of the context. That is, it can be identical to the context. If a subcontext is smaller than the context, we call it a **proper subcontext**.

Definition 28 (Subcontext)**.** *A context Γ^* is a subcontext of a context Γ (denoted as $\Gamma^* \subseteq \Gamma$) if all declarations in Γ^* occur in Γ in the same order. A subcontext Γ^* is a proper subcontext of Γ (denoted $\Gamma^* \subset \Gamma$) if Γ^* has fewer elements than Γ.*

For example, let Γ be $x : \sigma$, $y : \tau$, and $z : \mu$. Here are some subcontexts: $\Gamma^* = x : \sigma, y : \tau$. $\Gamma^{**} = y : \tau, z : \mu$. Each of these is a smaller portion of Γ, but the order is preserved.

Here are some examples that are not subcontexts, because they have the order wrong: $\Gamma^* = y : \tau, x : \sigma$. $\Gamma^{**} = z : \mu, y : \tau$.

35.3 Permutation

A **permutation** of a context is a copy of the context, with the declarations shuffled into a different order. So, the permutation and the context have the same elements (every element in the context is in the permutation, and every element in the permutation is in the context).

Definition 29 (Permutation)**.** *A context Γ^* is a permutation of a context Γ if all declarations in Γ^* are in Γ, and all declarations in Γ are in Γ^*.*

For example, let Γ be $x : \sigma$, $y : \tau$, and $z : \mu$. Here are some permutations. $\Gamma^* = z : \mu, x : \sigma, y : \tau$. $\Gamma^{**} = z : \mu, y : \tau, x : \sigma$. $\Gamma^{***} = x : \sigma, z : \mu, y : \tau$.

35.4 Projection

A **projection** is basically a context, filtered down so it includes only a specified set of variables. In the language of set theory, we say that a projection of a context onto a filter set is the *intersection* of the two sets. The intersection is the list of items that are in both sets. The intersection symbol is $\cap$.

Definition 30 (Projection)**.** *If Γ is a context and Φ is a set of variable names, the projection of Γ onto Φ (denoted as $\Gamma \upharpoonright \Phi$) is the intersection $\Gamma \cap \Phi$.*

For example, let Γ be $x : \sigma$, $y : \tau$, and $z : \mu$. Let the filter set Φ be u, x, and y. The projection $\Gamma \upharpoonright \Phi$ is: $x : \sigma$, $y : \tau$. The variable z is not listed in Φ, and the variable u is not in Γ. But x and y are in both, so Γ is filtered down to the set that includes just (the typed) x and y.

Chapter 36

No Untyped Variables

One important property of $\lambda \rightarrow$ judgments is that, every free variable in a judgment is declared in the context. There are no variables that occur in a judgment which are not declared in the context.

Lemma 1 (No Untyped Free Variables)**.** *Let FV be the free variables as defined before, and let "dom" denote the domain of a context as it was defined a moment ago. If $\Gamma \vdash M : \sigma$, then $FV(M) \subseteq dom(\Gamma)$.*

36.1 The Meaning of the Lemma

Look at this lemma carefully. First, it puts forward a judgment:

$$\Gamma \vdash M : \sigma \tag{36.1}$$

This judgment says that, given a context Γ, we can derive that the term M has the type σ.

Next, the lemma says: "$FV(M) \subseteq dom(\Gamma)$." Read this like so: "then the free variables of M will be a subset of the domain of Γ."

In essence, this lemma is saying that every free variable we find in M will be found in the context. What happens in the context? Variables are given types. So, this means that every variable we find in M will have a type. There can be no untyped variables in judgments.

36.2 Setting up the Proof

Assume a judgment: $\Gamma \vdash M : \sigma$. Let us call this judgment $\mathcal{J}$:

$$\mathcal{J} \equiv \Gamma \vdash M : \sigma$$

We want to prove our lemma, which says that all the free variables in M will be in the domain of the context. So the property we want to prove (which we shall call P) is this:

$$P = FV(M) \subseteq dom(\Gamma)$$

We want to prove that this holds for every element in a collection. Which collection? It is the collection that $\Gamma \vdash M : \sigma$ belongs to, namely the set of all derivable judgments $\mathbb{J}$.

So, we want to prove that, no matter which judgment $\mathcal{J}$ turns out to be, the free variables of M — which we are denoting as $FV(M)$ — will be in the domain of the context — which we are denoting as $dom(\Gamma)$.

There are no base cases for $\mathbb{J}$, but there are three constructors. The first case is when $\mathcal{J}$ is a judgment that we derive using the *var* rule. The second case is when $\mathcal{J}$ is a judgment we derive using the *appl* rule. And the third case is when $\mathcal{J}$ is a judgment we derive using the *abstr* rule.

Here is our proof template, with the initial information filled in. The lemma we are proving has been reformulated to match the structure of the template, but it means the same thing:

Lemma. *Let A be the set of all derivable judgments $\mathbb{J}$. Let P be the property that, given a derivable judgment $\mathcal{J}$ — denoted as $\Gamma \vdash M : \sigma$ — this holds: $FV(M) \subseteq dom(\Gamma)$. Then: $\forall x \in A, P(x)$.*

Proof. By structural induction.

- The *var* case. ______.
- The *appl* case. ______.
- The *abstr* case. ______.

Therefore, $\forall x \in A, P(x)$. □

To finish this proof, we need to fill in each blank. To fill in each blank, we need to show that P is preserved by each derivation rule.

Each derivation rule is a constructor: it starts with one or more judgments as premises, and it produces from those another judgment as a conclusion:

$$\frac{\text{Judgment 1 (Premise)} \qquad \text{Judgment 2 (Premise)} \qquad \ldots}{\text{New Judgment (Conclusion)}}$$

So for each rule, we need to prove that, assuming the premise judgments have the property P, then the conclusion judgment that gets produced will have the property P too.

36.3 The *var* Case

Recall that the *var* rule looks like this:

$$\text{Variable Rule (var)}\ \frac{\varnothing}{\Gamma, x : A \vdash x : A}$$

So, if $\mathcal{J}$ is the result of applying a *var* rule, then $\mathcal{J}$ is going to look like the conclusion of the rule (the statement underneath the line):

$$\mathcal{J} = \Gamma, x : A \vdash x : A$$

We must show that in this sort of judgment, the free variables of the judgment are included in the domain of the context. We can do that:

- The judgment part is $\vdash x : A$.
- The free variables of $x : A$ is just $\{x\}$. So $FV(x : A) = \{x\}$.
- The context is $\Gamma, x : A$. The domain of this will include x, as well as any other variables listed in Γ. So $dom(\Gamma, x : A) = dom(\Gamma) \cup dom(x : A)$. And of course, $dom(x : A)$ is just $\{x\}$.
- Hence, the free variables $FV(x : A)$ are indeed included in the domain of the context $dom(\Gamma, x : A)$.

Let's fill in the first case of the proof:

Lemma. *Let A be the set of all derivable judgments $\mathbb{J}$. Let P be the property that, given a derivable judgment $\mathcal{J}$ — denoted as $\Gamma \vdash M : \sigma$ — this holds: $FV(M) \subseteq dom(\Gamma)$. Then: $\forall x \in A, P(x)$.*

Proof. By structural induction.

- The *var* case. Let $\mathcal{J} = \Gamma, x : A \vdash x : A$. Then $FV(x : A) = \{x\}$, and $dom(\Gamma, x : A) = dom(\Gamma) \cup dom(x : A)$. But $dom(x : A) = \{x\}$, so $FV(x : A) = dom(\Gamma, x : A)$.
- The *appl* case. ______.
- The *abstr* case. ______.

Therefore, $\forall x \in A, P(x)$. □

36.4 The *appl* Case

Recall that the application rule looks like this (I have changed the placeholder names to avoid name conflicts):

$$\text{Application Rule (appl)}\ \frac{\Gamma \vdash L : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash LN : B}$$

What we want to show is that this derivation preserves the property P. That is, we want to show that, if we assume that this property holds for each of the premises, then after the judgment, it still holds for the conclusion.

36.4.1 P Holds for the Premises

So, let's assume that this property holds for each of the premises. The premises are two:

$$\begin{aligned} \Gamma \vdash L : A \to B \\ \Gamma \vdash N : A \end{aligned}$$

If we assume that the free variables of each premise are in the domain of the context, we have:

$$\begin{aligned} FV(L) \subseteq dom(\Gamma) \\ FV(N) \subseteq dom(\Gamma) \end{aligned}$$

In other words, the free variables of L plus the free variables of N, will be in the domain of Γ:

$$FV(L) \cup FV(N) \subseteq dom(\Gamma)$$

That is what we get when we assume that P holds for the premises of the *appl* rule.

36.4.2 P Holds for the Conclusion

Now let us show that, if P holds for the premises, P is preserved in the conclusion of the judgment too. The conclusion of a derived application looks like this:

$$\Gamma \vdash LN : B$$

Are the free variables of this judgment included in the domain of the context? Yes:

- The judgment part is $\vdash LN : B$.

- The free variables of that are going to be the free variables of each component, namely L and N — i.e., $FV(L)$, and $FV(N)$ — put together. In set theoretic notation: $FVL \cup FVN$.
- But that is exactly what we have in the premises: the free variables of L and the free variables of N.
- So the property in question is preserved.

Let's fill in the *appl* case of the proof:

Lemma. *Let A be the set of all derivable judgments $\mathbb{J}$. Let P be the property that, given a derivable judgment $\mathcal{J}$ — denoted as $\Gamma \vdash M : \sigma$ — this holds: $FV(M) \subseteq dom(\Gamma)$. Then: $\forall x \in A, P(x)$.*

Proof. By structural induction.

- The *var* case. Let $\mathcal{J} = \Gamma, x : A \vdash x : A$. Then $FV(x : A) = \{x\}$, and $dom(\Gamma, x : A) = dom(\Gamma) \cup dom(x : A)$. But $dom(x : A) = \{x\}$, so $FV(x : A) = dom(\Gamma, x : A)$.
- The *appl* case. Let $\mathcal{J} = \Gamma \vdash LN : B$. The premises of $\mathcal{J}$ are $\Gamma \vdash L : A \to B$ and $\Gamma \vdash N : A$. If we assume that P holds for both of these premises, then we get $FV(L) \subseteq dom(\Gamma)$ and $FV(N) \subseteq dom(\Gamma)$. In other words, $FV(L) \cup FV(N) \subseteq dom(\Gamma)$. P holds for $\mathcal{J}$ too, because the free variables of $LN : B$ are $FV(L)$ and $FV(N)$, which is the same as $FV(L) \subseteq FV(N)$. So in both the premises and the conclusion of an *appl* derivation, $FV(L) \cup FV(N) \subseteq dom(\Gamma)$.
- The *abstr* case. ______.

Therefore, $\forall x \in A, P(x)$. □

36.5 The *abstr* Case

Now we need to prove that P holds for the abstractions. Recall that the rule for an abstraction looks like this:

$$\text{Abstraction Rule (abstr)}\ \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B}$$

We want to show that this judgment preserves P too. To do that, we assume that it holds for the premise of the judgment, and then we show that it holds for the conclusion.

36.5.1 P Holds for the Premise

So, let's assume that the property holds true of the premise. That is, the free variables of $M : B$ — which we can denote as $FV(M : B)$ — are contained in the domain of the context — which we can denote as $dom(\Gamma, x : A)$. In other words, we have the variables in Γ, plus x:

$$FV(M : B) \subseteq dom(\Gamma) \cup dom(x : A)$$

That is what we get when we assume that P holds for the premises of the *abstr* rule.

36.5.2 P Holds for the Conclusion

Is P preserved in the conclusion of the *abstr*? Yes:

- The judgment part is $\vdash \lambda x : A.M$.
- The free variables in that will be every free variable in M, minus the binding variable x. That is: $FV(M) - \{x\}$.
- The domain of the premise's context includes $dom(\Gamma)$ plus $\{x\}$. Here we have the free variables of M without x, and that's certainly a subset of the variables of M plus x.

Let's fill in our final case:

Lemma. *Let A be the set of all derivable judgments $\mathbb{J}$. Let P be the property that, given a derivable judgment $\mathcal{J}$ — denoted as $\Gamma \vdash M : \sigma$ — this holds: $FV(M) \subseteq dom(\Gamma)$. Then: $\forall x \in A, P(x)$.*

Proof. By structural induction.

- The *var* case. Let $\mathcal{J} = \Gamma, x : A \vdash x : A$. Then $FV(x : A) = \{x\}$, and $dom(\Gamma, x : A) = dom(\Gamma) \cup dom(x : A)$. But $dom(x : A) = \{x\}$, so $FV(x : A) = dom(\Gamma, x : A)$.
- The *appl* case. Let $\mathcal{J} = \Gamma \vdash LN : B$. The premises of $\mathcal{J}$ are $\Gamma \vdash L : A \to B$ and $\Gamma \vdash N : A$. If we assume that P holds for both of these premises, then we get $FV(L) \subseteq dom(\Gamma)$ and $FV(N) \subseteq dom(\Gamma)$. In other words, $FV(L) \cup FV(N) \subseteq dom(\Gamma)$. P holds for $\mathcal{J}$ too, because the free variables of $LN : B$ are $FV(L)$ and $FV(N)$, which is the same as $FV(L) \subseteq FV(N)$. So in both the premises and the conclusion of an *appl* derivation, $FV(L) \cup FV(N) \subseteq dom(\Gamma)$.
- The *abstr* case. Let $\mathcal{J} = \Gamma \vdash \lambda x : A.M : A \to B$. The premise of $\mathcal{J}$ is $\Gamma, x : A \vdash M : B$. If we assume that P holds for that premise, we get $FV(M : B) \subseteq dom(\Gamma) \cup dom(x : A)$. P holds for $\mathcal{J}$ too, because the free variables of $\lambda x : A.M$ are $FV(M) - \{x\}$, which are a subset of

$dom(\Gamma) \cup dom(x : A)$. So in both the premise and the conclusion of an *abstr* derivation, $FV(M) \subseteq dom(\Gamma)$.

Therefore, $\forall x \in A, P(x)$. $\square$

Chapter 37

Thinning, Condensing, and Permutations

Take any derivable judgment of the form $\Gamma \vdash M : A$. There are certain alterations we can make to the context, which do not affect the derivability of $M : A$. There are three in particular: we call them thinning, condensing, and permutations. We can prove these using structural induction.

37.1 Thinning

Suppose a judgment $\mathscr{J}$ is derivable in a certain context Γ. Now suppose we add more declarations to the context, to make a larger context Γ^*. We don't delete or alter any of the original declarations in Γ. We simply add more declarations, to make Γ^*. So the original context Γ is a subcontext of the larger context Γ^*.

If that is the case, then $\mathscr{J}$ will be derivable in the larger context Γ^*, just as it was derivable in the smaller context Γ. The idea here is that, if you add more things to a context, you don't alter derivability.

When you add more declarations to a context, we call that **thinning** the context. This can be a misleading name, because you might think that a thinned context is one with elements *removed* from it. But that's not the idea here. A context is thinned when declarations are *added* to it.

The key property of thinning is that it doesn't alter derivability. Let us prove this.

37.2 Setting Up the Proof

The property P we want to prove is this: if we have $\Gamma \subseteq \Gamma^*$, then: if $\Gamma \vdash M : \sigma$ is derivable, then so is $\Gamma^* \vdash M : \sigma$. Let us state this as a lemma:

Lemma 2 (Thinning). *Let Γ and Γ^* be contexts such that $\Gamma \subseteq \Gamma^*$. If $\Gamma \vdash M : \sigma$, then also $\Gamma^* \vdash M : \sigma$.*

What collection does this hold for? This is a lemma about judgments, so it holds for all derivable judgments $\mathbb{J}$.

Let us start by filling in the first parts of our proof template. As before, I have restated the lemma in what follows to make it more explicit how it fits into the proof.

Lemma. *Let A be the set of all derivable judgments $\mathbb{J}$. Let P be the property that, if Γ and Γ^* are contexts such that $\Gamma \subseteq \Gamma^*$, if $\Gamma \vdash M : \sigma$ is derivable, so also is $\Gamma^* \vdash M : \sigma$. Then: $\forall x \in A, P(x)$.*

Proof. By structural induction.

- The *var* case. ______.
- The *appl* case. ______.
- The *abstr* case. ______.

Therefore, $\forall x \in A, P(x)$. □

37.3 The *var* Case

Assume that a judgment is derived by applying the *var* rule. Does P hold? Yes:

- An instance of the *var* rule looks like this: $\Delta, x : \sigma \vdash M : \sigma$.
- The context is: $\Gamma = \Delta, x : \sigma$.
- Now expand the context by adding more variable declarations: $\Gamma^* = \Delta, x : \sigma, y : \tau$.
- Γ is a subcontext of Γ^*: $\Gamma \subseteq \Gamma^*$.
- The judgment is still valid by the *var* rule, with the expanded context: $\Delta, x : \sigma, y : \tau \vdash M : \sigma$.

Lemma. *Let A be the set of all derivable judgments $\mathbb{J}$. Let P be the property that, if Γ and Γ^* are contexts such that $\Gamma \subseteq \Gamma^*$, if $\Gamma \vdash M : \sigma$ is derivable, so also is $\Gamma^* \vdash M : \sigma$. Then: $\forall x \in A, P(x)$.*

Proof. By structural induction.

- The *var* case. Let $\Gamma = \Delta, x : \sigma$, and let $\Gamma^* = \Delta, x : \sigma, y : \tau$. If $\Gamma \vdash M : \sigma$, then $\Gamma^* \vdash M : \sigma$, and $\Gamma \subseteq \Gamma^*$.

- The *appl* case. ______.
- The *abstr* case. ______.

Therefore, $\forall x \in A, P(x)$. $\square$

37.4 The *appl* Case

Assume a judgment is derived by the *appl* rule. So we start with two judgments $\Gamma \vdash L : A \to B$ and $\Gamma \vdash N : A$, and from that we derive $\Gamma \vdash LN : B$. Does P hold for this? Yes:

- If $\Gamma \vdash L : A \to B$ and $\Gamma \vdash N : A$ are both derivable, then so is $\Gamma \vdash LN : B$, by applying the *appl* rule.
- Expand Γ by adding more variable declarations: $\Gamma^* = \Gamma, x : A, y : B$. So $\Gamma \subseteq \Gamma^*$.
- If $\Gamma^* \vdash L : A \to B$ and $\Gamma^* \vdash N : A$ are both derivable, then so is $\Gamma^* \vdash LN : B$, by applying the *appl* rule.

Lemma. *Let A be the set of all derivable judgments $\mathbb{J}$. Let P be the property that, if Γ and Γ^* are contexts such that $\Gamma \subseteq \Gamma^*$, if $\Gamma \vdash M : \sigma$ is derivable, so also is $\Gamma^* \vdash M : \sigma$. Then: $\forall x \in A, P(x)$.*

Proof. By structural induction.

- The *var* case. Let $\Gamma = \Delta, x : \sigma$, and let $\Gamma^* = \Delta, x : \sigma, y : \tau$. If $\Gamma \vdash M : \sigma$, then $\Gamma^* \vdash M : \sigma$, and $\Gamma \subseteq \Gamma^*$.
- The *appl* case. If $\Gamma \vdash L : A \to B$ and $\Gamma \vdash N : A$ are both derivable, then so is $\Gamma \vdash LN : B$, by applying the *appl* rule. Expand Γ by adding more variable declarations: $\Gamma^* = \Gamma, x : A, y : B$. So $\Gamma \subseteq \Gamma^*$. If $\Gamma^* \vdash L : A \to B$ and $\Gamma^* \vdash N : A$ are both derivable, then so is $\Gamma^* \vdash LN : B$, by applying the *appl* rule.
- The *abstr* case. ______.

Therefore, $\forall x \in A, P(x)$. $\square$

37.5 The *abstr* Case

Assume that a judgment is derived by applying the *abstr* rule. Does P hold? Yes:

- If $\Gamma, x : \sigma \vdash M : \tau$ is derivable, then $\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau$ is derivable, by applying the *abstr* rule.
- Expand Γ by adding more variable declarations: $\Gamma^* = \Gamma, y : \tau, z : \sigma$. So $\Gamma \subseteq \Gamma^*$.
- If $\Gamma^*, x : \sigma \vdash M : \tau$ is derivable, then $\Gamma^* \vdash \lambda x : \sigma.M : \sigma \to \tau$ is derivable, by applying the *abstr* rule.

Lemma. *Let A be the set of all derivable judgments $\mathbb{J}$. Let P be the property that, if Γ and Γ^* are contexts such that $\Gamma \subseteq \Gamma^*$, if $\Gamma \vdash M : \sigma$ is derivable, so also is $\Gamma^* \vdash M : \sigma$. Then: $\forall x \in A, P(x)$.*

Proof. By structural induction.

- The *var* case. Let $\Gamma = \Delta, x : \sigma$, and let $\Gamma^* = \Delta, x : \sigma, y : \tau$. If $\Gamma \vdash M : \sigma$, then $\Gamma^* \vdash M : \sigma$, and $\Gamma \subseteq \Gamma^*$.
- The *appl* case. If $\Gamma \vdash L : A \to B$ and $\Gamma \vdash N : A$ are both derivable, then so is $\Gamma \vdash LN : B$, by applying the *appl* rule. Expand Γ by adding more variable declarations: $\Gamma^* = \Gamma, x : A, y : B$. So $\Gamma \subseteq \Gamma^*$. If $\Gamma^* \vdash L : A \to B$ and $\Gamma^* \vdash N : A$ are both derivable, then so is $\Gamma^* \vdash LN : B$, by applying the *appl* rule.
- The *abstr* case. If $\Gamma, x : \sigma \vdash M : \tau$ is derivable, then $\Gamma \vdash \lambda x : \sigma.M : \sigma \to \tau$ is derivable, by applying the *abstr* rule. Expand Γ by adding more variable declarations: $\Gamma^* = \Gamma, y : \tau, z : \sigma$. So $\Gamma \subseteq \Gamma^*$. If $\Gamma^*, x : \sigma \vdash M : \tau$ is derivable, then $\Gamma^* \vdash \lambda x : \sigma.M : \sigma \to \tau$ is derivable, by applying the *abstr* rule.

Therefore, $\forall x \in A, P(x)$. □

37.6 Condensing

Suppose you have a derivable judgment $\Gamma \vdash M : A$. We can filter the context down so it includes only the free variables that occur in M, and $M : A$ will still be derivable.

The way we filter is with *projecting*, which we defined before. So we get the free variables of M, like this: $FV(M)$. Then we project Γ onto them:

$\Gamma \restriction FV(M)$. That gives us a new, filtered down context, that includes only the free variables in M. In that new, slimmer context, M will still be derivable, because it still has (only) the variables needed for M.

Lemma 3 (Condensing)**.** *If* $\Gamma \vdash M : A$*, then also* $\Gamma \restriction FV(M) \vdash M : A$*.*

We could prove this with structural induction too, but I will not do that here. You should try it yourself.

37.7 Permutations

Suppose you have a derivable judgment $\Gamma \vdash M : A$. You can swap around the declarations in Γ to get a new, shuffled context Γ^*. We call this a *permutation* (which we defined before).

If $\Gamma \vdash M : A$ is derivable, then $M : A$ is derivable in the permuted context too. Permutation preserves derivability.

Lemma 4 (Permutations)**.** *If* $\Gamma \vdash M : A$ *and* Γ^* *is a permutation of* Γ*, then* $\Gamma^* \vdash M : A$*.*

I will not prove this either, but you should try this yourself too.

37.8 Other Properties

There are other properties of $\lambda \rightarrow$ we can prove with structural induction.

- We can prove that every derivable judgment is derived exactly and uniquely in the way defined by the derivation rules. We call this the **Generation Lemma**.
- We can prove that every subterm of a derivable term is legal.
- We can prove that a typed term can have one and only one type.

Chapter 38

Substitution

We need to modify our definition of substitution, for $\lambda \rightarrow$.

38.1 Substitution in λ

Recall the definition of substitution from our discussion of the untyped λ. If you want to take a term M, and replace every occurrence of the free variable ϕ in it with another term N, you do that recursively, according to the following rules:

- Variables: you replace the variable with N if the variable is the value you're trying to replace, otherwise the replacement has no effect.
- Applications: to do a substitution in an application term, you push the substitution to each of the terms the application is made from.
- Abstractions: you first replace the bound variable with a fresh variable, then you do the substitution in the body of the term.

Here is the full definition, repeated verbatim from before:

Definition (Substitution)**.** *Let M, N, P, and Q be placeholders that can stand for any λ term, and let ϕ, ψ, and ξ be placeholders that can stand for any symbol in the alphabet V. Then substitution is defined as the result of taking a term M, and replacing every free occurrence of ϕ in it with another term N. This is notated as $M[\phi := N]$. The replacement must proceed according to the following rules:*

- *If M is (ϕ), then $M[\phi := N] = N$.*
- *If M is (ψ) and $\psi \neq \phi$, then $M[\phi := N] = M$.*
- *If M is (PQ), then $M[\phi := N] = ((P[\phi := N])(Q[\phi := N]))$.*
- *If M is $\lambda\psi.P$, then $M[\phi := N] = \lambda\xi.(P[\psi := \xi]([\phi := N]))$, where $\xi \in Fr(M)$.*

38.2 Substitution in $\lambda \rightarrow$

We want the same definition for $\lambda \rightarrow$, but we need to introduce types. The definition can stay the same, except for the abstraction case. There, we need to put a type decoration on the bound variable. Instead of:

$$\lambda\psi.P$$

we must speak of a term like this:

$$\lambda\psi : \sigma.P$$

Here is the revised definition:

Definition 31 (Typed Substitution)**.** *Let M, N, P, and Q be placeholders that can stand for any λ term, let ϕ, ψ, and ξ be placeholders that can stand for any symbol in the alphabet V, and let σ be a placeholder that can stand for any symbol in the type variable alphabet $\mathbb{V}$. Then substitution is defined as the result of taking a term M, and replacing every free occurrence of ϕ in it with another term N. This is notated as $M[\phi := N]$. The replacement must proceed according to the following rules:*

- *If M is (ϕ), then $M[\phi := N] = N$.*
- *If M is (ψ) and $\psi \neq \phi$, then $M[\phi := N] = M$.*
- *If M is (PQ), then $M[\phi := N] = ((P[\phi := N])(Q[\phi := N]))$.*
- *If M is $\lambda\psi : \sigma.P$, then $M[\phi := N] = \lambda\xi : \sigma.(P[\psi := \xi]([\phi := N]))$, where $\xi \in Fr(M)$.*

Notice that the newly renamed bound variable ξ has the same type as ψ — namely, the type σ. This makes sense, of course. An α-conversion only replaces bound variables with fresh variable symbols. So the types should stay the same. All that should change are the variable symbols.

Chapter 39

Abstracting over Types

In the untyped λ, we can do abstraction over free variables. What this means is this: we can mark a variable as replaceable. And then we can perform computations by replacing occurrences of that variable with some other term.

In the simply typed $\lambda \to$, we can do essentially the same thing: we can mark a variable as replaceable, and then perform computations. However, the variables are typed, and the computations must have the right types too.

So what about the types themselves? Can we abstract over them? That is, can we mark a *type* as replaceable, and then perform computations where we replace the type with another type?

Yes, we can. This is called **second-order lambda calculus**. It is also called **System F**. We will simply call it $\lambda 2$.

39.1 Many Identity Functions

Consider this untyped abstraction:

$$\lambda x.x \tag{39.1}$$

This is often called the **identity function**, because whatever value you put in for x, it returns (or reduces to) that same, identical value.

Once we introduce types, as in $\lambda \to$, we can now construct many identity functions. For instance, here is the identity function for values of type σ:

$$\lambda x : \sigma.x \tag{39.2}$$

Here is one for values of type τ:

$$\lambda x : \tau.x \tag{39.3}$$

Here is one for values with an arrow type $\sigma \to \tau$:

$$\lambda x : \sigma \to \tau.x \tag{39.4}$$

We could go on and on, writing down identity functions for any number of types.

This might seem abstract, ut I have used σ and τ as placeholders that can stand for any type whatever.

Suppose σ stands for natural numbers. Then $\lambda x : \sigma.x$ is an identity function for natural numbers. If we write *nat* instead of σ, we can see this clearly:

$$\lambda x : nat.x \tag{39.5}$$

What about the identity function for boolean values, i.e., "true" or "false"? Well, booleans aren't natural numbers, so we cannot use the identity function for natural numbers. In the simply typed $\lambda \rightarrow$, we need a separate identity function, for the boolean type:

$$\lambda x : bool.x \tag{39.6}$$

39.2 A General Identity Function

We can generalize the process of abstraction, so that we can apply it to type variables too.

39.2.1 Untyped abstraction

To begin, observe what happens when we abstract over variables in untyped λ. We start with a term, say the free variable x:

$$x \tag{39.7}$$

We then select a free variable from the expression, say x, and we mark it as replaceable. We do that with the binding expression λx, which we put up front:

$$\lambda x.x \tag{39.8}$$

We can reduce this expression by replacing x with whatever value we want. Suppose we want to replace x with y:

$$(\lambda x.x)y \tag{39.9}$$

When we do the β-reduction, the x is replaced with y:

$$y \tag{39.10}$$

39.2.2 Simply typed abstraction

Now add types. We start with a term, say $x : \sigma$:

$$x : \sigma \tag{39.11}$$

Then we select a free variable from the expression, say $x : \sigma$, and we mark it as replaceable. We use a binding expression again, but this time it is typed:

$$\lambda x : \sigma.x \tag{39.12}$$

According to the *abstr* typing rule, this whole expression has type $\sigma \to \sigma$, so we can write it like this:

$$(\lambda x : \sigma.x) : \sigma \to \sigma \tag{39.13}$$

We can reduce this expression by replacing x with whatever other value we like, provided that it is of type σ. Suppose we want to replace x with $y : \sigma$:

$$((\lambda x : \sigma.x)y \tag{39.14}$$

When we do the β-reduction, the x is replaced with y:

$$y : \sigma \tag{39.15}$$

39.2.3 Untyped Type Abstraction

Let us now repeat this process, but let us abstract over type variables.

As before, we start with a term, say the typed abstraction from before (I have added parentheses to keep the scope clear):

$$\lambda x : \sigma.(x) \tag{39.16}$$

Then we select a variable and mark it as replaceable. Here though, the variables we can select are not regular term variables. Rather, they are *type variables.*

In this term, we can see a type variable, namely σ, so let's pick that. To mark it as replaceable, we use a binding expression: $\lambda\sigma$. So, let's stick that onto the front, just as we do any other binding expression:

$$\lambda\sigma.(\lambda x : \sigma.(x)) \tag{39.17}$$

Now, we can reduce this expression by replacing σ with whatever value we want. Suppose we want to replace σ with τ:

$$(\lambda\sigma.(\lambda x : \sigma.(x)))\tau \tag{39.18}$$

When we do the β-reduction, the σ is replaced with τ:

$$\lambda x : \tau.(x) \tag{39.19}$$

39.2.4 Typed Type Abstraction

In the last abstraction, we abstracted over type variables. But we did that in an untyped way. That is, we did not restrict the type of value you could replace σ with.

Consequently, it would be completely legal to try and replace σ with something that isn't a type variable like τ. For instance, we could try to replace σ with an application term (yz):

$$(\lambda\sigma.(\lambda x : \sigma.(x)))(yz) \tag{39.20}$$

If we were to do the β-reduction, the σ would be replaced with (xy), which would yield:

$$\lambda x : (yz).(x) \tag{39.21}$$

That's a nonsense expression. It is not at all clear what it even means, or how it could be reduced.

So let's use types for the type variables too, so that we can say that only certain types of values can be put into the place of the type variables.

For the time being, let us say that there is a single type that all type variables from $\mathbb{V}$ are inhabitants of. Let us denote this type of all types with an asterisk, $*$. Think of this as a shorthand symbol that stands for the type $\mathbb{V}$.

Now we can stipulate in our binding expression $\lambda\sigma$ that the type of σ must be $*$. That is, the binding expression can be: $\lambda\sigma : *$. The whole abstraction now looks like this:

$$\lambda\sigma : *.(\lambda x : \sigma.(x)) \tag{39.22}$$

Now we have an abstraction over the type variables that will always be sensible. For according to this, you can only replace σ with another type variable (some other member of the type $*$).

39.2.5 The General Function

That leaves us with a general identity function:

$$\lambda\sigma : *.(\lambda x : \sigma.(x)) \tag{39.23}$$

We can use this function to generate any of the typed identity functions we discussed above. We could replace σ with any type variable from $\mathbb{V}$, to get any type-specific identity function we need.

39.2.6 Second Order Lambda

The technique we just discussed is called **type abstraction**, because we use it to abstract over types. It is also called **second order lambda abstraction**.

It is called second-order because it is a higher level than the abstraction we did before. The first level of abstraction is abstracting over regular term variables. So λ and $\lambda\rightarrow$ have first-order lambda abstraction.

The second level of abstraction is abstracting over type variables. We will formalize the ideas we discussed here in what follows, and as we noted earlier, we will call this system $\lambda 2$. The $\lambda 2$ system has second-order lambda abstraction.

Chapter 40

Π-Types

In $\lambda 2$, we can form abstractions over type variables. But to do this correctly, we need to introduce a new kind of type decoration, called a Π-type.

40.1 First Order Abstraction

Consider this abstraction:

$$\lambda x : \alpha . x \tag{40.1}$$

In this expression, x is marked as a replaceable symbol. So, we can put any other value (of type α) in the place of x, and the result will be that same value (with type α).

What is the type of this expression? It is $\alpha \to \alpha$. It takes an input of type α, and it produces an output of type α.

40.2 Second Order Types

Suppose we take the same expression from above:

$$\lambda x : \alpha . x \tag{40.2}$$

But suppose this time around we mark α as replaceable:

$$\lambda \alpha : *. (\lambda x : \alpha . (x)) \tag{40.3}$$

Now we have an expression that lets us replace α with other types. For instance, we could replace α with β. That would give us this:

$$\lambda x : \beta . (x) \tag{40.4}$$

If we were to reduce that expression, we'd get a result with type β.

Alternatively, we could replace α with γ. That would give us this:

$$\lambda x : \gamma.(x) \tag{40.5}$$

If we were to reduce that expression, we'd get a result with type γ.

What is the type of the original expression? Here is the expression:

$$\lambda\alpha : *.(\lambda x : \alpha.(x)) \tag{40.6}$$

Can it be anything like $\alpha \to \alpha$? No, it cannot, because the types do not stick as αs. Rather, the types change to βs or γs, depending on what we replace α with.

To say anything sensible about the type, we have to say something more like this: "The input/output types are not static, unchanging types. Rather, they are computable. To compute them, replace α in the type decorations that follow with another type."

40.3 Second Order Binding

What we have here is something that looks very much like binding variables, except in the type decorations.

- In first order binding, you select a *variable* and mark it as replaceable.
- In second order binding, you do something very similar, except you select a *type variable* and mark it as replaceable.

But, there is an important difference:

- In first order binding, the replacements are done in the *body* of the bound expressions.
- In second order binding, the replacements are done in the *type decorations* of the bound expressions.

So it is as if we have a parallel kind of binding/replacement that happens in the realm of type decorations, rather than in the body of the expressions themselves.

40.4 Constructing Π-Types

A typed term has this form:

$$M : A \tag{40.7}$$

M is a term, and A is the type decoration.

When we add a type abstraction, we select a type variable and mark it as replaceable. Suppose we pick, say, α, to replace. We would mark that like this:

$$\lambda\alpha : *.(M : A) \tag{40.8}$$

This says we can replace every occurrence of α that occurs in the type decorations A for M.

What is the type of this?

$$(\lambda\alpha : *.(M : A)) : \quad ?? \tag{40.9}$$

It is a Π-type. To construct it, we put Π and the binding variable $\alpha : *$ up front, followed by a dot and A. Like this:

$$(\lambda\alpha : *.(M : A)) : \Pi\alpha : *.A \tag{40.10}$$

The type is $\Pi\alpha : *.A$, which says: "the type is the type you get when you replace every α that occurs in A."

40.5 The Format of Type Binding

We call this second-order binding Π **binding**, or **type binding**. For this, we introduce a new binding symbol: Π (the capital Greek letter "Pi"). The format is parallel to λ binding.

Lambda binding looks like this:

$$(\underbrace{\lambda \overbrace{x : \alpha}^{\text{Binding variable}}}_{\text{Binding expression}} \quad . \quad (\underbrace{M}_{\text{Replace in body}} : A) \quad)$$

The binding variable x is the one you can replace, and the place it gets replaced is in the body of the expression, M.

Π binding is parallel, but the binding symbol is Π (rather than λ), the variable you can replace is a type variable α (rather than a regular variable x), and the replacement happens in the type decoration, A (rather than the body of the bound expression, M).

$$(\underbrace{\Pi \quad \overbrace{\alpha : *}^{\text{Binding variable}}}_{\text{Binding expression}} \quad . \qquad (M \; : \; \underbrace{A}_{\text{Replace in type decoration}}) \quad)$$

Of course, Π binding occurs only in the type decoration, not in the terms (whereas λ binding happens in the terms).

These special type decorations that contain a Π binding are called **Π-types**, or **second-order types**, or sometimes **product types**.

40.6 Π-Type Example

Suppose we have this term:

$$\lambda x : \sigma.(x) \tag{40.11}$$

The type of this term is $\sigma \to \sigma$:

$$(\lambda x : \sigma.(x)) : \sigma \to \sigma \tag{40.12}$$

Suppose we want to mark σ as replaceable. We would do that by writing this:

$$\lambda \sigma : *.((\lambda x : \sigma.(x)) : \sigma \to \sigma) \tag{40.13}$$

What is the type of this whole expression? To construct it, we put Π and $\sigma : *$ up front, followed by a dot, and then A.

What is A? It is the type of the whole term governed by the $\lambda\sigma : *$ binding expression. In this case, the whole governed term is the identity function, which has the type $\sigma \to \sigma$. So A is $\sigma \to \sigma$:

$$(\lambda \sigma : *.((\lambda x : \sigma.(x)) : \sigma \to \sigma)) : \Pi\sigma : *.(\sigma \to \sigma) \tag{40.14}$$

Chapter 41

$\lambda 2$ Types

In $\lambda 2$, we need to expand our definition of types, so that we can include Π-types along with the others. For that, we will define a new set of type variables for $\lambda 2$, which we will call $\mathbb{T}2$.

41.1 Type Variables

In $\lambda 2$, we can still create individual type variables in just the same way as $\lambda \to$: select a symbol from the $\mathbb{V}$ alphabet and wrap it in parentheses.

Recall that the $\mathbb{V}$ alphabet contains lowercase Greek letters, with subscripts if more are needed:

$$\alpha, \beta, \gamma, \delta, \ldots \beta_1, \beta_2, \ldots \tag{41.1}$$

So, to form an individual type variable in $\lambda 2$, you simply select one of these symbols, wrap it in parentheses, and you're done. Here are some examples:

$$(\alpha) \tag{41.2}$$
$$(\tau) \tag{41.3}$$
$$(\sigma_2) \tag{41.4}$$

To put this more formally: any symbol from $\mathbb{V}$ is a valid symbol in $\mathbb{T}2$. The first bunch of symbols in $\mathbb{T}2$ are all the symbols from $\mathbb{V}$. If we let α be a placeholder that can stand for any symbol in $\mathbb{V}$, then:

$$\text{if } \alpha \in \mathbb{V}, \text{ then } (\alpha) \in \mathbb{T}2 \tag{41.5}$$

Of course, we can drop the parentheses if no ambiguity comes of it, and with individual type variables, we rarely need parentheses.

(Note that, if we write the type variables without parentheses, they are indistinguishable from $\mathbb{V}$ symbols. But this rarely leads to confusion, since the context usually tells us if we are talking about symbols in the alphabet $\mathbb{V}$, or type variables in $\mathbb{T}2$.)

41.2 Arrow Types

The second bunch of symbols in $\mathbb{T}2$ are arrow types. To form an arrow type, you take any two types from $\mathbb{T}2$, you join them with an arrow, and you wrap them in parentheses.

For example, we could take α (which is in $\mathbb{T}2$, and make an arrow type from α like this:

$$(\alpha \to \alpha) \tag{41.6}$$

Or we could take σ and τ, and make an arrow type from them:

$$(\sigma \to \tau) \tag{41.7}$$

To put this more formally: any two types in $\mathbb{T}2$ combined with an arrow are valid types in $\mathbb{T}2$. If we let A and B be placeholders than can stand for any type in $\mathbb{T}2$, then:

$$\text{if } A, B \in \mathbb{T}2, \text{ then } (A \to B) \in \mathbb{T}2 \tag{41.8}$$

Note that this definition says you can take *any* types from $\mathbb{T}2$ to form an arrow type. So, you can form arrows from individual type variables in $\mathbb{T}2$, but you could also form arrows by taking other arrows from $\mathbb{T}2$.

For instance, if $\alpha \to \alpha$ and β are in $\mathbb{T}2$, then this is a valid arrow:

$$((\alpha \to \alpha) \to \beta) \tag{41.9}$$

Here are other valid examples:

$$(\alpha \to (\sigma \to \tau)) \tag{41.10}$$
$$((\sigma \to \sigma) \to (\sigma \to \tau)) \tag{41.11}$$
$$((\sigma) \to ((\sigma \to \tau) \to (\tau))) \tag{41.12}$$

Note especially that in the last example, the type on the right side of the arrow is itself an arrow type, and the type on the left side of *that* arrow is also an arrow type. So, you can compose more and more complex arrows by forming them from more and more complex types in $\mathbb{T}2$.
We can drop parentheses if it leaves us with no ambiguity. Usually we can drop the outermost parentheses, but not the other ones. For instance, take this arrow type:

$$(\alpha \to (\alpha \to \beta)) \tag{41.13}$$

We can drop the outer parentheses, to get this:

$$\alpha \to (\alpha \to \beta) \tag{41.14}$$

But what if we drop the next set of parethenses? That gives us this:

$$\alpha \to \alpha \to \beta \tag{41.15}$$

This is not allowed, since it is not clear if that means this:

$$(\alpha \to \alpha) \to \beta \tag{41.16}$$

Or this:

$$\alpha \to (\alpha \to \beta) \tag{41.17}$$

Many authors will establish a convention so that they don't have to write so many parentheses. As always, it is a good idea to find the part of the text where the author stipulates their parentheses conventions.

41.3 Π-Types

The third bunch of symbols in $\mathbb{T}2$ are Π-types. To form one of these, you take any symbol — call it α — from the alphabet $\mathbb{V}$, then you take any type — call it A — from $\mathbb{T}2$, and finally you put them together like this:

$$(\Pi\alpha.A) \tag{41.18}$$

Two points to notice. First, α must be an individual type variable selected from $\mathbb{V}$. It cannot be a complex type from $\mathbb{T}2$.

Second, A must be a type selected from $\mathbb{T}2$. And this one can be a complex type. So, it might be a simple type variable. Like this:

$$(\Pi\alpha.\alpha) \tag{41.19}$$

But it might be that A is a complex type, like an arrow. For instance:

$$(\Pi\alpha.(\alpha \to \beta)) \tag{41.20}$$

Or this:

$$(\Pi\alpha.(\alpha \to (\alpha \to \beta))) \tag{41.21}$$

A can even be another Π-type from $\mathbb{T}2$. For example, this:

$$(\Pi\alpha.(\Pi\beta.(\beta))) \tag{41.22}$$

It could be a nested Π-type::

$$(\Pi\alpha.(\Pi\beta.(\Pi\gamma.(\ldots)))) \tag{41.23}$$

To put this more formally, every Π-type constructed from a symbol in $\mathbb{V}$ and another type in $\mathbb{T}2$ is a valid $\mathbb{T}2$ type. If we let α stand for any symbol in $\mathbb{V}$ and if we let A stand for any type in $\mathbb{T}2$, then:

$$\text{if } \alpha \in \mathbb{V} \text{ and } A \in \mathbb{T}2, \text{ then } (\Pi\alpha.(A)) \in \mathbb{T}2 \tag{41.24}$$

Parentheses can of course be dropped if it leads to no ambiguity, and here we just need to be careful when A is a complex type.

41.4 Arrow Types with Π-Types

It is worth mentioning again that arrow types can be formed by taking *any* two types from $\mathbb{T}2$ and joining them with an arrow. That includes Π-types too. You can form an arrow type with a Π-type on either side of the arrow. Here are some examples:

$$(\Pi\alpha.(\alpha)) \to (\alpha) \tag{41.25}$$

$$(\Pi\beta.(\alpha)) \to (\beta) \tag{41.26}$$

41.5 Definition of $\mathbb{T}2$ Types

There are three ways to form a type in $\mathbb{T}2$:

- A type variable — form this one by taking a symbol from $\mathbb{V}$.
- An arrow type — form this one by joining any two other types from $\mathbb{T}2$ with an arrow.
- A Π-type — form this one by taking a symbol from $\mathbb{V}$ and a type from $\mathbb{T}2$, and putting them together with a Π binder.

We can formulate a formal definition for $\mathbb{T}2$ by putting those cases together.

Definition 32 ($\mathbb{T}2$ Types)**.** *Let $\mathbb{V}$ be the alphabet of type symbols as defined before. Then $\mathbb{T}2$ is the smallest set of types that satisfy the following conditions, where α is a placeholder that can stand for any symbol from $\mathbb{V}$, and A, B are placeholders that can stand for any types from $\mathbb{T}2$:*

- *If $\alpha \in \mathbb{V}$, then $(\alpha) \in \mathbb{T}2$.*
- *If $A, B \in \mathbb{T}2$, then $(A \to B) \in \mathbb{T}2$.*
- *If $\alpha \in \mathbb{V}$ and $A \in \mathbb{T}2$, then $(\Pi\alpha.A) \in \mathbb{T}2$.*

Parentheses may be dropped if it leads to no ambiguity.

Chapter 42

$\Lambda 2$ Pretyped Terms

In $\lambda 2$, we need to include new types of pretyped terms. The reason is that we now allow second-order abstractions, and with that, we also allow second-order applications. In this chapter, we will define a new set of pretyped terms for $\lambda 2$, which we will call $\Lambda 2$.

42.1 Variable Terms

The first bunch of terms in $\Lambda 2$ are variable terms. We form these the same way that we did before: take a symbol from the variable alphabet V, and wrap it in parentheses.

Recall that the variable alphabet V is the set of individual letters, $a, b, \ldots x, y, \ldots$, with subscripts if we need to make more:

$$V = \{a, b, c, \ldots, x, y, \ldots, x_1, x_2, \ldots\} \tag{42.1}$$

So, to form a variable term in $\Lambda 2$, take any one of those symbols, and wrap it in parentheses. Here are some examples:

$$(a) \tag{42.2}$$
$$(b) \tag{42.3}$$
$$(x) \tag{42.4}$$
$$(y) \tag{42.5}$$
$$(x_2) \tag{42.6}$$

To put this more formally: any symbol from V, wrapped in parentheses, is a valid term in $\Lambda 2$. If we let x stand for any symbol from V, then:

$$\text{if } x \in V, \text{ then } (x) \in \Lambda 2 \tag{42.7}$$

Of course, we can drop parentheses if it leads to no ambiguity.

42.2 First Order Applications

The second bunch of terms in Λ2 are first-order application terms. We form these much as we did before. We take any two terms from Λ2, and we put them next to each other, wrapped in parentheses. For instance:

$$((x)(y)) \tag{42.8}$$
$$((x)(x)) \tag{42.9}$$
$$((a)(b)) \tag{42.10}$$

Also, the left and right hand terms need not be simple variable terms. They can be *any* term from Λ2, including complex terms like other first-order applications or abstractions.

Here are some more examples, where the left and/or right hand terms of the application are themselves complex:

$$((x)(xy)) \tag{42.11}$$
$$((xx)(y(yz))) \tag{42.12}$$
$$((\lambda b.(a))(b)) \tag{42.13}$$

To put this more formally: any application formed by putting two terms from Λ2 together and wrapping them in parentheses is a valid term in Λ2. If we let M and N stand for any two terms from Λ2, then:

$$\text{if } M, N \in \Lambda 2 \text{ then } (MN) \in \Lambda 2 \tag{42.14}$$

Of course, parentheses can be omitted if it leads to no ambiguity. In this case, care must be taken if the left and/or right hand sides of the application are themselves complex.

42.3 First Order Abstractions

The third bunch of terms in Λ2 are first-order abstractions. We can form these terms much as we did before: take any symbol — say, x — from V, take a type — call it A — from $\mathbb{T}2$, and take any term — call it M — from Λ2, and put them together to form $(\lambda x : A.(M))$. Here are some examples:

$$(\lambda x : \sigma.(x)) \tag{42.15}$$
$$(\lambda x : \tau.(xy)) \tag{42.16}$$
$$(\lambda x : \sigma.(\lambda y : \tau.(x))) \tag{42.17}$$

Note that the body of these expressions can themselves be complex terms like applications or abstractions.

We can put this more formally: any expression of the form $(\lambda x : A.(M))$ is a valid term in $\Lambda 2$, where x can be any symbol from V, A can be any type from $\mathbb{T}2$, and M can be any term from $\Lambda 2$.

$$\text{if } x \in V, A \in \mathbb{T}2, \text{ and } M \in \Lambda 2 \text{ then } (\lambda x : A.(M)) \in \Lambda 2 \tag{42.18}$$

As always, parentheses can be dropped if it leads to no ambiguity.

42.4 Second Order Abstractions

So far, the pretyped terms in $\Lambda 2$ have looked very much like the pretyped terms from $\lambda \rightarrow$. But now we will add another bunch of terms that weren't present in λ or $\lambda \rightarrow$: we will add **second-order abstraction terms**.

These are formed much like first-order abstraction terms, but instead of taking a regular variable symbol from V, you take a type variable symbol from $\mathbb{V}$.

So, to form a second-order abstraction term, you take a type variable — say, α — from $\mathbb{V}$, and a term — call it M — from $\Lambda 2$, and you put them together in this form: $(\lambda \alpha : *.(M))$. Here are some examples:

$$(\lambda \alpha : *.(xy)) \tag{42.19}$$
$$(\lambda \alpha : *.(\lambda x : \alpha.(x))) \tag{42.20}$$
$$(\lambda \sigma : *.(\lambda \tau : *.(\lambda x : \sigma.(y))) \tag{42.21}$$

Note that here too, the body of the expression — what we are calling term M — can be complex.

To put this more formally: any abstraction formed with a type variable from $\mathbb{V}$ and a term from $\Lambda 2$ is a valid term in $\Lambda 2$.

$$\text{if } \alpha \in \mathbb{V} \text{ and } M \in \Lambda 2 \text{ then } (\lambda \alpha.(M)) \in \Lambda 2 \tag{42.22}$$

Here too, parentheses can be dropped if it leaves us with no ambiguity. Care must be taken with the body of the abstraction — the term M — if it is itself a complex term.

42.5 Second Order Applications

In addition to second order abstraction terms, we also have **second order application terms**. Like first order applications, these are formed from two terms. But with second order applications, the right hand term is a type from $\mathbb{T}2$. Here are some examples:

$$((\lambda \alpha : *.(x))\beta) \tag{42.23}$$
$$((\lambda \alpha : *.(\lambda x : \alpha.(x)))\beta) \tag{42.24}$$

We use second order applications when we want to provide a type variable as a replacement for a type abstraction. So, in second order applications, the left hand term is a type abstraction (a second-order abstraction term), and the right hand term is a type (from $\mathbb{T}2$).

To put this more formally: any application you form where the first term is a term from $\Lambda 2$ and the second term is a type from $\mathbb{T}2$ is a valid term in $\Lambda 2$. If we let M stand for any term from $\Lambda 2$ and if we let A stand for any type from $\mathbb{T}2$, then:

$$\text{if } M \in \Lambda 2 \text{ and } A \in \mathbb{T}2, \text{ then } (MA) \in \Lambda 2 \tag{42.25}$$

Of course, we can drop parentheses here too, but only if it leads to no ambiguity.

42.6 Definition of $\Lambda 2$ Pretyped Terms

We have seen that there are five ways to construct pretyped $\Lambda 2$ terms:

- Variable terms — you form one of these by taking any symbol from V.
- First order application terms — you form one of these by taking any two terms — call them M and N — from $\Lambda 2$ and then you put them together to form (MN).
- First order abstraction terms — you form one of these by taking a variable – call it x — from V, a type — call it A — from $\mathbb{T}2$, and a term — call it M — from $\Lambda 2$, and then you put them together to form $\lambda x : A.(M)$.
- Second order abstraction terms — you form one of these by taking a type — say, α — from $\mathbb{V}$ and a term — call it M — from $\Lambda 2$, and you put them together to form $\lambda \alpha : *.(M)$.
- Second order application terms — you form one of these by taking a term — call it M — from $\Lambda 2$ and a type — call it A — from $\mathbb{T}2$, and then you put them together to form (MA).

We can combine all of these methods to formulate a formal definition for $\Lambda 2$ pretytped terms:

Definition 33 (Pretyped $\Lambda 2$ Terms)**.** *Let V be the alphabet of variables defined before, let $\mathbb{V}$ be the alphabet of type symbols defined before, and let $\mathbb{T}2$ be the set of types defined before. Then $\Lambda 2$ is the smallest set of terms defined by the following conditions, where x is a placeholder that stands for any symbol from V, α is a placeholder that stands for any symbol from $\mathbb{V}$, A is a placeholder that stands for any type from $\mathbb{T}2$, and M, N are placeholders that stand for any terms from $\Lambda 2$:*

- *If $x \in V$, then $(x) \in \Lambda 2$.*
- *If $M, N \in \Lambda 2$, then $(MN) \in \Lambda 2$.*

- *If* $x \in V$, $A \in \mathbb{T}2$, *and* $M \in \Lambda 2$, *then* $(\lambda : \alpha.(M)) \in \Lambda 2$.
- *If* $\alpha \in \mathbb{V}$ *and* $M \in \Lambda 2$, *then* $(\lambda \alpha : *.(M)) \in \Lambda 2$.
- *If* $M \in \Lambda 2$ *and* $A \in \mathbb{T}2$, *then* $(MA) \in \Lambda 2$.

Parentheses can be omitted if it leads to no ambiguity.

Chapter 43

$\lambda 2$ Contexts

In $\lambda 2$, we need to expand our definition of statements, declarations, and contexts, so that we can handle second-order types.

43.1 Statements

Recall from $\lambda \to$ that typing **statements** have this form:

$$M : A \tag{43.1}$$

where M is a pretyped term from $\Lambda_{\mathbb{T}}$, and A is a type from $\mathbb{T}$. We say that M is the **subject** of the statement.

In $\lambda 2$, we have similar typing statements. They have the same form:

$$M : A \tag{43.2}$$

but M is a pretyped term from $\Lambda_{\mathbb{T}2}$, and A is a type variable from $\mathbb{T}2$.

For example, this includes any term that looks like a statement from $\lambda \to$:

$$x : \sigma \tag{43.3}$$
$$y : \sigma \to \tau \tag{43.4}$$
$$\lambda x : \sigma.(x) : \sigma \to \sigma \tag{43.5}$$
$$\tag{43.6}$$

But it also includes terms that include Π-types. For instance:

$$\lambda \sigma : *.(\lambda x : \sigma.(x)) : \Pi \sigma : *.(\sigma \to \sigma) \tag{43.7}$$

So, the set of statements we can form in $\lambda 2$ is larger than the set we can form in $\lambda \to$.

There is one other type of statement that we can make in $\lambda 2$. We can state that a type variable has a type (the type of types, the asterisk type). It has this form:

$$A : * \tag{43.8}$$

where A is from $\mathbb{T}2$, and the asterisk is the type of types we discussed before. For example:

$$\alpha : * \tag{43.9}$$
$$\sigma : * \tag{43.10}$$
$$\tau : * \tag{43.11}$$

So, altogether, statements in $\lambda 2$ can be of the form $M : A$, or $A : *$. Formally, we can define statements for $\lambda 2$ like this:

Definition 34 (Statements in $\lambda 2$)**.** *The set of statements $S_{\lambda 2}$ in $\lambda 2$ is the smallest set of strings formed in either of the following two ways:*

- *If $M \in \Lambda_{\mathbb{T}2}$ and $A \in \mathbb{T}2$, then $M : A \in S_{\lambda 2}$.*
- *If $A \in \mathbb{T}2$, then $A : * \in S_{\lambda 2}$.*

If a statement $s \in S_{\lambda 2}$ has the form $M : A$, then M is the subject of the statement. Otherwise, if a statement s has the form $A : *$, then A is the subject of the statement.

43.2 Declarations

Recall from $\lambda \rightarrow$ that a **declaration** is a special kind of type statement: in particular, it is one where the subject is a variable. So, declarations in $\lambda \rightarrow$ have this form:

$$x : \sigma \tag{43.12}$$

where x is a variable from V, and σ is a type variable from $\mathbb{T}$.

In $\lambda 2$, declarations are also statements where the subject is a variable. However, in $\lambda 2$, the subject can be a regular variable from V, or it can be a type variable from $\mathbb{T}2$.

Definition 35 (Declarations in $\lambda 2$)**.** *The set of declarations $D_{\lambda 2}$ in $\lambda 2$ is the smallest subset of statements from $S_{\lambda 2}$ — that is, $D_{\lambda 2} \subseteq S_{\lambda 2}$ — such that, for each $s \in S_{\lambda 2}$:*

- *If s is of the form $M : A$, and $M \in V$, then $s \in D_{\lambda 2}$.*
- *If s is of the form $A : *$ and $A \in \mathbb{T}2$, then $s \in D_{\lambda 2}$.*

43.3 Contexts, Informally

Recall from $\lambda \rightarrow$ that type judgments have this form:

$$\Gamma \vdash M : A \tag{43.13}$$

where Γ is a context, M is a pretyped term from $\Lambda_{\mathbb{T}}$, and A is a type from $\mathbb{T}$.

The **context**, Γ, is a list of declarations — it provides all the variables (and their types) that can be used in deciding if the statement $M : A$ is correct.

In $\lambda 2$, regular variables are declared in the context too, just as they were with $\lambda \rightarrow$. So, for instance, we might have a context that includes values like this:

$$\Gamma = x : \sigma, y : \alpha \rightarrow \beta \tag{43.14}$$

But we can also declare type variables in $\lambda 2$ contexts, which have the form $A : *$. So, in addition to values like the above, we might also have values like this in a context:

$$\Gamma = \alpha : *, \beta : * \tag{43.15}$$

43.4 Declare before Use

The basic rule is this: **declare before use**. That is to say, every variable or type variable must be declared before it can be used. This is particularly important for complex types. Consider this context:

$$\Gamma = x : \sigma \rightarrow \sigma \tag{43.16}$$

This is actually an illegal context, in $\lambda 2$. Why? Because we've used σ, but we haven't declared σ.

We need to declare $\sigma : *$ before we can declare $x : \sigma \rightarrow \sigma$. Here is the correct context:

$$\Gamma = \sigma : *, x : \sigma \rightarrow \sigma \tag{43.17}$$

Consider this context:

$$\Gamma = \alpha : *, y : \alpha \rightarrow \beta \tag{43.18}$$

This is an illegal context too. In it, we declare α before we use it, but we do not declare β before we use it. Here it is, corrected:

$$\Gamma = \alpha : *, \beta : *, y : \alpha \rightarrow \beta \tag{43.19}$$

43.5 All the Cases

Think about all the cases we need to define a context. First, a context can be totally empty. There can be nothing in it. We can symbolize that as the empty set:

$$\Gamma = \varnothing \tag{43.20}$$

Second, we can add to a context an individual type variable declaration of the form $\alpha : *$. Suppose we have a context already — call it Δ:

$$\Gamma = \Delta \tag{43.21}$$

Now suppose that α is a symbol in our type variable alphabet V, and α is not already declared in Δ. Then we can add $\alpha : *$ to the context:

$$\Gamma = \Delta, \alpha : * \tag{43.22}$$

Third, we can add declarations of regular variables from V. For instance, suppose we want to declare that the variable x has the type α. To do that, we first must declare α as a type:

$$\Gamma = \Delta, \alpha : * \tag{43.23}$$

And then, we can add $x : \alpha$ to the context:

$$\Gamma = \Delta, \alpha : *, x : \alpha \tag{43.24}$$

The key is that we declared α as $\alpha : *$ before we used it in the declaration $x : \alpha$.

Here is another example. Suppose we want to declare the variable y has the type $\alpha \to \beta$. To do that, we need to first declare α and β:

$$\Gamma = \Delta, \alpha : *, \beta : * \tag{43.25}$$

Then we can add $y : \alpha \to \beta$ to the context:

$$\Gamma = \Delta, \alpha : *, \beta : *, y : \alpha \to \beta \tag{43.26}$$

43.6 Contexts Defined

Notice from the previous examples that we can declare types in the context, which themselves employ types that need to be declared. This means we need to define contexts recursively. Here is the definition.

Definition 36 (Contexts in λ2)**.** *A valid context in λ2 is any list Γ of declarations that satisfy the following rules:*

- *If there are no declarations, $\varnothing$ is a valid context Γ.*

- *If Δ is a valid $\lambda 2$ context, α is a symbol in $\mathbb{V}$, and a is not declared in Δ, then $\Delta, \alpha : *$ is a valid context Γ.*
- *If Δ is a valid $\lambda 2$ context, x is a symbol in V, ρ is a type in $\mathbb{T}2$, and all type variables $\sigma_1, \sigma_2, \ldots$ that occur in ρ are declared as Δ, $\sigma_1 : *$, $\sigma_2 : *$, $\ldots$, then $\Delta, \sigma_1 : *$, $\sigma_2 : *$, $\ldots$, $x : \rho$ is a valid context Γ.*

43.7 Domains

Recall from $\lambda \to$ that the **domain** of a type judgment is a list of all variables, without their types. It is just a list of the *subjects* of all declarations in the context.

In $\lambda 2$, a domain is much the same, but it includes both regular variables like x and y, as well as type variables like α or σ. Here are the different cases.

- If $\Gamma = \varnothing$, then the domain of Γ is just $\varnothing$.
- If $\Gamma = \Delta, \alpha : *$, then the domain of Γ is the domain of Δ, plus α. For example, if Δ is nothing, then the domain of $\Delta, \alpha : *$ is just $\{\alpha\}$. If Δ already has some variables declared in it, for instance, x and β, then the domain of $\Delta, \alpha : *$ is $\{x, \beta, \alpha\}$.
- If $\Gamma = \Delta, x : \rho$, then the domain of Γ is the domain of Δ, plus x. If Γ is empty, then the domain of $\Delta, x : \rho$ is just $\{x\}$. If Δ already has some variables declared in it, for instance y, σ, and β, then the domain of $\Delta, x : \rho$ is $\{y, \sigma, \beta, x\}$.

More formally, here is a definition. We will denote the domain of a context Γ as $dom(\Gamma)$.

Definition 37 (Domains in $\lambda 2$)**.** *The domain of a context Γ — denoted as $dom(\Gamma)$ is the set that satisfies the following conditions:*

- *If $\Gamma = \varnothing$, then $dom(\Gamma) = \varnothing$.*
- *If $\Gamma = \Delta, \alpha : *$, then $dom(\Gamma) = dom(\Delta) \cup \{\alpha\}$.*
- *If $\Gamma = \Delta, x : \rho$, then $dom(\Gamma) = dom(\Delta) \cup \{x\}$.*

Chapter 44

$\lambda 2$ Derivation Rules

Let us discuss the derivation rules we use to build type judgments in $\lambda 2$.

44.1 The *var* rule

First, we have the *var* rule. This is just like the *var* rule for $\lambda \rightarrow$. It says: if you have a context with a variable of the form $x : \alpha$ declared in it, then you can infer that the judgment $x : \alpha$ is correct. Here it is:

$$\text{Var}\ \frac{\Gamma, x : \alpha}{\Gamma, x : \alpha \vdash x : \alpha}$$

We could also write this without any premises, like so:

$$\text{Var}\ \frac{}{\Gamma \vdash x : \alpha}$$

But we'd have to specify that there are some extra conditions that must be present:

- Γ must be a valid $\lambda 2$ context.
- $x : \alpha \in \Gamma$.

44.1.1 Example

Here is a valid derivation, using the *var* rule:

```
1  | α : ∗
   |------
   | 2  | x : α
   |    |--------
   |    | 3   x : α    var, 2
   |
```

Notice that we first declared $\alpha : *$ in the outermost context, and then we declared $x : \alpha$. What would happen if we just declared $x : \alpha$? The derivation would look like this:

```
1 | x : α
  |--------
  | 2   x : α   var, 1
```

Is that a valid derivation in $\lambda 2$? No, it is not, because the context is not a valid $\lambda 2$ context.

44.2 The *form* rule

In $\lambda 2$, we also have another type of variable declaration: we can declare *type variables*, with the format $A : *$. In $\lambda 2$, we can derive valid type judgments about these too. So, if $A : *$ is a properly declared type in a context Γ, then this is a valid judgment in $\lambda 2$:

$$\Gamma \vdash A : * \tag{44.1}$$

The only special requirement here is that any type variables used in A must be declared in the context Γ, as we discussed in the last chapter. So, for all $\sigma_1, \sigma_2, \ldots \in A$, if we have a context that looks like this:

$$\Gamma \cup \{\sigma_1 : *, \sigma_2 : *, \ldots \in A\} \tag{44.2}$$

Then we can formulate a type judgment that looks like this:

$$\Gamma \cup \{\sigma_1 : *, \sigma_2 : *, \ldots \in A\} \vdash A : * \tag{44.3}$$

This is called the **formation** rule. Let's formulate it like this:

$$\text{Form}\ \frac{\Gamma \cup \{\sigma_1 : *, \sigma_2 : *, \ldots \in A\}}{\Gamma \cup \{\sigma_1 : *, \sigma_2 : *, \ldots \in A\} \vdash A : *}$$

44.2.1 Example

Here is a valid derivation, using the *form* rule:

```
1 | α : *
  |------
  | 2 | x : α
  |   |--------
  |   | 3   α : *   form, 1
  |
```

The judgment is this: $\alpha : *$, which says that α is a type. It is derived from the outermost context on line 1, which declares $\alpha : *$.

Here is another valid derivation, using the *form* rule:

```
1 | α : *
  |------
  | 2 | β : *
  |   |------
  |   | 3 | x : α → β
  |   |   |------------
  |   |   | 4   α → β : *    form, 3
  |   |
  |
```

The judgment is this: $\alpha \to \beta : *$, which says that $\alpha \to \beta$ is a type. It is derived from the declaration on line 3. That declaration employs the types α and β, and it is valid because they are declared on lines 1 and 2.

44.3 The *abstr* rule

The *abstr* rule in $\lambda 2$ is much as it was in $\lambda \to$. Suppose you have previously derived the judgment that $M : \beta$, from a context where a variable such as $x : \alpha$ is declared:

$$\Gamma, x : \alpha \vdash M : \beta \tag{44.4}$$

We can mark $x : \alpha$ as replaceable, as an abstraction. The type of the result is then $\alpha \to \beta$:

$$\Gamma, \vdash \lambda x : \alpha.M : \alpha \to \beta \tag{44.5}$$

As a rule, we write that like this:

$$\text{Abstr}\ \frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x : \alpha.M : \alpha \to \beta}$$

44.3.1 Example

Here is an example of a valid derivation in $\lambda 2$, using the *abstr* rule:

```
1 | $\alpha : *$
  |----------
  | 2 | $\beta : *$
  |   |----------
  |   | 3 | $x : \alpha$
  |   |   |----------
  |   |   | 4 | $y : \beta$
  |   |   |   |------------------------
  |   |   |   | 5  $y : \beta$                             var, 4
  |   |   |   | 6  $\lambda x : \alpha.y : \alpha \to \beta$    abstr, 5 (and 3)
  |   |   |
  |   |
  |
```

The judgment is on line 6. It says that $\lambda x : \alpha.y$ has type $\alpha \to \beta$. That is correct, because $y : \beta$ was derived previously (on line 5), and the variable $x : \alpha$ was already declared in the context (on line 3).

44.4 The *abstr*$_2$ rule

In λ2, we can also perform type abstractions, as we saw before when we discussed Π-types. So, we need a rule to form those type judgments too. Let's call this the **abstr**$_2$ rule.

Suppose you have already derived that a term M has a type A using a context with a type variable $\alpha : *$:

$$\Gamma, \alpha : * \vdash M : A \tag{44.6}$$

We can mark $\alpha : *$ as replaceable, as a type abstraction. The type of the result is then $\Pi\alpha : *.A$:

$$\Gamma \vdash M : \Pi\alpha : *.A \tag{44.7}$$

As a rule, we write that like this:

$$\text{Abstr}_2\ \frac{\Gamma, \alpha : * \vdash M : A}{\Gamma \vdash M : \Pi\alpha : *.A}$$

44.4.1 Example

Here is an example of a valid derivation in λ2, using the *abstr*$_2$ rule:

1 $\alpha : *$

  2 $\beta : *$

    3 $x : \alpha$

      4 $y : \beta$

        5 $y : \beta$ *var*, 4

        6 $y : \Pi\alpha : *.\beta$ $abstr_2$, 5 (and 1)

The judgment is again on line 6. It says that y has type $\Pi alpha : *.\beta$. That is correct, because $y : \beta$ was derived previously (on line 5), and the type variable $\alpha : *$ was already declared in the context (on line 1).

44.5 The *appl* rule

In $\lambda 2$, the *appl* rule is much the same as it was in $\lambda \rightarrow$. Suppose you have previously derived two judgments. One says that a term M has the type $\alpha \rightarrow \beta$:

$$\Gamma \vdash M : \alpha \rightarrow \beta \tag{44.8}$$

The other says a term N has the type α:

$$\Gamma \vdash N : \alpha \tag{44.9}$$

Since the first term M is an arrow type from α to β, and the second term has the right kind of input type (namely, α), you can put those two terms together to form an application, which has the type β:

$$\Gamma \vdash MN : \beta \tag{44.10}$$

We write the rule like this:

$$\text{Appl}\ \frac{\Gamma \vdash M : \alpha \rightarrow \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta}$$

44.5.1 Example

Here is an example of a valid derivation in $\lambda 2$, using the *appl* rule:

> 1 $\alpha : *$
>
>> 2 $\beta : *$
>>
>>> 3 $x : \alpha \to \beta$
>>>
>>>> 4 $y : \alpha$
>>>>
>>>> 5 $x : \alpha \to \beta$ *var*, 3
>>>> 6 $y : \alpha$ *var*, 4
>>>> 7 $xy : \beta$ *appl*, 5 and 6

The judgment is on line 7. It says that xy has type β. That is correct, because $x : \alpha \to \beta$ was derived previously (on line 5), and $y : \alpha$ was also derived previously (on line 6). Since y has the right input type for x, we can form an application.

44.6 The *appl*$_2$ rule

Applications specify what you should replace bound variables with. In $\lambda 2$, we don't only have variable abstractions. We also have type abstractions, with Π-types. So, we also need a higher order kind of application, which lets us specify what you should replace bound *type variables* with too.

Let us call this the **appl**$_2$ rule. It runs as follows. Suppose we have previously derived that a term M has a type $\Pi\alpha : *.A$:

$$\Gamma \vdash M : \Pi\alpha : *.A \tag{44.11}$$

Suppose that we have also previously derived that there is a type $B : *$:

$$\Gamma \vdash B : * \tag{44.12}$$

We can then apply $B : *$ to $M : \Pi\alpha : *.A$. After all, $\Pi\alpha : *.A$ says that the type variable $\alpha : *$ can be replaced in A with another type, and $B : *$ is another type. So, we can put the two together into an application. The resulting type will be A, with every α in it replaced by B:

$$\Gamma \vdash MB : A[\alpha := B] \tag{44.13}$$

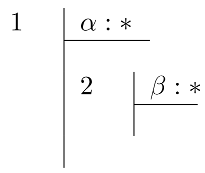
Note that the judgment has the form of an application: it consists of two terms, M and B. But in this case, the second term is a *type*, which we are denoting as B. Also, the resulting type is the original type of M, which we denoted as A, with every α in it replaced by B.

We can put this all as an inference rule, like this:

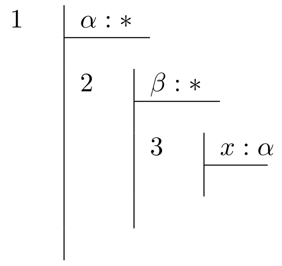
$$\text{Appl}_2\ \frac{\Gamma \vdash M : \Pi\alpha : *.A \qquad \Gamma \vdash B : *}{\Gamma \vdash MB : A[\alpha := B]}$$

44.6.1 An Example

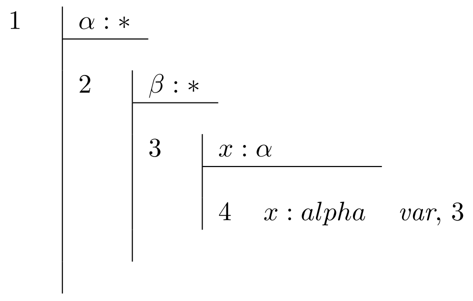
Here is an example of a valid derivation in $\lambda 2$, using the *appl*$_2$ rule. First, let us declare two type variables in our context: α and β.



Next, let us declare a variable x with type α in the context:



Next, let us use the *var* rule to derive $x : \alpha$:



And let us use the *form* rule to derive that $\beta : *$ is a type:

```
1 | α : ∗
  |------
  | 2 | β : ∗
  |   |------
  |   | 3 | x : α
  |   |   |--------------
  |   |   | 4  x : alpha     var, 3
  |   |   | 5  β : ∗         form, 2
  |   |
  |
```

Let us now take the judgment in line 4, which is $x : \alpha$, and let us mark α as replace-able. Let us say that you can replace $\alpha : *$ with another type. For that, we use the *abstr*$_2$ rule:

```
1 | α : ∗
  |------
  | 2 | β : ∗
  |   |------
  |   | 3 | x : α
  |   |   |--------------
  |   |   | 4  x : alpha     var, 3
  |   |   | 5  β : ∗         form, 2
  |   |   | 6  x : Πα : ∗.α  abstr₂, 3 (and 1)
  |   |
  |
```

Finally, let us stipulate that we will replace $\alpha : *$ with $\beta : *$. We do that with the *appl*$_2$ rule:

```
1 | α : ∗
  |------
  | 2 | β : ∗
  |   |------
  |   | 3 | x : α
  |   |   |--------------
  |   |   | 4  x : alpha     var, 3
  |   |   | 5  β : ∗         form, 2
  |   |   | 6  x : Πα : ∗.α  abstr₂, 3 (and 1)
  |   |   | 7  xβ : β        appl₂, 6 and 5
  |   |
  |
```

44.6.2 Another Example

Here is a second example. Suppose we have three type variables in context: α, β, and σ.

$$\begin{array}{r|l} 1 & \alpha : * \\ \cline{2-2} & \begin{array}{r|l} 2 & \beta : * \\ \cline{2-2} & \begin{array}{r|l} 3 & \sigma : * \\ \cline{2-2} & \end{array} \end{array} \end{array}$$

Now suppose we have in our context a variable, x, which has an arrow type from α to β:

$$\begin{array}{r|l} 1 & \alpha : * \\ \cline{2-2} & \begin{array}{r|l} 2 & \beta : * \\ \cline{2-2} & \begin{array}{r|l} 3 & \sigma : * \\ \cline{2-2} & \begin{array}{r|l} 4 & x : \alpha \to \beta \\ \cline{2-2} & \end{array} \end{array} \end{array} \end{array}$$

Let's use the *var* rule to derive that $x : \alpha \to \beta$, and then let's use the *abstr*$_2$ rule to mark $\alpha : *$ in that arrow type as replaceable:

$$\begin{array}{r|l} 1 & \alpha : * \\ \cline{2-2} & \begin{array}{r|l} 2 & \beta : * \\ \cline{2-2} & \begin{array}{r|l} 3 & \sigma : * \\ \cline{2-2} & \begin{array}{r|lll} 4 & x : \alpha \to \beta & & \\ \cline{2-4} & 5 & x : \alpha \to \beta & \textit{var}, 4 \\ & 6 & x : \Pi\alpha.(\alpha \to \beta) & \textit{abstr}_2, 5 \text{ (and 1)} \end{array} \end{array} \end{array} \end{array}$$

Next, let's use the *form* rule to derive that $\sigma : *$ is a type:

```
1 | α : *
  |------
  | 2 | β : *
  |   |------
  |   | 3 | σ : *
  |   |   |------
  |   |   | 4 | x : α → β
  |   |   |   |-----------------
  |   |   |   | 5  x : α → β          var, 4
  |   |   |   | 6  x : Πα.(α → β)     abstr₂, 5 (and 1)
  |   |   |   | 7  σ : *              form, 3
  |   |   |
  |   |
  |
```

And finally, let's use the *appl*$_2$ rule. Let's stipulate that we want to replace $\alpha : *$ in the arrow type with σ. That gives us:

```
1 | α : *
  |------
  | 2 | β : *
  |   |------
  |   | 3 | σ : *
  |   |   |------
  |   |   | 4 | x : α → β
  |   |   |   |-----------------
  |   |   |   | 5  x : α → β          var, 4
  |   |   |   | 6  x : Πα.(α → β)     abstr₂, 5 (and 1)
  |   |   |   | 7  σ : *              form, 3
  |   |   |   | 8  xσ : σ → β         appl₂, 6 and 7
  |   |   |
  |   |
  |
```

Chapter 45

Kinds

Another type of lambda calculus is called $\lambda\omega$. In $\lambda\omega$, we take the whole abstraction/application/reduction mechanism of computation that we have been applying to terms, and we apply it to types.

45.1 Type Operations in $\lambda 2$

In $\lambda 2$, we can perform some rudimentary computation on types. We can mark type variables as replaceable (abstraction), and we can replace those marked type variables with other type variables (application and reduction).

However, with $\lambda 2$, we can only abstract over terms that already have a specified type. We cannot create new types that aren't attached to any term, and then perform abstraction and application on those new types, completely apart from terms.

45.2 Type Operations in $\lambda\omega$

With $\lambda\omega$, we are free to compute any type whatsoever. We can build types from types, and build more types from those types. We can abstract over any of these types, and we can perform applications and reductions with those types, just as we can do with terms in $\lambda^{\to}$.

So, $\lambda\omega$ introduces **type operators**, which is just a fancy way to say that we can perform lambda operations on types.

45.3 Types and Kinds

Atomic types are single-symbol types from the type alphabet $\mathbb{V}$. For example:

$$\alpha, \beta, \sigma, \tau, \ldots \tag{45.1}$$

The type of these types is $*$. Hence, we could write this:

$$\alpha : * \tag{45.2}$$
$$\beta : * \tag{45.3}$$
$$\sigma : * \tag{45.4}$$
$$\tau : * \tag{45.5}$$

But, to help keep things clearer, let's use double dots to indicate the type of types:

$$\alpha :: * \tag{45.6}$$
$$\beta :: * \tag{45.7}$$
$$\sigma :: * \tag{45.8}$$
$$\tau :: * \tag{45.9}$$

We will continue to declare the type of term variables with a single colon, e.g.,

$$x : \alpha \tag{45.10}$$
$$y : \beta \tag{45.11}$$
$$z : \sigma \to \tau \tag{45.12}$$

Note the type of that last example: z has the type $\sigma \to \tau$. And indeed, in addition to single-character atomic types, there are also arrow types that we construct with arrows. For instance:

$$\alpha \to \alpha \tag{45.13}$$
$$\alpha \to \beta \tag{45.14}$$
$$(\alpha \to \beta) \to \alpha \tag{45.15}$$

The type of these types are complex too. For instance, the type of the type $\alpha \to \alpha$ is $* \to *$. Hence:

$$\alpha \to \alpha :: * \to * \tag{45.16}$$
$$\alpha \to \beta :: * \to * \tag{45.17}$$
$$\sigma \to \tau :: * \to * \tag{45.18}$$
$$(\alpha \to \beta) \to \alpha :: (* \to *) \to * \tag{45.19}$$

This leaves us with lots of different types of types. In the above examples, we witness these type-of-types:

$$* \tag{45.20}$$
$$* \rightarrow * \tag{45.21}$$
$$(* \rightarrow *) \rightarrow * \tag{45.22}$$

What is the type of *these*? In $\lambda\omega$, we introduce a new symbol to stand for this super type. It is $\square$. So, the type of $*$ is $\square$, the type of $* \rightarrow *$ is also $\square$, and so on.

$$* :: \square \tag{45.23}$$
$$* \rightarrow * :: \square \tag{45.24}$$
$$(* \rightarrow *) \rightarrow * :: \square \tag{45.25}$$

45.4 Defining Kinds

Anything that has the type $\square$ is called a **kind**. Formally, we can define it as any type built from one or more $*$ symbols and arrows.

Definition 38 (Kinds)**.** *The set $\mathbb{K}$ of all kinds is the smallest set that satisfies the following conditions:*

- $(*) \in \mathbb{K}$
- *For any $j, k \in \mathbb{K}$, $(j \rightarrow k) \in \mathbb{K}$*

Parentheses can be omitted if it results in no ambiguity.

That definition tells us how to construct the set of kinds. We start with an empty set:

$$\mathbb{K} = \varnothing \tag{45.26}$$

Next, we can add a single $(*)$ to the set. So now we have (with parentheses dropped):

$$\mathbb{K} = \{*\} \tag{45.27}$$

Next, we can take any kinds that are already in the set, and use arrows to build more kinds to put in the set. There is only one kind in the set at this point, so we can take it twice, and use an arrow to make a new kind, $(* \rightarrow *)$. Now the set has two kinds in it (with parentheses dropped):

$$\mathbb{K} = \{*, * \rightarrow *\} \tag{45.28}$$

Now can build more kinds using the same procedure: we take two kinds from the set, and put an arrow between them to build a new one.

For instance, we can take the first kind, which is $*$, and we can take the second kind, which is $* \to *$, and then we can put them together to build a new kind: $* \to (* \to *)$.

$$\mathbb{K} = \{*, * \to *, * \to (* \to *)\} \tag{45.29}$$

We can also take the second kind (which is $* \to *$) and the second kind (which is $*$), and put them together to make yet another kind: $(* \to *) \to *$. Now $\mathbb{K}$ has these items in it:

$$\mathbb{K} = \{*, * \to *, * \to (* \to *), (* \to *) \to *\} \tag{45.30}$$

We can go on like this infinitely, to construct more and more complex kinds. And indeed, that is exactly what the definition of $\mathbb{K}$ specifies: the set $\mathbb{K}$ contains $*$, plus any other combination of two items from $\mathbb{K}$.

Chapter 46

Sort and Var Introduction

In $\lambda^{\rightarrow}$, we declare term variables in the context. For instance, we introduce a variable x that has a type σ, like this:

$$\Gamma = x : \sigma \tag{46.1}$$

The requirement for when we need to do this is simple: if we want to use a term variable, we must declare it in the context.

In $\lambda 2$, we also declare term variables in the context. But we also declare their types. So, for instance (using the double dots convention for kinds):

$$\Gamma = \sigma :: *, x : * \tag{46.2}$$

The rule is fairly straightforward here too: if you want to use a term variable like $x : \sigma$, you must declare its type $\sigma :: *$ in the context.

The one complication in $\lambda 2$ comes with terms that have complex types. For instance, if you want to use a term with complex type like $x : \sigma \to \tau$, you must declare both σ and τ:

$$\Gamma = \sigma :: *, \tau :: *, x : \sigma \to \tau \tag{46.3}$$

In $\lambda\omega$, the rules are not so simple. $\lambda\omega$ is very strict about how you can declare types and kinds in contexts.

The procedure goes like this: you start with an empty context, then you start introducing declarations one at a time, using specified introduction rules.

46.1 The Sort Rule

For any given derivation, you should start with an empty context:

$$\Gamma = \varnothing \tag{46.4}$$

The **sort** rule says we can make one judgment at this point. The judgment is this: we can judge that $*$ is of type $\square$. Here is the rule:

$$\textit{sort}\ \frac{}{\varnothing \vdash * :: \square}$$

This is a zero-premise inference rule, meaning that we can start a derivation with it. We do not need to derive any previous judgments before we can use it.

46.1.1 Tree Example

With the *sort* rule, we can construct simple derivations. These derivations consist of only a single step, but they are still derivations.

Here is a tree example:

$$\frac{\varnothing}{\varnothing \vdash * :: \square}$$

This says that we start with nothing at the top of the tree, and then we derive the judgment that $*$ is a kind (it has the super-type $\square$).

Of course, we needn't write out the empty set at the top, so this says the same thing:

$$\frac{}{\varnothing \vdash * :: \square}$$

46.1.2 List Example

Here is the same derivation, using a list style of derivation:

$$1. \quad \varnothing \vdash * :: \square \quad \textit{sort}$$

46.1.3 Flag Example

Here is the same derivation, using the flagged context declaration style:

$$\begin{array}{r|l} 1 & \varnothing \\ \hline & \\ & 2 \quad * :: \square \quad \textit{sort}, 1 \end{array}$$

Note that we start with an empty context, and then from that context, we use the *sort* rule to derive the judgment that $* :: \square$.

Of course, it is tedious to always write the empty context there, so we can just omit it, on the understanding that everybody understands how the *sort* rule works. Then the derivation simply looks like this:

$$1 \quad * :: \square \quad \textit{sort}$$

46.2 The Var Rule

The *var* rule is a rule that lets you introduce variables into the context and judgment. It typically stated in a generic form, something like this:

$$var\ \frac{\Gamma \vdash A :: \mathcal{S}}{\Gamma, \phi :: A \vdash \phi :: A}\ \text{where } \phi \notin \Gamma$$

This is stated in a generic form, because $A :: \mathcal{S}$ is supposed to stand for one of these two options:

- $* :: \Box$, in which case $\phi :: A$ should be something like $\alpha : *$
- $\alpha :: *$, in which case $\phi :: A$ should be something like $x : \alpha$

Hence, this one rule really breaks out into two rules. Under one formulation of it, it lets you introduce type variables:

$$\textit{type-var intro}\ \frac{\Gamma \vdash * :: \Box}{\Gamma, \alpha :: * \vdash \alpha :: *}\ \text{where } \alpha \notin \Gamma$$

Under the other formulation of it, it lets you introduce term variables:

$$\textit{term-var intro}\ \frac{\Gamma \vdash \alpha :: *}{\Gamma, x : \alpha \vdash x : \alpha}\ \text{where } x \notin \Gamma$$

To keep these two formulations separate, we will treat the *var* rule as if it were these two, separate rules.

46.3 Type-Var Introduction

Once we have a judgment which asserts $* :: \Box$, we can introduce type variables. The rule for this runs as follows (where α can stand for any type from $\mathbb{T}$):

$$\textit{type-var intro}\ \frac{\Gamma \vdash * :: \Box}{\Gamma, \alpha :: * \vdash \alpha :: *}\ \text{where } \alpha \notin \Gamma$$

This says that if you have a judgment $* :: \Box$, you can then introduce a type variable α of type $*$ on both sides of the $\vdash$ symbol. That is, you put $\alpha :: *$ in the context, and then you put $\alpha :: *$ in the judgment too.

Note that Γ can be any valid context. For instance, it could be empty, or it could have any number of declarations in it. The *type-var intro* rule requires only that it is the same above the line and below the line — that is, it is the same in the premise as it is in the derived conclusion, except that in the derived conclusion $\alpha :: *$ is added to it.

Note also that in this rule, α can stand for any type from $\mathbb{T}$. For instance, it could stand for an atomic type like α or σ, but it could also stand for a complex type such as $\alpha \rightarrow \beta$ or $(\sigma \rightarrow \tau) \rightarrow \sigma$.

46.3.1 Tree Example

Here is an example of deriving the judgment $\alpha :: *$, using the tree format.

$$\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{\varnothing}{\varnothing \vdash * :: \Box}}{\varnothing, \alpha :: * \vdash \alpha :: *}$$

In this derivation, we start with nothing (at the top of the tree). Then we use the *sort* rule to derive the judgment $* :: \Box$. That gives us a judgment that matches the form of the premise in the *type-var intro* rule, so we can use *type-var intro* to then introduce $\alpha :: *$.

Of course, we can write this without the empty set symbols:

$$\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *}$$

Here is another derivation. It is almost identical, except we use the *type-var intro* rule to introduce a different type variable, namely $\sigma :: *$.

$$\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \Box}}{\sigma :: * \vdash \sigma :: *}$$

Here is a third example, where we introduce a complex type:

$$\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \Box}}{\sigma \to \tau :: * \vdash \sigma \to \tau :: *}$$

Note that Γ can be any valid context. It can be empty, as it is in the three examples we just walked through. But it could also have something in it. For instance, suppose we have derived a judgment that looks like this (note that the context is not empty):

$$\dfrac{\vdots}{\alpha :: *, \beta :: * \vdash * :: \Box}$$

We can still use the *type-var intro* rule to introduce another type variable here — say, $\sigma :: *$. The only requirement is that we have to keep the context the same, except that we add the new $\sigma :: *$. The derivation looks like this:

$$\textit{type-var intro}\ \dfrac{\dfrac{\vdots}{\alpha :: *, \beta :: * \vdash * :: \Box}}{\alpha :: *, \beta :: *, \sigma :: * \vdash \sigma :: *}$$

Notice that the context above the *type-var intro* line — which is $\alpha :: *, \beta :: *$ — is the same below the line, except that below the line we add $\sigma :: *$ to the context (and the judgment).

46.3.2 List Example

Here is the derivation of $\alpha :: *$ from above, using a list style of derivation:

$$\begin{array}{lll} 1. & \varnothing \vdash * :: \Box & \textit{sort} \\ 2. & \varnothing, \alpha :: * \vdash \alpha :: * & \textit{type-var intro}, 1 \end{array}$$

Again, we can drop the empty set symbols if we like:

$$\begin{array}{lll} 1. & \varnothing \vdash * :: \Box & \textit{sort} \\ 2. & \alpha :: * \vdash \alpha :: * & \textit{type-var intro}, 1 \end{array}$$

Here is the derivation of $\sigma :: *$ from above:

$$\begin{array}{lll} 1. & \varnothing \vdash * :: \Box & \textit{sort} \\ 2. & \sigma :: * \vdash \sigma :: * & \textit{type-var intro}, 1 \end{array}$$

And here is a derivation of $\sigma \to \tau :: *$ from above:

$$\begin{array}{lll} 1. & \varnothing \vdash * :: \Box & \textit{sort} \\ 2. & \sigma \to \tau :: * \vdash \sigma \to \tau :: * & \textit{type-var intro}, 1 \end{array}$$

Assume that we have derived $\alpha :: *, \beta :: * \vdash * :: \Box$, so that we have a derivation that looks something like this:

$$\begin{array}{ll} & \vdots \\ 7. & \alpha :: *, \beta :: * \vdash * :: \Box \end{array}$$

We could then use *type-var intro* to introduce $\sigma :: *$ as we did above, in the tree style derivation:

$$\begin{array}{lll} & \vdots & \\ 7. & \alpha :: *, \beta :: * \vdash * :: \Box & \\ 8. & \alpha :: *, \beta :: *, \sigma :: * \vdash \sigma :: * & \textit{type-var intro}, 7 \end{array}$$

46.3.3 Flag Example

Let us construct the derivation of $\alpha :: *$ from above, using the flagged context declaration style. Let us begin by first deriving $* :: \Box$ as we did before:

$$\begin{array}{l|lll} 1 & \varnothing & & \\ \hline & 2 & * :: \Box & \textit{sort}, 1 \end{array}$$

At this point, we have a judgment which states that $* :: \Box$, and that is exactly what we need for a premise if we want to use the *type-var intro* rule. So now we can use the *type-var intro* rule, and introduce the type variable $\alpha :: *$.

According to the *type-var intro* rule, we must introduce $\alpha :: *$ into both the context, and into the judgment. In a flag-style derivation, we have to do that on two separate lines, since contexts and judgments are separated onto distinct lines in flag-style derivations.

First then, we put $\alpha :: *$ in the context:

```
1 │ ∅
  ├──────────
  │
  │ 2   * :: □   sort, 1
  │ 3 │ α :: *
  │   ├───────
  │   │
  │
```

Then, we make the judgment underneath it that $\alpha :: *$:

```
1 │ ∅
  ├──────────
  │
  │ 2   * :: □   sort, 1
  │ 3 │ α :: *
  │   ├──────────
  │   │
  │   │ 4   α :: *   type-var intro, 2, 3
  │
```

For clarity, I have marked that line 4 is derived by using the *type-var intro* rule, and that it comes from lines 2 and 3. Strictly speaking, we don't have to mention line 3, but you can if you feel that you need the extra bookkeeping.

Of course, we do not need to put $\varnothing$ on its own line, so we could rewrite the derivation like this:

```
1   * :: □   sort
2 │ α :: *
  ├──────────
  │
  │ 3   α :: *   type-var intro, 1, 2
```

Here is the derivation of $\sigma :: *$ from above.

```
1   * :: □   sort
2 │ σ :: *
  ├──────────
  │
  │ 3   σ :: *   type-var intro, 1, 2
```

And here is the derivation of $\sigma \to \tau :: *$ from above.

```
1    ∗ :: □    sort
2   │ σ → τ :: ∗
    ├──────────────
    │
    │ 3   σ → τ :: ∗    type-var intro, 1, 2
```

Suppose we have derived $\alpha :: *, \beta :: * \vdash * :: \Box$. In the flag-style derivation, that would look something like this:

```
1    ∗ :: □    sort
     ⋮
6   │ α :: ∗
    ├──────────
    │
    │ 7   α :: ∗    type-var intro, 1, 6
    │ 8  │ β :: ∗
    │    ├──────────
    │    │
    │    │ 9    β :: ∗    type-var intro, 1, 8
    │    │      ⋮
    │    │ 13   ∗ :: □
    │
```

Here we have the judgment that $* :: \Box$, which is the premise we need to use the *type-var-intro* rule. So we can introduce $\sigma :: *$ here too:

```
1    ∗ :: □    sort
     ⋮
6   │ α :: ∗
    ├──────────
    │
    │ 7   α :: ∗    type-var intro, 1, 6
    │ 8  │ β :: ∗
    │    ├──────────
    │    │
    │    │ 9    β :: ∗    type-var intro, 1, 8
    │    │      ⋮
    │    │ 13   ∗ :: □
    │    │ 14  │ σ :: ∗
    │    │     ├──────────
    │    │     │
    │    │     │ 15   σ :: ∗    type-var intro, 13, 14
    │    │
    │
```

46.4 Term-Var Introduction

Once we have a judgment which asserts some type variable, say $\alpha :: *$, we can then introduce a term variable that has that type, e.g., $x : \alpha$. The rule runs as follows (where x can stand for any term variable and α can stand for any type from $\mathbb{T}$):

$$\textit{term-var intro}\ \frac{\Gamma \vdash \alpha :: *}{\Gamma, x : \alpha \vdash x : \alpha}\ \text{where } x \notin \Gamma$$

This is very similar to *type-var intro*, except that here we are introducing not a type, but rather a term that has that type.

The rule says that if you have a judgment $\alpha :: *$, you can then introduce a term variable x of type α on both sides of the $\vdash$ symbol. That is, you put $x : \alpha$ in the context, and then you put $x : \alpha$ in the judgment too.

As with the *type-var intro* rule, Γ can be any valid context, and α can be any type from $\mathbb{T}$.

46.4.1 Tree Example

Here is an example of deriving the judgment $x :: \alpha$, using the tree format.

$$\textit{term-var intro}\ \dfrac{\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *, x : \alpha \vdash x : \alpha}$$

In this derivation, we use *sort* and *type-var intro* to introduce the type $\alpha : *$. That gives us a judgment which matches the form of the premise in the *term-var intro* rule, so we can use *term-var intro* to introduce $x : \alpha$.

Here is another derivation. It is almost identical to the one we just did, except in this one we introduce a complex type variable, namely $\sigma \to \tau$.

$$\textit{term-var intro}\ \dfrac{\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square}}{\sigma \to \tau :: * \vdash \sigma \to \tau :: *}}{\sigma \to \tau :: *, x : \sigma \to \tau \vdash x : \sigma \to \tau}$$

Notice that in these derivations, the context Γ stays the same above and below the *term-var intro* line, except that below the line we add the new term variable to it ($x : \alpha$ or $x : \sigma \to \tau$, as the case may be).

It is important in these $\lambda\omega$ derivations that we are very strict and very exact as we handle the context.

46.4.2 List Example

Here is the derivation of $x : \alpha$ from above, using a list style of derivation:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\alpha :: * \vdash \alpha :: *$	*type-var intro*, 1
3.	$\alpha :: *, x : \alpha \vdash x : \alpha$	*term-var intro*, 2

And here is the derivation of $x : \sigma \to \tau$ from above, using a list style of derivation:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\sigma \to \tau :: * \vdash \sigma \to \tau :: *$	*type-var intro*, 1
3.	$\sigma \to \tau :: *, x : \sigma \to \tau \vdash x : \sigma \to \tau$	*term-var intro*, 2

46.4.3 Flag Example

Here is the same derivation of $x : \alpha$ that we did above, but in the flag-style derivation:

```
1    * :: □    sort
2  | α :: *
   |---------
   |
   | 3   α :: *   type-var intro, 1, 2
   | 4 | x : α
   |   |---------
   |   |
   |   | 5   x : α   term-var intro, 3, 4
   |
```

And here is the derivation of $x : \sigma \to \tau$:

```
1    * :: □    sort
2  | σ → τ :: *
   |-------------
   |
   | 3   σ → τ :: *   type-var intro, 1, 2
   | 4 | x : σ → τ
   |   |-------------
   |   |
   |   | 5   x : σ → τ   term-var intro, 3, 4
   |
```

Chapter 47

Weakening

The *sort*, *type-var intro*, and *term-var intro* rules are very strict. There is another rule called *weakening* that allows us to introduce more declarations into the context.

Like the *var* rule, the *weakening* rule is often stated in a generic form that lets it cover both type variables and term variables. To keep things clearer, we will split the *weakening* rule into a pair of rules: one for type variables, and one for term variables.

47.1 The Idea of Weakening and Thinning

Suppose you have a judgment that looks something like this:

$$\Gamma \vdash A : B \tag{47.1}$$

For the moment, let's not care what $A : B$ is. Let's just assume that it is validly derived judgment of whatever sort we please. (It could be a single or double dot judgment, like $x : \alpha$, or $\alpha :: *$.)

The idea of weakening is this. We can add any arbitrary declaration to the context, and that won't change the fact that $A : B$ is a valid judgment.

Here is an example from $\lambda^{\rightarrow}$. This is a valid judgment:

$$x : \alpha \vdash x : \alpha \tag{47.2}$$

According to the *var* rule, if a variable is declared to have a particular type in the context, then you can make a judgment which asserts that the variable has that particular type.

Now, let's add some extra stuff into the context. For instance:

$$x : \alpha, y : \beta \vdash x : \alpha \tag{47.3}$$

Is the judgment still valid? Yes it is. Adding $y : \beta$ did nothing to change this. If anything $y : \beta$ is just some irrelevant noise on the side.

We can add more declarations too. For instance:

$$x : \alpha, y : \beta, \ldots, z : \alpha \to \beta \vdash x : \alpha \tag{47.4}$$

None of this changes the validity of the judgment. It still follows that x has type α. We can add all the superfluous declarations to the context that we like, and the judgment still holds.

Adding superfluous stuff to the context is generally called **thinning**. The basic concept is just that we are thinning out the context by adding extra stuff into it. And indeed, recall that earlier we used structural induction to prove that thinning has no affect on the validity of derivations.

Thinning is a general term though. It refers to *any* thinning of the context. That is, it refers to adding extra declarations anywhere within the context — you can put extra declarations at the beginning of the context, somewhere in the middle, or at the end.

Weakening is a particular type of thinning. Weakening occurs only when you add an extra declaration to the *end* of the context. In $\lambda\omega$, we only allow weakening (there are no rules in $\lambda\omega$ for thinning in general; there are only rules for weakening).

47.2 The Type Weakening Rule

The **type weakening** rule says that if you have derived some judgment $A : B$, you can weaken the context by declaring a type. Roughly, the rule looks like this (where α can be any type from $\mathbb{T}$):

$$\textit{type weakening}\ \frac{\Gamma \vdash A : B}{\Gamma, \alpha :: * \vdash A : B}$$

The basic idea is just that of weakening: if you've derived $A : B$, you can weaken the context with another declaration (in this case, $\alpha :: *$), but that doesn't change anything. You can still derive $A : B$ from that context.

If it helps, you can think of this rule as having a slot with question marks instead of $\alpha :: *$. Since $\alpha :: *$ is really just a placeholder for any type variable declaration, a slot that you can fill in is just as good:

$$\textit{type weakening}\ \frac{\Gamma \vdash A : B}{\Gamma, ?? \vdash A : B}$$

The question marks serve as a visual indicator to signify that you can put any type variable in their place, in order to weaken the context. But even when you've weakened the context, you can still derive $A : B$.

Nevertheless, the slot with question marks is a bit too uninformative, because they don't indicate what sort of declaration you can put in their place. Since we are formulating a *type weakening* rule, what we want to say is that you can only put a *type variable* declaration in the place of the question marks.

So it is better to formulate the rule with $\alpha :: *$ as a placeholder, because that makes it clear that, with this rule, we can only weaken the context with a type variable declaration:

$$\textit{type weakening}\ \frac{\Gamma \vdash A : B}{\Gamma, \alpha :: * \vdash A : B}$$

Still, the rule is not quite correct, when it is stated in this way. The reason is that $\alpha :: *$ cannot be introduced into the context unless $*$ is introduced first. So on the side, we need an additional judgment which asserts that $* :: \Box$ is derived from the same context. Hence, the full rule is written like this:

$$\textit{type weakening}\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash * :: \Box}{\Gamma, \alpha :: * \vdash A : B}$$

This says that *if* you have derived $A : B$ from the context Γ, *and* if you have derived $* :: \Box$ from the *same* context Γ, *then* you can weaken the context with a type variable declaration such as $\alpha :: *$. By weakening the context in this way, you won't change the fact that you can still derive $A : B$ from that context.

47.2.1 Tree Example

Let's use the *type weakening* rule to derive the judgment $\alpha :: * \vdash * :: \Box$. To start, use the *sort* rule twice, to get two copies of $* :: \Box$, side by side:

$$\frac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \Box} \qquad \dfrac{}{\varnothing \vdash * :: \Box}\ \textit{sort}}{}$$

Notice that we have two premises here, of this form:

$$\frac{\textit{sort}\ \dfrac{}{\Gamma \vdash A : B} \qquad \dfrac{}{\Gamma \vdash * :: \Box}\ \textit{sort}}{}$$

On the left side, there is a judgment $\Gamma \vdash A : B$, where Γ is empty and $A : B$ is $* :: \Box$. On the right there is a judgment $\Gamma \vdash * :: \Box$, where Γ is empty (the same as it is on the left).

That matches the two premises we need to use the *type weakening* rule. So, with the *type weakening* rule, we should be able to derive the same judgment from the left, but with the context weakened by some extra type variable declaration.

Let's put the left judgment down below the line, with a slot where we can put the weakening declaration:

$$\textit{type weakening}\ \frac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \Box} \qquad \dfrac{}{\varnothing \vdash * :: \Box}\ \textit{sort}}{\varnothing, ?? \vdash * :: \Box}$$

What shall we weaken it with? We can pick any type variable declaration. Let's pick $\alpha :: *$:

$$\textit{type weakening}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square} \qquad \dfrac{}{\varnothing \vdash * :: \square}\ \textit{sort}}{\varnothing, \alpha :: * \vdash * :: \square}$$

Here it is again, but we weaken the context with the type variable $\sigma :: *$:

$$\textit{type weakening}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square} \qquad \dfrac{}{\varnothing \vdash * :: \square}\ \textit{sort}}{\varnothing, \sigma :: * \vdash * :: \square}$$

And here it is a third time, but we weaken the context with the type variable $\sigma \to \tau :: *$:

$$\textit{type weakening}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square} \qquad \dfrac{}{\varnothing \vdash * :: \square}\ \textit{sort}}{\varnothing, \sigma \to \tau :: * \vdash * :: \square}$$

47.2.2 List Example

Let us construct the derivation of $\alpha :: * \vdash * :: \square$ from above, using a list-style derivation. First, let us use *sort* twice:

1.	$\varnothing \vdash * :: \square$	*sort*
2.	$\varnothing \vdash * :: \square$	*sort*

Now we have the two premises we need to weaken the first context. Let's put the 3rd line down, with a slot filled with question marks to show where we can weaken the first judgment:

1.	$\varnothing \vdash * :: \square$	*sort*
2.	$\varnothing \vdash * :: \square$	*sort*
3.	$\varnothing, ?? \vdash * :: \square$	*type weakening*, 1, 2

What shall we put in the place of the question marks? Here it is with $\alpha :: *$:

1.	$\varnothing \vdash * :: \square$	*sort*
2.	$\varnothing \vdash * :: \square$	*sort*
3.	$\varnothing, \alpha :: * \vdash * :: \square$	*type weakening*, 1, 2

Here it is with $\sigma :: *$:

1.	$\varnothing \vdash * :: \square$	*sort*
2.	$\varnothing \vdash * :: \square$	*sort*
3.	$\varnothing, \sigma :: * \vdash * :: \square$	*type weakening*, 1, 2

And here it is with $\sigma \to \tau :: *$:

1.	$\varnothing \vdash * :: \square$	*sort*
2.	$\varnothing \vdash * :: \square$	*sort*
3.	$\varnothing, \sigma \to \tau :: * \vdash * :: \square$	*type weakening*, 1, 2

47.2.3 Flag Example

Let us construct the same derivations again, but this time let's use a flag-style derivation. First, let us use *sort* twice:

$$\begin{array}{lll} 1 & * :: \Box & \textit{sort} \\ 2 & * :: \Box & \textit{sort} \end{array}$$

Now let's introduce the slot where we can weaken the context:

$$\begin{array}{llll} 1 & * :: \Box & \textit{sort} & \\ 2 & * :: \Box & \textit{sort} & \\ 3 & \multicolumn{3}{|l}{??} \\ \cline{2-3} & \multicolumn{3}{|l}{4 \quad * :: \Box \quad \textit{type weakening}, 1, 2} \end{array}$$

This says that we can add something to the context, and still derive the judgment $* :: \Box$. Let's put $\alpha :: *$ in there:

$$\begin{array}{llll} 1 & * :: \Box & \textit{sort} & \\ 2 & * :: \Box & \textit{sort} & \\ 3 & \multicolumn{3}{|l}{\alpha :: *} \\ \cline{2-3} & \multicolumn{3}{|l}{4 \quad * :: \Box \quad \textit{type weakening}, 1, 2} \end{array}$$

Here is the same derivation, but weakened with $\sigma :: *$:

$$\begin{array}{llll} 1 & * :: \Box & \textit{sort} & \\ 2 & * :: \Box & \textit{sort} & \\ 3 & \multicolumn{3}{|l}{\sigma :: *} \\ \cline{2-3} & \multicolumn{3}{|l}{4 \quad * :: \Box \quad \textit{type weakening}, 1, 2} \end{array}$$

And here it is weakened with $\sigma \to \tau :: *$:

$$\begin{array}{llll} 1 & * :: \Box & \textit{sort} & \\ 2 & * :: \Box & \textit{sort} & \\ 3 & \multicolumn{3}{|l}{\sigma \to \tau :: *} \\ \cline{2-3} & \multicolumn{3}{|l}{4 \quad * :: \Box \quad \textit{type weakening}, 1, 2} \end{array}$$

47.3 The Term Weakening Rule

The **term weakening** rule says that if you have derived some judgment $A : B$, you can weaken the context by declaring a term variable. Roughly, the rule looks like this (where x can be any term variable and α can be any type from $\mathbb{T}$):

$$\textit{term weakening}\ \frac{\Gamma \vdash A : B}{\Gamma, x : \alpha \vdash A : B}$$

The basic idea is again just that of weakening: if you've derived $A : B$, you can weaken the context with another declaration (in this case, $x : \alpha$), but that doesn't change anything. You can still derive $A : B$ from that context.

However, stated in this form, the rule is not quite correct. The reason is that $x : \alpha$ cannot be introduced into the context unless α is already declared first. So we also need a judgment which asserts that $\alpha :: *$. Hence, the full rule is written like this:

$$\textit{term weakening}\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash \alpha :: *}{\Gamma, x : \alpha \vdash A : B}$$

This says that *if* you have derived $A : B$ from the context Γ, *and* if you have derived $\alpha :: *$ from the *same* context Γ, *then* you can weaken the context with a term variable like $x : \alpha$. By weakening the context, it won't change the fact that you can still derive $A : B$ from that context.

As before, we could replace $x : \alpha$ with a slot full of question marks, in order to indicate that we can put any term variable declaration in the slot. Like this:

$$\textit{term weakening}\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash \alpha :: *}{\Gamma, ?? \vdash A : B}$$

But the question marks are not informative enough. The point we want to emphasize with this rule is that we can weaken the context with a *term variable* declaration, and in particular we have to weaken it with a variable that is of type α. So it is better to put $x : \alpha$ instead, as we had it before:

$$\textit{term weakening}\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash \alpha :: *}{\Gamma, x : \alpha \vdash A : B}$$

Note that the type variable α has to be the same in the top right premise and in the declaration below the line. The following would be invalid:

$$\textit{term weakening}\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash \alpha :: *}{\Gamma, x : \beta \vdash A : B}$$

Why is that invalid? Because the type $\beta :: *$ hasn't been introduced anywhere. We've only introduced $\alpha :: *$ (in the top right judgment), so you can only weaken the context with a term variable that has type α.

This would be valid though:

$$\textit{term weakening}\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash \beta :: *}{\Gamma, x : \beta \vdash A : B}$$

47.3.1 Tree Example

Suppose we have derived that $\alpha :: * \vdash \alpha :: *$. For instance, suppose we have done this through *sort* and *type-var intro*:

$$\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash \alpha :: *}$$

Here the context is $\alpha :: *$, and the judgment is also $\alpha :: *$. (The fact that they are the same is irrelevant. Later we will see examples where the context and judgments are different.)

What I would like to do now is weaken the context by adding some term variable to the context. For instance, suppose I want to add $x : \alpha$ to it, to get something like this:

$$\textit{term weakening}\ \dfrac{\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *, x : \alpha \vdash \alpha :: *}$$

However, I cannot introduce $x : \alpha$ without first deriving $\alpha :: *$ from the same context. So I need to also derive that, over on the side.

Of course, we know how to derive $\alpha :: *$ from the context $\alpha :: *$. We just did it at the start of this section, using the *sort* and *type-var intro* rules. Hence, I can begin that derivation over to the right, by starting with the *sort* rule:

$$\textit{term weakening}\ \dfrac{\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash \alpha :: *} \qquad \begin{array}{c} \dfrac{}{\varnothing \vdash * :: \square}\ \textit{sort} \\ \vdots \end{array}}{}$$

Then I can use the *type-var intro* rule to derive $\alpha :: * \vdash \alpha :: *$:

$$\textit{term weakening}\ \dfrac{\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash \alpha :: *} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \square}\ \textit{sort}}{\alpha :: * \vdash \alpha :: *}\ \textit{type-var intro}}{}$$

Now I have the correct premises on the left and right. I have the same context on the right as I do over on the left (the context is $\alpha :: *$), and I have the type introduced on the right (the judgment is $\alpha :: *$). So I can use the *term weakening* rule to introduce a term variable of type α into the context to weaken it. Let's use $x : \alpha$:

$$\textit{term weakening}\ \dfrac{\textit{type-var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash \alpha :: *} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \square}\ \textit{sort}}{\alpha :: * \vdash \alpha :: *}\ \textit{type-var intro}}{\alpha :: *, x : \alpha \vdash \alpha :: *}$$

47.3.2 List Example

Let us construct the same derivation, using a list-style derivation. First, let us use *sort* and *type-var intro* to derive that $\alpha :: * \vdash \alpha :: *$:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\alpha :: * \vdash \alpha :: *$	*type-var intro*, 1

What I want to do next is weaken the context with, say, $x : \alpha$, but I cannot do that because I have not introduced the type $\alpha :: *$ already. So, let's do that. We can use *sort* and *type-var intro* again to do that:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\alpha :: * \vdash \alpha :: *$	*type-var intro*, 1
3.	$\varnothing \vdash * :: \Box$	*sort*
4.	$\alpha :: * \vdash \alpha :: *$	*type-var intro*, 3

Now I have on line 2 a judgment of the form $\Gamma \vdash A : B$ (where Γ is $\alpha :: *$ and $A : B$ is $\alpha :: *$). I also have on line 3 a judgment of the form $\Gamma \vdash \alpha :: *$ (where Γ is the same context as on line 2, namely $\alpha :: *$). And those are the two premises I need to use the *term weakening* rule. So, now I can weaken the context with $x : \alpha$:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\alpha :: * \vdash \alpha :: *$	*type-var intro*, 1
3.	$\varnothing \vdash * :: \Box$	*sort*
4.	$\alpha :: * \vdash \alpha :: *$	*type-var intro*, 3
5.	$\alpha :: *, x : \alpha \vdash \alpha :: *$	*term weakening*, 2, 4

47.3.3 Flag Example

Let us construct the same derivation again, but this time let's use a flag-style derivation. First, let us use *sort* and *type-var intro* to derive that $\alpha :: * \vdash \alpha :: *$:

1	$* :: \Box$	*sort*	
2	$\alpha :: *$		
	3	$\alpha :: *$	*type-var intro*, 1, 2

Next, we need to derive that again, just as we did before.

```
1   $* :: \Box$   *sort*
2   | $\alpha :: *$
    |-----------
    |
    | 3   $\alpha :: *$   *type-var intro*, 1, 2
    | 4   $* :: \Box$   *sort*
    | 5   | $\alpha :: *$
    |     |-----------
    |     |
    |     | 6   $\alpha :: *$   *type-var intro*, 4, 5
    |
```

Now we have the two premises we need to use the term weakening rule, namely lines 3 and 6. The rule says we can weaken the context, so let's introduce $x : \alpha$ into the context:

```
1   $* :: \Box$   *sort*
2   | $\alpha :: *$
    |-----------
    |
    | 3   $\alpha :: *$   *type-var intro*, 1, 2
    | 4   $* :: \Box$   *sort*
    | 5   | $\alpha :: *$
    |     |-----------
    |     |
    |     | 6   $\alpha :: *$   *type-var intro*, 4, 5
    |     | 7   | $x : \alpha$
    |     |     |--------
    |     |
    |
```

And finally, the rule says that even after we've weakened the context (through line 6 and 7), we can still derive the same judgment as we did in line 3, which is $\alpha :: *$:

```
1   $* :: \Box$   *sort*
2   | $\alpha :: *$
    |-----------
    |
    | 3   $\alpha :: *$   *type-var intro*, 1, 2
    | 4   $* :: \Box$   *sort*
    | 5   | $\alpha :: *$
    |     |-----------
    |     |
    |     | 6   $\alpha :: *$   *type-var intro*, 4, 5
    |     | 7   | $x : \alpha$
    |     |     |-----------
    |     |     |
    |     |     | 8   $\alpha :: *$   *term weakening*, 3, 6, 7
    |     |
    |
```

Chapter 48

Weakening Tricks

With *sort* and *type/term-var intro* rules, we can declare type and term variables in both the context and the judgment simultaneously. With *weaking*, we are able to add type and term variable declarations to the end of a context.

This is still very limited. If we want to add arbitrary types and terms to the context, we have to be clever in our use of the rules. By combining the just-mentioned rules in different ways, we can find ways to introduce a variety of different declarations into the context.

48.1 Adding a Second Type Declaration

We can use *sort* to derive $* :: \square$:

$$\textit{sort}\ \frac{}{\varnothing \vdash * :: \square}$$

To introduce $\alpha :: *$ into the context, we can use the sort rule twice, and then we can use weakening:

$$\textit{type weakening}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \square} \qquad \dfrac{}{\varnothing \vdash * :: \square}\ \textit{sort}}{\alpha :: * \vdash * :: \square}$$

How do we add another type declaration to that context? Suppose we want to add $\beta :: *$ to the context, so that we have this:

$$\alpha :: *, \beta :: * \vdash * :: \square$$

How do we do that? Weakening is the rule that can do this. But we need to derive $\alpha :: * \vdash * :: \square$ twice, as follows (I omit the labels for each inference to save space):

$$\dfrac{\dfrac{}{\varnothing \vdash * :: \square} \qquad \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash * :: \square} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \square} \qquad \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash * :: \square}$$

That leaves us with the right form of premises to use weakening. Here is the same derivation, with Γ and $A : B$ substituted in, so we can see the structure of the premises:

$$\dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\Gamma \vdash A : B} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\Gamma \vdash * :: \Box}$$

Since we have premises in the right form, we can use type-weakening. The type weakening rule says we can put the premise on the left (which is of the form $\Gamma \vdash A : B$) down under the line, and we can weaken it with a type declaration. Like this, with a slot for where the weakening type declaration goes:

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\Gamma \vdash A : B} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\Gamma \vdash * :: \Box}}{\Gamma, ?? \vdash A : B}$$

Of course, we want to introduce $\beta :: *$, so let's put that in:

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\Gamma \vdash A : B} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\Gamma \vdash * :: \Box}}{\Gamma, \beta :: * \vdash A : B}$$

Now let us put our original values back in. We replace Γ with the original context (which was $\alpha :: *$), and we replace $A : B$ with the original judgment on the left (which was $* :: \Box$):

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash * :: \Box} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash * :: \Box}}{\alpha :: *, \beta :: * \vdash * :: \Box}$$

48.1.1 List Example

We can do the same thing, using a list-style derivation. First, we derive $\alpha :: * \vdash * :: \Box$:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\varnothing \vdash * :: \Box$	*sort*
3.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 1, 2

Then we do it again to get our second copy:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\varnothing \vdash * :: \Box$	*sort*
3.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 1, 2
4.	$\varnothing \vdash * :: \Box$	*sort*
5.	$\varnothing \vdash * :: \Box$	*sort*
6.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 4, 5

Then we can weaken the context of the judgment from line 3 with $\beta :: *$:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\varnothing \vdash * :: \Box$	*sort*
3.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 1, 2
4.	$\varnothing \vdash * :: \Box$	*sort*
5.	$\varnothing \vdash * :: \Box$	*sort*
6.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 4, 5
7.	$\alpha :: *, \beta :: * \vdash * :: \Box$	*type weakening*, 3, 6

48.2 Adding a Third Type Declaration

We can use the same technique to introduce more type variables. For instance, we could introduce $\gamma :: *$, or $\sigma \to \tau :: *$.

The trick to notice is that we make two copies of the derivation, and then we weaken the one on the left. Think about how we did this above:

- When we derived $\alpha :: * \vdash * :: \Box$, we used two copies of $\varnothing \vdash * :: \Box$, and then we weakened the one on the left.

- When we derived $\alpha :: *, \beta :: * \vdash * :: \Box$, we used two copies of $\alpha :: * \vdash * :: \Box$, and then we weakened the one on the left.

Likewise, if we want to add another type variable to the context of $\alpha :: *, \beta :: * \vdash * :: \Box$, we can use two copies of $\alpha :: *, \beta :: * \vdash * :: \Box$, and then weaken the one on the left with another type variable, for instance $\gamma :: *$ or $\sigma \to \tau :: *$.

Of course, the trees are now too big to fit on the page. So let's introduce an abbreviation. Take our original derivation of $\alpha :: *, \beta :: * \vdash * :: \Box$, which is this:

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash * :: \Box} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash * :: \Box}}{\alpha :: *, \beta :: * \vdash * :: \Box}$$

Let's replace everything above the bottom line with a *QED* and vertical dots, to indicate that we've already derived it. Like this:

$$\frac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box}$$

Then we can repeat that same derivation on the right:

$$\frac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box} \qquad \frac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box}$$

And now we can use *type weakening* to weaken the context. Here it is with question marks to indicate where we will weaken it:

$$\frac{\dfrac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box} \qquad \dfrac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box}}{\alpha :: *, \beta :: *, ?? \vdash * :: \Box}$$

Now we can put $\gamma :: *$ in the place of the question marks, or $\sigma \to \tau :: *$, or any other valid type declaration. Here we put in $\gamma :: *$:

$$\frac{\dfrac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box} \qquad \dfrac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box}}{\alpha :: *, \beta :: *, \gamma :: * \vdash * :: \Box}$$

We can repeat this to add another type declaration like $\sigma \to \tau :: *$ too. Again, we use two copies of the derivation, and weaken the one on the left. To use the vertical dots abbreviation:

$$\frac{\dfrac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: *, \gamma :: * \vdash * :: \Box} \qquad \dfrac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: *, \gamma :: * \vdash * :: \Box}}{\alpha :: *, \beta :: *, \gamma :: *, \sigma \to \tau :: * \vdash * :: \Box}$$

48.2.1 List Example

We can do the same thing, using a list-style derivation. First, we derive $\alpha :: *, \beta :: * \vdash * :: \Box$ as we did above:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\varnothing \vdash * :: \Box$	*sort*
3.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 1, 2
4.	$\varnothing \vdash * :: \Box$	*sort*
5.	$\varnothing \vdash * :: \Box$	*sort*
6.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 4, 5
7.	$\alpha :: *, \beta :: * \vdash * :: \Box$	*type weakening*, 3, 6

Then we perform the same derivation to make a second copy:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\varnothing \vdash * :: \Box$	*sort*
3.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 1, 2
4.	$\varnothing \vdash * :: \Box$	*sort*
5.	$\varnothing \vdash * :: \Box$	*sort*
6.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 4, 5
7.	$\alpha :: *, \beta :: * \vdash * :: \Box$	*type weakening*, 3, 6
8.	$\varnothing \vdash * :: \Box$	*sort*
9.	$\varnothing \vdash * :: \Box$	*sort*
10.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 8, 9
11.	$\varnothing \vdash * :: \Box$	*sort*
12.	$\varnothing \vdash * :: \Box$	*sort*
13.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 11, 12
14.	$\alpha :: *, \beta :: * \vdash * :: \Box$	*type weakening*, 10, 13

And now we can weaken the context for the judgment on line 7. So, for instance, we can add $\gamma :: *$:

1.	$\varnothing \vdash * :: \Box$	*sort*
2.	$\varnothing \vdash * :: \Box$	*sort*
3.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 1, 2
4.	$\varnothing \vdash * :: \Box$	*sort*
5.	$\varnothing \vdash * :: \Box$	*sort*
6.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 4, 5
7.	$\alpha :: *, \beta :: * \vdash * :: \Box$	*type weakening*, 3, 6
8.	$\varnothing \vdash * :: \Box$	*sort*
9.	$\varnothing \vdash * :: \Box$	*sort*
10.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 8, 9
11.	$\varnothing \vdash * :: \Box$	*sort*
12.	$\varnothing \vdash * :: \Box$	*sort*
13.	$\alpha :: * \vdash * :: \Box$	*type weakening*, 11, 12
14.	$\alpha :: *, \beta :: * \vdash * :: \Box$	*type weakening*, 10, 13
15.	$\alpha :: *, \beta :: *, \gamma :: * \vdash * :: \Box$	*type weakening*, 7, 14

Chapter 49

More Weakening Tricks

We can take the derivations we worked through in the last chapter to construct further ways to introduce types and terms into the context.

49.1 Weakening Type Variable Judgments

Suppose we have a judgment where a type variable is declared on both sides of the $\vdash$ symbol:

$$\alpha :: * \vdash \alpha :: * \tag{49.1}$$

Suppose we want to weaken the context by adding another type variable declaration. Something like this:

$$\alpha :: *, ?? \vdash \alpha :: * \tag{49.2}$$

For instance, suppose we want to put $\beta :: *$ in the place of the question marks, to get this:

$$\alpha :: *, \beta :: * \vdash \alpha :: * \tag{49.3}$$

How do we do this? We can work backwards. Let's start by putting the judgment at the bottom of a proof tree:

$$\frac{}{\alpha :: *, \beta :: * \vdash \alpha :: *}$$

This is the result of type weakening, so we need two premises above it:

$$\frac{\begin{array}{cc} ? & ? \\ \vdots & \vdots \end{array}}{\alpha :: *, \beta :: * \vdash \alpha :: *}$$

On the left, we need the same judgment that we derive at the bottom, but without the weakening declaration $\beta :: *$.

$$\dfrac{\dfrac{\overset{?}{\vdots}}{\alpha :: * \vdash \alpha :: *} \qquad \overset{?}{\vdots}}{\alpha :: *, \beta :: * \vdash \alpha :: *}$$

On the right, we need the same context (without the weakening $\beta :: *$), and we need the judgment to be $* :: \Box$.

$$\dfrac{\dfrac{\overset{?}{\vdots}}{\alpha :: * \vdash \alpha :: *} \qquad \dfrac{\overset{?}{\vdots}}{\alpha :: * \vdash * :: \Box}}{\alpha :: *, \beta :: * \vdash \alpha :: *}$$

If we can derive those two premises, then the bottom line will be a valid derivation through *type weakening*.

Let's turn to the judgment on the left. It is this:

$$\alpha :: * \vdash \alpha :: * \tag{49.4}$$

How do we derive this? We can use *type-var intro* and *sort*:

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *} \qquad \dfrac{\overset{?}{\vdots}}{\alpha :: * \vdash * :: \Box}}{\alpha :: *, \beta :: * \vdash \alpha :: *}$$

Now let's turn to the judgment on the right. It is this:

$$\alpha :: * \vdash * :: \Box \tag{49.5}$$

How do we derive that? This looks like the kind of thing we discussed in the last chapter. We have a judgment which asserts that $* :: \Box$, and its context is weaken with $\alpha :: *$. We can derive that using the technique from the last chapter: we use *sort* twice, then we weaken the one on the left with $\alpha :: *$:

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \qquad \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash * :: \Box}}{\alpha :: *, \beta :: * \vdash \alpha :: *}$$

That completes the derivation. We have now weakened a statement $\alpha :: * \vdash \alpha :: *$ with the declaration $\beta :: *$.

49.2 Weakening Type Judgments Again

Suppose we wanted to weaken the above judgment again. That is, suppose we want to take $\alpha :: *, \beta :: * \vdash \alpha :: *$, and add $\gamma :: *$ to the context, so as to get this:

$$\alpha :: *, \beta :: *, \gamma :: * \vdash \alpha :: * \tag{49.6}$$

The technique is just as we did above, just with some more complication on the right side. Let's work backwards again. Let's begin with the judgment we want to derive, and put that at the bottom of a proof tree.

$$\dfrac{}{\alpha :: *, \beta :: *, \gamma :: * \vdash \alpha :: *}$$

To get this, we need to start with $\alpha :: *, \beta :: * \vdash \alpha :: *$, and then weaken it. So we need two premises above it:

$$\dfrac{\dfrac{\begin{array}{c}?\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash \alpha :: *} \qquad \dfrac{\begin{array}{c}?\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box}}{\alpha :: *, \beta :: *, \gamma :: * \vdash \alpha :: *}$$

We know how to get the judgment on the left, since we did it in the last section. We could paste in the whole derivation above that judgment, on the left branch:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \qquad \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash * :: \Box}}{\alpha :: *, \beta :: * \vdash \alpha :: *} \qquad \dfrac{\begin{array}{c}?\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box}}{\alpha :: *, \beta :: *, \gamma :: * \vdash \alpha :: *}$$

But to save space, let's replace that derivation with QED and vertical dots:

$$\dfrac{\dfrac{\begin{array}{c}QED\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash \alpha :: *} \qquad \dfrac{\begin{array}{c}?\\ \vdots\end{array}}{\alpha :: *, \beta :: * \vdash * :: \Box}}{\alpha :: *, \beta :: *, \gamma :: * \vdash \alpha :: *}$$

That leaves the right side of the tree. How do we derive $\alpha :: *, \beta :: * \vdash * :: \Box$? This requires the same technique we used in the last chapter: we weaken judgments with $* :: \Box$ on the right. We already worked out this derivation above, so we can just graft it onto the branch here:

$$\dfrac{\dfrac{\overset{QED}{\vdots}}{\alpha :: *, \beta :: * \vdash \alpha :: *} \qquad \dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \square} \quad \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash * :: \square} \quad \dfrac{\dfrac{}{\varnothing \vdash * :: \square} \quad \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash * :: \square}}{\alpha :: *, \beta :: * \vdash * :: \square}}{\alpha :: *, \beta :: *, \gamma :: * \vdash \alpha :: *}$$

We could repeat this technique to add more type declarations into the context, e.g., $\gamma :: *$, $\sigma \to \tau :: *$, and so on.

49.3 Adding a Term Variable Declaration

Suppose we have a judgment of $* :: \square$, with one or more type declarations in the context. Something like this:

$$\alpha :: *, \beta :: * \vdash * :: \square \tag{49.7}$$

Suppose we want to weaken the context with a term variable of type α or β. For instance, suppose we want to add $x : \alpha$ to the context, to make this judgment:

$$\alpha :: *, \beta :: *, x : \alpha \vdash * :: \square$$

How do we do that? Term weakening is the rule, but we need to perform some other derivations first, to set up the premises correctly.

Recall that the *term weakening* rule to introduce a term of type α looks like this:

$$\dfrac{\Gamma \vdash A : B \qquad \Gamma \vdash \alpha :: *}{\Gamma, x : \alpha \vdash A : B}$$

In place of $A : B$, we want $* :: \square$, so let's fill that in:

$$\dfrac{\Gamma \vdash * :: \square \qquad \Gamma \vdash \alpha :: *}{\Gamma, x : \alpha \vdash * :: \square}$$

The context we want to weaken is $\alpha :: *, \beta :: *$, so let's add that in:

$$\dfrac{\alpha :: *, \beta :: * \vdash * :: \square \qquad \alpha :: *, \beta :: * \vdash \alpha :: *}{\alpha :: *, \beta :: *, x : \alpha \vdash * :: \square}$$

We know how to derive the judgment on the left. We did it at the end of the last chapter. We can fill it in:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \square} \quad \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash * :: \square} \quad \dfrac{\dfrac{}{\varnothing \vdash * :: \square} \quad \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash * :: \square}}{\alpha :: *, \beta :: * \vdash * :: \square} \qquad \alpha :: *, \beta :: * \vdash \alpha :: *}{\alpha :: *, \beta :: *, x : \alpha \vdash * :: \square}$$

But that takes up a lot of space, so let's just use QED and vertical dots to show that we've derived it:

$$\dfrac{\dfrac{\begin{matrix} QED \\ \vdots \end{matrix}}{\alpha :: *, \beta :: * \vdash * :: \square} \qquad \alpha :: *, \beta :: * \vdash \alpha :: *}{\alpha :: *, \beta :: *, x : \alpha \vdash * :: \square}$$

The question now is, how do we derive the statement on the right side? Let's put vertical dots and a question mark to show what we need to fill in:

$$\dfrac{\dfrac{\begin{matrix} QED \\ \vdots \end{matrix}}{\alpha :: *, \beta :: * \vdash * :: \square} \qquad \dfrac{\begin{matrix} ? \\ \vdots \end{matrix}}{\alpha :: *, \beta :: * \vdash \alpha :: *}}{\alpha :: *, \beta :: *, x : \alpha \vdash * :: \square}$$

We know how to derive this side too. We did it in the last section. We can paste the whole derivation in:

$$\dfrac{\dfrac{\begin{matrix} QED \\ \vdots \end{matrix}}{\alpha :: *, \beta :: * \vdash * :: \square} \qquad \dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash \alpha :: *} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \square} \qquad \dfrac{}{\varnothing \vdash * :: \square}}{\alpha :: * \vdash * :: \square}}{\alpha :: *, \beta :: * \vdash \alpha :: *}}{\alpha :: *, \beta :: *, x : \alpha \vdash * :: \square}$$

And there we have it. By putting together the other derivations we learned, we can introduce a declaration like $x : \alpha$ into the context.

As we mentioned before, $\lambda\omega$ is quite strict in what it lets you introduce into the context. But there is a nice level of rigor here. Types and typed values can only enter into the picture if they are legally derived.

Chapter 50

Forming Arrow Types

So far, we have learned how to expand the context with single-character type variables like $\alpha :: *$ or single-character term variables like $x : \alpha$. But we also have arrow types. We have arrow kinds like $* \to *$, and we have arrow types like $\alpha \to \beta$.

There is one rule called **formation** that is often used to derive both types of arrows. But we will split it up into two distinct rules, to keep things separate.

50.1 Kind arrows

The **kind-arrow formation** rule has this form:

$$\frac{\Gamma \vdash A :: \Box \qquad \Gamma \vdash B :: \Box}{\Gamma \vdash A \to B :: \Box}$$

What this says is that if we derive a kind A in a context Γ, and then we also derive another kind B in that same context Γ, we can derive the arrow kind made from A and B.

Note that A and B can be the same kinds. For instance, suppose we have derived $* :: \Box$ twice. So now, we have two judgments of this form:

$$\begin{array}{cc} \vdots & \vdots \\ \Gamma \vdash * :: \Box & \Gamma \vdash * :: \Box \end{array}$$

We can then use the *kind-arrow form* rule to construct the kind $* \to * :: \Box$:

$$\frac{\begin{array}{cc} \vdots & \vdots \\ \Gamma \vdash * :: \Box & \Gamma \vdash * :: \Box \end{array}}{\Gamma \vdash * \to * :: \Box}$$

Here is a simple example. Suppose we use *sort* twice:

$$\textit{sort}\ \frac{}{\varnothing \vdash * :: \Box} \qquad \frac{}{\varnothing \vdash * :: \Box}\ \textit{sort}$$

We can then use the *kind-arrow form* rule to construct $* \to * :: \Box$:

$$\textit{kind-arrow form}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \Box} \qquad \dfrac{}{\varnothing \vdash * :: \Box}\ \textit{sort}}{\varnothing \vdash * \to * :: \Box}$$

Here is another example. The *QED* and vertical dots indicate derivations that we put together in the previous chapters:

$$\dfrac{\begin{array}{c} QED \\ \vdots \\ \alpha :: * \vdash * :: \Box \end{array} \qquad \begin{array}{c} QED \\ \vdots \\ \alpha :: * \vdash * :: \Box \end{array}}{\alpha :: * \vdash * \to * :: \Box}$$

Here is another example, where the second kind is itself an arrow kind:

$$\dfrac{\begin{array}{c} QED \\ \vdots \\ \alpha :: * \vdash * :: \Box \end{array} \qquad \dfrac{\begin{array}{c} QED \\ \vdots \\ \alpha :: * \vdash * :: \Box \end{array} \qquad \begin{array}{c} QED \\ \vdots \\ \alpha :: * \vdash * :: \Box \end{array}}{\alpha :: * \vdash * \to * :: \Box}}{\alpha :: * \vdash * \to (* \to *) :: \Box}$$

50.2 Type arrows

The **type-arrow formation** rule has this form (where α and β can stand for any types from $\mathbb{T}$):

$$\frac{\Gamma \vdash \alpha : * \qquad \Gamma \vdash \beta : *}{\Gamma \vdash \alpha \to \beta : *}$$

This says that if we derive a type $\alpha : *$ in a context Γ, and then we also derive another type $\alpha : *$ in that same context Γ, we can derive the arrow kind $\alpha \to \beta : *$ made from $\alpha : *$ and $\beta : *$.

Here is a simple example. Suppose we use *sort* and *type-var intro* to generate $\alpha :: * \vdash \alpha :: *$ twice:

$$\dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *} \qquad \dfrac{}{\alpha :: * \vdash \alpha :: *}\ \textit{sort}$$

We can then use the *type-arrow form* rule to construct $\alpha \to \alpha :: *$:

$$\begin{array}{c} \dfrac{\textit{sort}\ \overline{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *} \qquad \overline{\alpha :: * \vdash \alpha :: *}\ \textit{sort} \\ \alpha :: * \vdash \alpha \to \alpha :: * \end{array}$$

Here is another example. The *QED* and vertical dots indicate derivations that we put together in the previous chapters:

$$\dfrac{\begin{array}{c} QED \\ \vdots \\ \alpha :: *, \beta :: * \vdash \beta :: * \end{array} \qquad \begin{array}{c} QED \\ \vdots \\ \alpha :: *, \beta :: * \vdash \beta :: * \end{array}}{\alpha :: *, \beta :: * \vdash \beta \to \beta :: *}$$

Here is another example, where the second kind is itself an arrow kind:

$$\dfrac{\begin{array}{c} QED \\ \vdots \\ \alpha :: *, \beta :: * \vdash \alpha :: * \end{array} \qquad \dfrac{\begin{array}{c} QED \\ \vdots \\ \alpha :: *, \beta :: * \vdash \beta :: * \end{array} \qquad \begin{array}{c} QED \\ \vdots \\ \alpha :: *, \beta :: * \vdash \beta :: * \end{array}}{\alpha :: *, \beta :: * \vdash \beta \to \beta :: *}}{\alpha :: *, \beta :: * \vdash \alpha \to (\beta \to \beta) :: *}$$

Chapter 51

Term Abstraction

In $\lambda\omega$, we have two kinds of abstraction. We can abstract over terms, as we have been able to do with $\lambda^{\rightarrow}$ and $\lambda 2$. But we can also abstract over types to form new types.

Often, authors will formulate a single rule that can apply to both kinds of abstraction. But as usual, we will split this out into two rules: one for term abstraction, and one for type abstraction.

51.1 About Term Abstraction

Think about the **abstraction** rule from $\lambda^{\rightarrow}$ and $\lambda 2$. The basic form of the rule is as follows (where M and N can stand for any pretyped term, and α and β can stand for any type in $\mathbb{T}$):

$$\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x : \alpha . M : \alpha \rightarrow \beta}$$

This says that if we have derived a legal term M of type β, and we also have an available variable x of type α in the context, then we can mark x as replaceable in M.

Of course, to mark x as replaceable in M, we prefix M with $\lambda x : \alpha$. This forms a new term, which we call an abstraction term. The type of this new abstraction term is $\alpha \rightarrow \beta$.

51.2 Term Abstraction

There is an important thing to notice about the *abstr* rule. When you use it to construct an abstraction, the abstraction has a new type that wasn't in the premises. To confirm this, look at the rule itself:

$$\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x : \alpha . M : \alpha \rightarrow \beta}$$

Above the line, we can see two types. First, we can see α (which is the type of x), and we can see β (which is the type of M). But we can't see the type $\alpha \to \beta$ anywhere.

With $\lambda\omega$, we are very strict with the types that enter into derivations. So we can't use this type $\alpha \to \beta$ unless we are sure that it can be derived. That is, we need to be sure of the following (where K is some kind in $\mathbb{K}$):

$$\Gamma \vdash \alpha \to \beta : K \tag{51.1}$$

Hence, we need to add that as a second premise to the *abstr* rule. So, here is the full **term abstraction rule**, for $\lambda\omega$:

$$\frac{\Gamma, x : \alpha \vdash M : \beta \qquad \Gamma \vdash \alpha \to \beta : K}{\Gamma \vdash \lambda x : \alpha.M : \alpha \to \beta}$$

This says that given a context Γ, *if* we have a term M of type β along with an available variable x of type α, *and* if the type $\alpha \to \beta$ is a legal type in the same context Γ, *then* we can mark $x : \alpha$ as replaceable in M, to form an abstraction term of type $\alpha \to \beta$.

51.2.1 An Example

Let's derive the identity function. In untyped λ, the identity function is this abstraction:

$$\lambda x.x \tag{51.2}$$

It takes some input (whatever you want to replace x with), and it returns that same output (whatever you replace x with).

Here though, we are working with types, so we need to add types to this identity function. Let's suppose that x has the type α. So we can add the type α to the bound variable:

$$\lambda x : \alpha.x \tag{51.3}$$

We also know the type of the whole abstraction. Abstractions are arrow types that go from the type of the bound variable to the type of the body. In this case, the type of the bound variable x is α. The body of this expression is also x, so it has the same type too, namely α. Hence, the type of the whole expression is $\alpha \to \alpha$:

$$\lambda x : \alpha.x : \alpha \to \alpha \tag{51.4}$$

And with that, we have a fully typed version of the identity function. This is what we want to derive for our example, using the rules of $\lambda\omega$.

Let's work backwards. First, we want to formulate our final statement as a judgment. There must be some context on the left. We don't know it yet, so let's just put question marks there. Let's also put question marks above the inference to mark what we need to figure out:

$$\dfrac{\begin{array}{c} ?? \\ \vdots \end{array}}{?? \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

In order to derive this abstraction, we need two premises before:

$$\dfrac{\begin{array}{c} ?? \\ \vdots \end{array} \qquad \begin{array}{c} ?? \\ \vdots \end{array}}{?? \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

Let's do the one on the left first. If we look at the *term abstr* rule, we can see that we need to remove the λ abstraction and move $x : \alpha$ to the context. That gives us this:

$$\dfrac{\dfrac{\begin{array}{c} ?? \\ \vdots \end{array}}{??, x : \alpha \vdash x : \alpha} \qquad \begin{array}{c} ?? \\ \vdots \end{array}}{?? \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

We also know that if we're going to use the variable x with type α, we need to declare the type $\alpha :: *$ as well. So we can add that to the context on the left too:

$$\dfrac{\dfrac{\begin{array}{c} ?? \\ \vdots \end{array}}{\alpha :: *, x : \alpha \vdash x : \alpha} \qquad \begin{array}{c} ?? \\ \vdots \end{array}}{?? \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

Beyond x and α, there are no other primitive variables or types in this expression. So let's assume for now that our context is complete. We can fill in the question marks on the bottom line:

$$\dfrac{\dfrac{\begin{array}{c} ?? \\ \vdots \end{array}}{\alpha :: *, x : \alpha \vdash x : \alpha} \qquad \begin{array}{c} ?? \\ \vdots \end{array}}{\alpha :: * \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

Now let's turn to the right hand side. If we look at the *term abstr* rule, it says that we need to derive the type $\alpha \to \alpha$ in the same context (minus $x : \alpha$). So let's put that as the premise on the right:

$$\dfrac{\dfrac{\begin{array}{c}??\\ \vdots\end{array}}{\alpha :: *, x : \alpha \vdash x : \alpha} \qquad \dfrac{\begin{array}{c}??\\ \vdots\end{array}}{\alpha :: * \vdash \alpha \to \alpha :: *}}{\alpha :: * \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

Now that we have established which two premises must come before our desired conclusion, we can turn to those premises. Our next task is to work out how to derive each of them. Let's do the one on the left first.

If you think about this one, what we see here is a context $\alpha :: *$, with a term variable $x : \alpha$ introduced on both sides of the $\vdash$ symbol. So this looks like an instance of the *term var intro* rule. Let's fill that in:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}??\\ \vdots\end{array}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *, x : \alpha \vdash x : \alpha} \qquad \dfrac{\begin{array}{c}??\\ \vdots\end{array}}{\alpha :: * \vdash \alpha \to \alpha :: *}}{\alpha :: * \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

On the left, we now have $\alpha :: * \vdash \alpha :: *$. How do we get that? Here we have a type $\alpha :: *$ introduced on both sides of the $\vdash$ symbol. So this looks like an instance of the *type var intro* rule, so let's fill that in:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *, x : \alpha \vdash x : \alpha} \qquad \dfrac{\begin{array}{c}??\\ \vdots\end{array}}{\alpha :: * \vdash \alpha \to \alpha :: *}}{\alpha :: * \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

Here it is again, but with the inferences labeled, so it's clear exactly how we got this:

$$\dfrac{\textit{term var intro}\ \dfrac{\textit{type var intro}\ \dfrac{\textit{sort}\ \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *, x : \alpha \vdash x : \alpha} \qquad \dfrac{\begin{array}{c}??\\ \vdots\end{array}}{\alpha :: * \vdash \alpha \to \alpha :: *}}{\alpha :: * \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

Now let's turn to the right side. We need to derive $\alpha :: * \vdash \alpha \to :: *$. How do we derive that?

We can use the *type arrow form* rule. The *type arrow form* rule says that if we derive a type $A :: K$ and we also derive a type $B :: K$, then we can derive an arrow from the two: $A \to B :: K$.

In this case, the arrow is $\alpha \to \alpha :: *$, so we want to derive $\alpha :: *$ and then again $\alpha :: *$. Let's fill in those two premises:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *, x : \alpha \vdash x : \alpha} \qquad \dfrac{\dfrac{\overset{??}{\vdots}}{\alpha :: * \vdash \alpha :: *} \quad \dfrac{\overset{??}{\vdots}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: * \vdash \alpha \to \alpha :: *}}{\alpha :: * \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

Now we need to derive $\alpha :: * \vdash \alpha :: *$. How do we derive that? We can use the *type var intro* rule. Here is the derivation, with the inference rules labeled:

$$\text{kind arrow form}\ \dfrac{\text{type var intro}\ \dfrac{\text{sort}\ \dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box}\ \text{sort}}{\alpha :: * \vdash \alpha :: *}\ \text{type var intro}}{\alpha :: * \vdash \alpha \to \alpha :: *}$$

Let's graft that derivation onto our tree (with labels removed, to save space):

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *, x : \alpha \vdash x : \alpha} \qquad \dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *} \quad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: * \vdash \alpha \to \alpha :: *}}{\alpha :: * \vdash \lambda x : \alpha . x : \alpha \to \alpha}$$

That completes the derivation.

Chapter 52

Type Abstraction

Recall that in $\lambda\omega$, we have two kinds of abstraction: we can abstract over terms, and we can abstract over types. We covered term abstraction in the last chapter. In this chapter, we cover type abstraction for $\lambda\omega$.

52.1 The Type Abstraction Rule

Type abstraction is similar to term abstraction, except that with type abstraction, we abstract over types instead of terms. Let's build up the rule piece by piece.

Suppose we have a context Γ. Suppose also that we can derive in this context some type, call it β, which has kind K_2. So we have this:

$$\Gamma \vdash \beta :: K_2 \tag{52.1}$$

Suppose that we also have in the context, another type, which we can call α, of kind K_1:

$$\Gamma, \alpha :: K_1 \vdash \beta :: K_2 \tag{52.2}$$

We can mark α as replaceable in β, by using a lambda binding expression. That would give us this:

$$\Gamma \vdash \lambda\alpha :: K_1.\beta \tag{52.3}$$

The kind of this is going to be $K_1 \to K_2$, since the type (or kind) of any abstraction is an arrow from the type (or kind) of the bound symbol to the type (or kind) of the body. And in this case, the bound symbol's kind is K_1, while the body's kind is K_2. Hence, the type of the whole abstraction:

$$\Gamma \vdash \lambda\alpha :: K_1.\beta : K_1 \to K_2 \tag{52.4}$$

Of course, since $\lambda\omega$ is very strict about what we introduce into the context, we need to make sure that we can derive the kind $K_1 \to K_2$ in the context Γ as well. So we also need to ensure that we can derive this judgment:

$$\Gamma \vdash K_1 \to K_2 :: \Box \tag{52.5}$$

We can put that all together, to formulate our **kind abstraction rule** (where α and β stand for any types from $\mathbb{T}$, and let K_1 and K_2 stand for any kinds from $\mathbb{K}$):

$$\frac{\Gamma, \alpha :: K_1 \vdash \beta :: K_2 \qquad \Gamma \vdash K_1 \to K_2 :: \Box}{\Gamma \vdash \lambda\alpha :: K_1.\beta :: K_1 \to K_2}$$

This says that given a context Γ, if we have in the context a type α of kind K_1, and we can derive another type β of kind K_2, and if we can also derive $K_1 \to K_2$, then we can mark α as replaceable in β, and form a type abstraction.

52.1.1 Example

Let us derive this abstraction:

$$\beta :: * \vdash \lambda\alpha :: *.\alpha \to \alpha :: * \to * \tag{52.6}$$

Let's work backwards. To begin, let's put this as our goal, at the bottom of a proof tree:

$$\frac{}{\beta :: * \vdash \lambda\alpha :: *.\alpha \to \alpha :: * \to *}$$

If we look at the *type abstr* rule, we know that we need two premises before this:

$$\frac{\beta :: *, \alpha :: * \vdash \alpha \to \alpha :: * \qquad \beta :: * \vdash * \to * :: \Box}{\beta :: * \vdash \lambda\alpha :: *.\alpha \to \alpha :: * \to *}$$

The next question is, how do we derive each of those premises?

$$\frac{\dfrac{\begin{array}{c} ?? \\ \vdots \end{array}}{\beta :: *, \alpha :: * \vdash \alpha \to \alpha :: *} \qquad \dfrac{\begin{array}{c} ?? \\ \vdots \end{array}}{\beta :: * \vdash * \to * :: \Box}}{\beta :: * \vdash \lambda\alpha :: *.\alpha \to \alpha :: * \to *}$$

Let's turn to the derivation on the left. Our target is this:

$$\frac{\begin{array}{c} ?? \\ \vdots \end{array}}{\beta :: *, \alpha :: * \vdash \alpha \to \alpha :: *}$$

How do we derive that? We can see an arrow kind on the right. So we know that we should use the *kind arrow form* rule. That rule tells us that if we want to derive $A \to B :: K$, we need to derive $A :: K$ and $B :: K$. Hence, we can fill in the premises:

$$\dfrac{\dfrac{\begin{matrix}??\\ \vdots\end{matrix}}{\beta :: *, \alpha :: * \vdash \alpha :: *} \qquad \dfrac{\begin{matrix}??\\ \vdots\end{matrix}}{\beta :: *, \alpha :: * \vdash \alpha :: *}}{\beta :: *, \alpha :: * \vdash \alpha \to \alpha :: *}$$

How do we derive the judgment on the left? Well, we can see a declaration of $\alpha :: *$ on both sides of the $\vdash$ symbol. So this is an instance of the *type var intro* rule. So let's fill in the premise for that:

$$\dfrac{\dfrac{\dfrac{\begin{matrix}??\\ \vdots\end{matrix}}{\beta :: * \vdash * :: \square}}{\beta :: *, \alpha :: * \vdash \alpha :: *} \qquad \dfrac{\begin{matrix}??\\ \vdots\end{matrix}}{\beta :: *, \alpha :: * \vdash \alpha :: *}}{\beta :: *, \alpha :: * \vdash \alpha \to \alpha :: *}$$

Now we need to figure out how to derive $\beta :: * \vdash * :: \square$. How do we get that? This is an instance of weakening an empty context with $\beta :: *$. So we can fill in the derivation for that:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \square} \quad \dfrac{}{\varnothing \vdash * :: \square}}{\beta :: * \vdash * :: \square}}{\beta :: *, \alpha :: * \vdash \alpha :: *} \qquad \dfrac{\begin{matrix}??\\ \vdots\end{matrix}}{\beta :: *, \alpha :: * \vdash \alpha :: *}}{\beta :: *, \alpha :: * \vdash \alpha \to \alpha :: *}$$

We can then copy that over to the right side, since the derivation is the same:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \square} \quad \dfrac{}{\varnothing \vdash * :: \square}}{\beta :: * \vdash * :: \square}}{\beta :: *, \alpha :: * \vdash \alpha :: *} \qquad \dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \square} \quad \dfrac{}{\varnothing \vdash * :: \square}}{\beta :: * \vdash * :: \square}}{\beta :: *, \alpha :: * \vdash \alpha :: *}}{\beta :: *, \alpha :: * \vdash \alpha \to \alpha :: *}$$

Now that we know how to derive this, let's abbreviate the whole derivation with a *QED* and vertical dots:

$$\begin{matrix}QED\\ \vdots\\ \beta :: *, \alpha :: * \vdash \alpha \to \alpha :: *\end{matrix}$$

And we can put that back into our original derivation:

$$\dfrac{\dfrac{\overset{QED}{\vdots}}{\beta :: *, \alpha :: * \vdash \alpha \to \alpha :: *} \qquad \dfrac{\overset{??}{\vdots}}{\beta :: * \vdash * \to * :: \Box}}{\beta :: * \vdash \lambda\alpha :: *.\alpha \to \alpha :: * \to *}$$

Now we need to work out the derivation on the right hand side. Our target is this:

$$\dfrac{\overset{??}{\vdots}}{\beta :: * \vdash * \to * :: \Box}$$

How do we derive this? We can see an arrow kind on the right side of the $\vdash$ symbol, so this must be an instance of the *kind arrow form* rule. If we look at that rule, we can see that it requires two premises. We need to derive $\beta :: * \vdash * :: \Box$ twice:

$$\dfrac{\dfrac{\overset{??}{\vdots}}{\beta :: * \vdash * :: \Box} \qquad \dfrac{\overset{??}{\vdots}}{\beta :: * \vdash * :: \Box}}{\beta :: * \vdash * \to * :: \Box}$$

How do we derive $\beta :: * \vdash * :: \Box$? This is an instance where we weaken the empty context with $\beta :: *$. So let's fill that in:

$$\dfrac{\dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\beta :: * \vdash * :: \Box} \qquad \dfrac{\dfrac{}{\varnothing \vdash * :: \Box} \quad \dfrac{}{\varnothing \vdash * :: \Box}}{\beta :: * \vdash * :: \Box}}{\beta :: * \vdash * \to * :: \Box}$$

Now that we know how to derive this, let's abbreviate it with a QED and vertical dots:

$$\dfrac{\overset{QED}{\vdots}}{\beta :: * \vdash * \to * :: \Box}$$

And we can graft it back into our original derivation:

$$\dfrac{\dfrac{\overset{QED}{\vdots}}{\beta :: *, \alpha :: * \vdash \alpha \to \alpha :: *} \qquad \dfrac{\overset{QED}{\vdots}}{\beta :: * \vdash * \to * :: \Box}}{\beta :: * \vdash \lambda\alpha :: *.\alpha \to \alpha :: * \to *}$$

And with that, we have completed the derivation.

Chapter 53

Term Application

In addition to (term and type) abstraction, we also have (term and type) application in $\lambda\omega$. Often, authors present one general rule that covers both cases. But as usual, we will treat term and type application as separate rules.

53.1 The Term Application Rule

If we have an abstraction, we can apply it to some other term, provided that the other term has the right type. Since an abstraction has an arrow type, e.g., $A \to B$, this means that the term we apply it to must have type A.

The rule can be formulated as follows, where M and N are any valid pretyped terms, and α and β are any types from $\mathbb{T}$:

$$\frac{\Gamma \vdash \lambda x : \alpha . M : \alpha \to \beta \qquad \Gamma \vdash N : \alpha}{\Gamma \vdash (\lambda x : \alpha . M) N : \beta}$$

53.1.1 Example

TBD

Chapter 54

Type Application

TBD

Chapter 55

Conversion

TBD

Chapter 56

Dependent Terms and Types

We will turn to our next type of calculus, called λP. Before we do that, let's talk about the notion of dependency.

The fundamental mechanism of computation in any form of lambda calculus occurs through abstraction, application, and β-reduction. Using these operations, we can take an expression, mark certain parts of it as replaceable, then we can stipulate a replacement, and finally we can replace the marked parts with the stipulated replacement. This results in a new expression (one with new values in place of certain of its parts).

There is an important point to notice here: the new expression depends on the value of the replacement. If we substitute an x in, we get one expression. If we substitute a y in, we get a different expression. So these constructed expressions depend on the values we substitute in.

We have seen a variety of different types of dependency. We have seen terms that depend on terms, terms that depend on types, and types that depend on types. There are also types that depend on terms. Let's look at each of these in turn.

56.1 Terms Depending on Terms

In the untyped λ, consider this β-reduction:

$$(\lambda x.xy)(z) =_\beta zy \tag{56.1}$$

Here we start with xy, then we replace x with z, which gives us zy.

Note that zy has an **identity**. It is precisely the term zy, rather than some other term, like uy.

Let's put this another way: zy' s identity depends on the fact that it is built from a z in its first slot. If it were built from, say, a u in its first slot, it would be a different term (it would be uy).

To describe this phenomenon, we say that zy **depends** on z. And since zy and z are both terms, we say that this is a case where a **term depends on a term**.

This occurs in the untyped λ and in $\lambda^{\rightarrow}$, whenever we apply β-reduction. In each case of a β-reduction, the resulting term depends on the term we substitute in to get the result.

56.2 Terms Depending on Types

In $\lambda 2$, consider this β-reduction:

$$\Pi\alpha.(\lambda x : \alpha.x)(\beta) =_{\beta} (\lambda x : \beta.x) \tag{56.2}$$

Here we start with an identity function for inhabitants of type α: $\lambda x : \alpha.x$. Then we mark α replaceable, which gives us: $\Pi\alpha.(\lambda x : \alpha.x)$. The type of this is $\Pi\alpha.(\alpha \to \alpha)$. Then we apply it to the type β. When we do the β-reduction, we replace α with β, which gives us this result: $\lambda x : \beta.x$. The type of this result is $\beta \to \beta$.

The result here is a term (it is not a type). $\lambda x : \beta.x$ is nothing more than a typed lambda term. However, it's identity depends on the type β. If β were replaced with a different type, say γ, then it would be a different term (it would be $\lambda x : \gamma.x$).

So $\lambda x : \beta.x$ **depends** on β. And since $\lambda x : \beta.x$ is a term while β is a type, we say that this is a case where a **term depends on a type**.

As our example makes clear, this sort of term-depending-on-a-type phenomenon occurs in $\lambda 2$, whenever we reduce Π-type applications.

56.3 Types Depending on Types

In $\lambda\underline{\omega}$, consider this β-reduction:

$$(\lambda\alpha :: *.\alpha \to \alpha)(\beta) =_{\beta} \lambda\beta :: *.\beta \to \beta \tag{56.3}$$

Here we start with a type $\alpha \to \alpha$, then we mark the α as replaceable, which gives us: $\lambda\alpha :: *.\alpha \to \alpha$. Then we apply that to β. When we replace α with β, we get $\lambda\beta :: *.\beta \to \beta$.

The result here is a type, and it's identity depends on the fact that it's built from β. If it were built from a different type, say γ, it would be a different term (it would be $\lambda\gamma :: *.\gamma \to \gamma$).

So $\lambda\beta :: *.\beta \to \beta$ **depends** on β. And since $\lambda\beta :: *.\beta \to \beta$ and β are both types, we say that this is a case where a **type depends on a type**.

This kind of dependency occurs in $\lambda\underline{\omega}$, whenever we perform β reduction on types.

56.4 Types Depending on Terms

In λP, we have types that are constructed from both a type and a term. For example, suppose that β is a type, and x is a term of type α. In λP, we can have a type like this:

$$\beta x \tag{56.4}$$

This is a type that is built from a type (β), *and* a term (x).

Of course, we can then mark the x as replaceable. We use the Π binder for that:

$$\Pi x : \alpha.(\beta x) \tag{56.5}$$

Now we can stipulate a replacement. We can say, for example, that we want to replace x with another term of type α, for instance y:

$$\Pi x : \alpha.(\beta x)(y : \alpha) \tag{56.6}$$

Then when we actually replace the x with the y (i.e., when we do a β-reduction), we get this:

$$\beta y \tag{56.7}$$

And of course, βy's identity depends on the fact that is built from y. After all, if it were built from another term, say z, it would be a different type (it would be βz.

So βy **depends** on y. And since β is a type and y is a term, we say that this is a case where a **type depends on a term**.

This kind of dependency occurs in λP, which we will study next.

56.5 Dependency in General

We can put all of this more generally: when you have an expression E_2, which is built by substituting another expression E_1 in it, we say that E_2 **depends** on E_1.

Since either of E_1 or E_2 can be a type or a term, we have a matrix of four possibilities:

- If E_1 is a term and E_2 is a term, then we have a case of **a term depending on a term** (we have this in λ and $\lambda^{\to}$).
- If E_1 is a type and E_2 is a term, then we have a case of **a term depending on a type** (we have this in $\lambda 2$).
- If E_1 is a type and E_2 is a type, then we have a case of **a type depending on a type** (we have this in $\lambda\underline{\omega}$.
- If E_1 is a term and E_2 is a type, then we have a case of **a type depending on a term** (we have this in λP.

Chapter 57

Sort, Var, and Weaken

In λP, the *sort*, *var*, and *weaken* rules are the same as they were in $\lambda\underline{\omega}$. Let us review these rules, and put them in a slightly different format.

57.1 Some Notation

Let us introduce some notation to help represent these rules in a more visual way.

- $\square$ — This will continue to stand for the super kind, i.e., the kind of all kinds.
- K — This can stand for any **k**ind, including $\square$, e.g., $*$, $* \to *$, $\square$, etc.
- T — This can stand for any **t**ype, e.g., α, β, $\alpha \to \beta$.
- t — This can stand for any **t**erm, e.g., x, y, etc.
- $\langle\ \ldots\ \rangle$ — This can stand for any context.
- $\langle\ \rangle$ — This will stand for an empty context.
- $\langle\ \ldots\ \rangle,\ E$ — This indicates that E is added to the context, so that the resulting context is $\langle\ \ldots,\ E\ \rangle$.
- $[\ ]$ — This can stand for any expression. Think of it as a slot that holds some expression.
- $\boxed{??}$ — This will stand for something we don't know yet, but need to fill in.
- $\boxed{??K}$ — This will stand for a kind we don't know yet, but need to fill in.
- $\boxed{??T}$ — This will stand for a type we don't know yet, but need to fill in.
- $\boxed{??t}$ — This will stand for a term we don't know yet, but need to fill in.

57.2 The *Sort* Rule

The sort rule looks like this:

$$\text{Sort}\ \frac{}{\langle\ \rangle \vdash * :: \square}$$

It says that if you have an empty context, you can assign the type $* :: \square$.

We can also write that like this:

$$\text{Sort}\ \frac{}{\varnothing \vdash * :: \square}$$

57.3 The *Var* Rule

In a generic form, the *var* rule (also called the *start* rule) is often stated like this:

$$\text{Var/Start}\ \frac{\Gamma \vdash A : s}{\Gamma,\ x : A \vdash x : A}$$

In this formulation, s refers to either $*$ or $\square$, A refers to either a type or a kind, and x refers to an instance of A. Γ, of course, refers to a context.

We break this out into two rules, one for types, and one for terms.

57.3.1 The *Type Var* Rule

Using our notation, the *type var* rule looks like this:

$$\text{Type var}\ \frac{\langle\ \ldots\ \rangle \vdash K :: \square}{\langle\ \ldots\ \rangle,\ T :: K \vdash T :: K}$$

When we use this rule, here are the parts we want to fill in:

$$\text{Type var}\ \frac{\langle\ \ldots\ \rangle \vdash K :: \square}{\langle\ \ldots\ \rangle,\ \boxed{??T} :: K \vdash \boxed{??T} :: K}$$

For instance, suppose K is $*$.

$$\text{Type var}\ \frac{\langle\ \ldots\ \rangle \vdash * :: \square}{\langle\ \ldots\ \rangle,\ \boxed{??T} :: * \vdash \boxed{??T} :: *}$$

With the *type var* rule, we can introduce any type T of kind $*$. For instance, we could introduce a type α:

$$\text{Type var}\ \frac{\langle\ \ldots\ \rangle \vdash * :: \square}{\langle\ \ldots\ \rangle,\ \boxed{\alpha} :: * \vdash \boxed{\alpha} :: *}$$

Or β:

$$\text{Type var}\ \frac{\langle\ \ldots\ \rangle \vdash * :: \square}{\langle\ \ldots\ \rangle,\ \boxed{\beta} :: * \vdash \boxed{\beta} :: *}$$

57.3.2 The *Term Var* Rule

The *term var* rule looks like this:

$$\text{Term var}\ \frac{\langle\ \dots\ \rangle \vdash T :: K}{\langle\ \dots\ \rangle,\ t : T \vdash t : T}$$

When we use this rule, here are the parts we want to fill in:

$$\text{Term var}\ \frac{\langle\ \dots\ \rangle \vdash T :: K}{\langle\ \dots\ \rangle,\ \boxed{??t} : T \vdash \boxed{??t} : T}$$

For instance, suppose T is α and K is $*$:

$$\text{Term var}\ \frac{\langle\ \dots\ \rangle \vdash \alpha :: *}{\langle\ \dots\ \rangle,\ \boxed{??t} : \alpha \vdash \boxed{??t} : \alpha}$$

With the *term var* rule, we can introduce any term t of type α. For instance, we could introduce x:

$$\text{Term var}\ \frac{\langle\ \dots\ \rangle \vdash \alpha :: *}{\langle\ \dots\ \rangle,\ \boxed{x} : \alpha \vdash \boxed{x} : \alpha}$$

Or y:

$$\text{Term var}\ \frac{\langle\ \dots\ \rangle \vdash \alpha :: *}{\langle\ \dots\ \rangle,\ \boxed{y} : \alpha \vdash \boxed{y} : \alpha}$$

57.4 The *Weakening* Rule

In a generic form, the *weakening* rule s often stated like this:

$$\text{Weakening}\ \frac{\Gamma \vdash A : B \qquad \Gamma \vdash C : s}{\Gamma,\ x : C \vdash A : B}$$

In this formulation, s refers to either $*$ or $\Box$, A, B, and C refer to types or kinds, and x refers to an instance of C. Γ, of course, refers to a context.

We break this out into two rules, one for types, and one for terms.

57.4.1 The *Type Weaken* Rule

The *type weaken* rule works in the following way. Start with any judgment:

$$\text{Type Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,]}{}$$

We don't care what the context is, so here we just write $\langle\ \ldots\ \rangle$. We don't care what the judgment is either, so here we just write $[\,]$.

What we want to do next is weaken the context, by adding a new declaration into it. To represent this, let's first carry the judgment down underneath the line:

$$\text{Type Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,]}{\langle\ \ldots\ \rangle \vdash [\,]}$$

Then let's add a slot next to the context, to indicate the spot we want to fill in:

$$\text{Type Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,]}{\langle\ \ldots\ \rangle,\ \boxed{??} \vdash [\,]}$$

The rule we're considering here is the *type* weakening rule, so what we want to put in place of the question marks is a type T (of some kind K):

$$\text{Type Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,]}{\langle\ \ldots\ \rangle,\ \boxed{??T} :: K \vdash [\,]}$$

But of course, if we want to introduce a type of kind K, then K must be a legal kind that we have already derived. So we need a second premise above the line, which says that we have derived that kind.

$$\text{Type Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash K :: \Box}{\langle\ \ldots\ \rangle,\ \boxed{??T} :: K \vdash [\,]}$$

Now we have the full structure of the *type weaken* rule in place. When we use it, we replace the question marks with some type. For instance, suppose K is $*$:

$$\text{Type Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash * :: \Box}{\langle\ \ldots\ \rangle,\ \boxed{??T} :: * \vdash [\,]}$$

Using the *type weaken* rule, we could introduce $\alpha :: *$:

$$\text{Type Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash * :: \Box}{\langle\ \ldots\ \rangle,\ \boxed{\alpha} :: * \vdash [\,]}$$

Or $\beta :: *$:

$$\text{Type Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash * :: \Box}{\langle\ \ldots\ \rangle,\ \boxed{\beta} :: * \vdash [\,]}$$

The full rule looks like this:

$$\text{Type Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash K :: \Box}{\langle\ \ldots\ \rangle,\ T :: K \vdash [\,]}$$

57.4.2 The *Term Weaken* Rule

The *term weaken* rule works in the following way. Start with any judgment:

$$\text{Term Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,]}{}$$

What we want to do next is weaken the context, by adding a new declaration into it. To represent this, let's first carry the judgment down underneath the line:

$$\text{Term Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,]}{\langle\ \ldots\ \rangle \vdash [\,]}$$

Then let's add a slot next to the context, to indicate the spot we want to fill in:

$$\text{Term Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,]}{\langle\ \ldots\ \rangle,\ \boxed{??} \vdash [\,]}$$

The rule we're considering now is the *term* weakening rule, so what we want to put in place of the question marks is a term t (of some type T):

$$\text{Term Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,]}{\langle\ \ldots\ \rangle,\ \boxed{??t} : T \vdash [\,]}$$

Of course, if we want to introduce a term of type T, then T must be a legal type that we have already derived. So we need a second premise above the line, which says that we have derived that type.

$$\text{Term Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash T :: K}{\langle\ \ldots\ \rangle,\ \boxed{??t} : T \vdash [\,]}$$

Now we have the full structure of the *term weaken* rule in place. When we use it, we replace the question marks with some term that inhabits the type T. For instance, suppose T is $\alpha :: *$:

$$\text{Term Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash \alpha :: *}{\langle\ \ldots\ \rangle,\ \boxed{??t} : \alpha \vdash [\,]}$$

Using the *term weaken* rule, we could introduce $x : \alpha$:

$$\text{Term Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash \alpha :: *}{\langle\ \ldots\ \rangle,\ \boxed{x} : \alpha \vdash [\,]}$$

Or $y : \alpha$:

$$\text{Term Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash \alpha :: *}{\langle\ \ldots\ \rangle,\ \boxed{y} : \alpha \vdash [\,]}$$

The full rule looks like this:

$$\text{Term Weaken}\ \frac{\langle\ \ldots\ \rangle \vdash [\,] \qquad \langle\ \ldots\ \rangle \vdash T :: K}{\langle\ \ldots\ \rangle,\ t : T \vdash [\,]}$$

Chapter 58

Starter Rules

Let us use the *sort*, *var*, and *weaken* rules from last chapter, and derive some further rules that are very useful when we start building any derivation.

58.1 The *Type-Var LR* Rule

The *type-var LR* rule is short for "introduce a **type var**iable on the **left** and **right**." We start with the *sort*:

$$\text{Sort}\ \frac{}{\langle\ \rangle \vdash * :: \Box}$$

Then we use the *type var* rule to introduce a type::

$$\text{Type Var}\ \frac{\text{Sort}\ \dfrac{}{\langle\ \rangle \vdash * :: \Box}}{\langle\ \rangle,\ T :: * \vdash T :: *}$$

We can omit the empty context on the bottom line, and we can drop the labels, to save space. So the derived **type-var LR** rule looks like this:

$$\frac{\dfrac{}{\langle\ \rangle \vdash * :: \Box}}{T :: * \vdash T :: *}$$

When we use this, we want to replace T with any type that inhabits $*$. To make this clear, let's mark the parts we can fill in:

$$\frac{\dfrac{}{\langle\ \rangle \vdash * :: \Box}}{\boxed{??T} :: * \vdash \boxed{??T} :: *}$$

Here are some examples:

$$\dfrac{\dfrac{}{\langle\,\rangle \vdash * :: \Box}}{\boxed{\alpha} :: * \vdash \boxed{\alpha} :: *}$$

$$\dfrac{\dfrac{}{\langle\,\rangle \vdash * :: \Box}}{\boxed{\beta} :: * \vdash \boxed{\beta} :: *}$$

The next time we want to use this, we don't want to write the whole derivation out. So going forward, let's abbreviate this. We'll write the bottom line only, and we'll write "type-var LR" above that:

$$\dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}$$

So, for instance:

$$\dfrac{\text{Type-var LR}}{\alpha :: * \vdash \alpha :: *}$$

$$\dfrac{\text{Type-var LR}}{\beta :: * \vdash \beta :: *}$$

58.2 The *Type-Var L* Rule

The *type-var L* rule is short for "introduce a **type var**iable on the **left**." We begin with two instances of *sort*:

$$\dfrac{\text{Sort}\ \dfrac{}{\langle\,\rangle \vdash * :: \Box} \qquad \dfrac{}{\langle\,\rangle \vdash * :: \Box}\ \text{Sort}}{}$$

Then we want to weaken the sort judgment on the left. To do that, we first carry the judgment down under the line:

$$\text{Type Weaken}\ \dfrac{\text{Sort}\ \dfrac{}{\langle\,\rangle \vdash * :: \Box} \qquad \dfrac{}{\langle\,\rangle \vdash * :: \Box}\ \text{Sort}}{\langle\,\rangle \vdash * :: \Box}$$

Then we make a slot where we went to add a declaration, to weaken the context:

$$\text{Type Weaken}\ \dfrac{\text{Sort}\ \dfrac{}{\langle\,\rangle \vdash * :: \Box} \qquad \dfrac{}{\langle\,\rangle \vdash * :: \Box}\ \text{Sort}}{\langle\,\rangle,\ \boxed{??} \vdash * :: \Box}$$

The rule we're considering here is a *type*-var rule, so we want to weaken the context with a type:

$$\text{Type Weaken}\ \dfrac{\text{Sort}\ \dfrac{}{\langle\,\rangle \vdash * :: \Box} \qquad \dfrac{}{\langle\,\rangle \vdash * :: \Box}\ \text{Sort}}{\langle\,\rangle,\ \boxed{??T} :: K \vdash * :: \Box}$$

The kind K of T must be a legal kind we've already declared. On the top right, we have derived the kind $*$, so we can fill that in, to replace K:

$$\text{Type Weaken}\ \dfrac{\text{Sort}\ \dfrac{}{\langle\,\rangle \vdash * :: \Box} \qquad \dfrac{}{\langle\,\rangle \vdash * :: \Box}\ \text{Sort}}{\langle\,\rangle,\ \boxed{??T} :: * \vdash * :: \Box}$$

With this, we can introduce any type that inhabits $*$. For instance, we could introduce α:

$$\text{Type Weaken}\ \dfrac{\text{Sort}\ \dfrac{}{\langle\,\rangle \vdash * :: \Box} \qquad \dfrac{}{\langle\,\rangle \vdash * :: \Box}\ \text{Sort}}{\langle\,\rangle,\ \boxed{\alpha} :: * \vdash * :: \Box}$$

Or β:

$$\text{Type Weaken}\ \dfrac{\text{Sort}\ \dfrac{}{\langle\,\rangle \vdash * :: \Box} \qquad \dfrac{}{\langle\,\rangle \vdash * :: \Box}\ \text{Sort}}{\langle\,\rangle,\ \boxed{\beta} :: * \vdash * :: \Box}$$

Here is the full derivation of the **type-var L** rule:

$$\text{Type Weaken}\ \dfrac{\text{Sort}\ \dfrac{}{\langle\,\rangle \vdash * :: \Box} \qquad \dfrac{}{\langle\,\rangle \vdash * :: \Box}\ \text{Sort}}{\langle\,\rangle,\ T :: * \vdash * :: \Box}$$

Let's abbreviate it:

$$\dfrac{\text{Type-var L}}{\langle\,\rangle,\ T :: * \vdash * :: \Box}$$

Here are some examples of using it:

$$\dfrac{\text{Type-var L}}{\langle\,\rangle,\ \alpha :: * \vdash * :: \Box}$$

$$\dfrac{\text{Type-var L}}{\langle\,\rangle,\ \beta :: * \vdash * :: \Box}$$

58.3 The *Term-Var LR* Rule

The *term-var LR* rule is short for "introduce a **term var**iable on the **left** and **right**." We start with an instance of the *type-var LR* rule:

$$\dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}$$

Then we use the *term var* rule to introduce a term of type T on both sides of the $\vdash$ symbol.

$$\dfrac{\dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}}{T :: *,\ \boxed{??t} : T \vdash \boxed{??t} : T}$$

For instance, suppose we used the *type-var LR* rule to introduce the type $\alpha :: *$ on the left and right:

$$\dfrac{\text{Type-var LR}}{\alpha :: * \vdash \alpha :: *}$$

We then want to introduce a term of type α on the left and right:

$$\dfrac{\dfrac{\text{Type-var LR}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *,\ \boxed{??t} : \alpha \vdash \boxed{??t} : \alpha}$$

We could introduce x:

$$\dfrac{\dfrac{\text{Type-var LR}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *,\ \boxed{x} : \alpha \vdash \boxed{x} : \alpha}$$

Or y:

$$\dfrac{\dfrac{\text{Type-var LR}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: *,\ \boxed{y} : \alpha \vdash \boxed{y} : \alpha}$$

Here is the full derivation for the **term-var LR** rule:

$$\dfrac{\dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}}{T :: *,\ t : T \vdash t : T}$$

Let's abbreviate it:

$$\dfrac{\text{Term-var LR}}{T :: *,\ t : T \vdash t : T}$$

Here are some examples of using it:

$$\dfrac{\text{Term-var LR}}{\alpha :: *,\ x : \alpha \vdash x : \alpha}$$

$$\dfrac{\text{Term-var LR}}{\alpha :: *,\ y : \alpha \vdash y : \alpha}$$

58.4 The *Term-Var L* Rule

The *term-var L* rule is short for “introduce a **term var**iable on the **left**.” We start with an instance of the *type-var L* rule:

$$\frac{\text{Type-var L}}{\langle\ \rangle,\ T :: * \vdash * :: \Box}$$

We don’t need the empty context, so we can drop it:

$$\frac{\text{Type-var L}}{T :: * \vdash * :: \Box}$$

Next we take an instance of the *type-var LR* rule:

$$\frac{\text{Type-var LR}}{T :: * \vdash T :: *}$$

And we put the two together as two premises in a derivation:

$$\frac{\dfrac{\text{Type-var L}}{T :: * \vdash * :: \Box} \qquad \dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}}{}$$

We can now weaken the left side. To weaken, we copy the judgment from the to left and put it below the line:

$$\frac{\dfrac{\text{Type-var L}}{T :: * \vdash * :: \Box} \qquad \dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}}{T :: * \vdash * :: \Box}$$

And we insert a slot where we can weaken it:

$$\frac{\dfrac{\text{Type-var L}}{T :: * \vdash * :: \Box} \qquad \dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}}{T :: *, \boxed{??} \vdash * :: \Box}$$

On the top right side, we have a type $T :: *$:

$$\frac{\dfrac{\text{Type-var L}}{T :: * \vdash * :: \Box} \qquad \dfrac{\text{Type-var LR}}{T :: * \vdash \boxed{T :: *}}}{T :: *, ?? \vdash * :: \Box}$$

So we can weaken with an inhabitant of that type T:

$$\frac{\dfrac{\text{Type-var L}}{T :: * \vdash * :: \Box} \qquad \dfrac{\text{Type-var LR}}{T :: * \vdash \boxed{T :: *}}}{T :: *, \boxed{\text{t : T}} \vdash * :: \Box}$$

And there we have it. Here is the full derivation of the **Term-Var L** rule:

$$\frac{\dfrac{\text{Type-var L}}{T :: * \vdash * :: \Box} \qquad \dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}}{T :: *, t : T \vdash * :: \Box}$$

Let's abbreviate it:

$$\frac{\text{Term-var L}}{T :: *, t : T \vdash * :: \Box}$$

Here are some examples:

$$\frac{\text{Term-var L}}{\alpha :: *, x : \alpha \vdash * :: \Box}$$

$$\frac{\text{Term-var L}}{\sigma :: *, y : \sigma \vdash * :: \Box}$$

Chapter 59

Formation

In λP, the formation rule is different from what it was in $\lambda\underline{\omega}$. In λP, we use the *formation* rule to form Π-types. This is different from $\lambda\underline{\omega}$, where we used the *formation* rule to form arrow-types (e.g., $\alpha \to \beta$) and arrow-kinds (e.g., $* \to *$). In λP, there are no arrow-types or arrow-kinds of the sort that we can form in $\lambda\underline{\omega}$, much like how in $\lambda\underline{\omega}$, we cannot form Π-types.

59.1 The *Formation* Rule

The formation rule is often stated in this generic form:

$$\text{Form}\ \frac{\Gamma, x : A \vdash B : s \qquad \Gamma \vdash A : *}{\Gamma \vdash (\Pi x : A.B) : s}$$

Here s is either $*$ or $\square$, A and B are types or kinds, and x is an instance of A. And of course, Γ is a context.

To understand this better, let's break this into two rules: one for when s is $\square$, and another for when s is $*$.

59.2 The $\square$-*Form* Rule

First, let's do the $\square$-*form* rule. Let's suppose first that we have some context, where we can derive some kind $K :: \square$:

$$\frac{\langle\ \dots\ \rangle \vdash K :: \square}{}$$

Let's also suppose we have a term t of type T that is free in the context:

$$\frac{\langle\ \dots\ \rangle,\ \boxed{t : T} \vdash K :: \square}{}$$

The formation rule says that if we find this situation, we can mark the term t as replaceable in K. We indicate that with the Π binder, like this:

$$\Pi t : T.K$$

This is, of course, a kind. So it inhabits the super kind:

$$(\Pi t : T.K) :: \Box$$

So that is the judgment we want to put under the line in our rule.

$$\frac{\langle\ \dots\ \rangle,\ t : T \vdash K :: \Box}{\langle\ \dots\ \rangle \vdash (\Pi t : T.K) :: \Box}$$

Of course, if we have a term t of type T, then T must be a legal type that's been derived in that same context. So we must add that as a second premise to the rule. Here is the full **□-form** rule:

$$\Box\text{-form}\ \frac{\langle\ \dots\ \rangle,\ t : T \vdash K :: \Box \qquad \langle\ \dots\ \rangle \vdash T :: *}{\langle\ \dots\ \rangle \vdash (\Pi t : T.K) :: \Box}$$

Note that with this rule, we generate a Π-*kind*, not a Π-*type*.

59.2.1 Alternative Notation

A Π-kind is an abstraction, so it takes us from the input type T to the kind K. If you like, we can write $K \to T$ as a shorthand:

$$(\Pi t : T.K) :: \Box = (T \to K) :: \Box \tag{59.1}$$

Given that, we could write the same rule like this:

$$\Box\text{-form}\ \frac{\langle\ \dots\ \rangle,\ t : T \vdash K :: \Box \qquad \langle\ \dots\ \rangle \vdash T :: *}{\langle\ \dots\ \rangle \vdash (T \to K) :: \Box}$$

59.2.2 Examples

Here is an example of the rule. Suppose K is $*$, T is α, and t is x. Then we can build up the derivation as follows. Suppose we have a context $\langle\ \dots\ \rangle$, in which we can derive the kind $* :: \Box$:

$$\frac{\langle\ \dots\ \rangle \vdash * :: \Box}{}$$

Now suppose we have a term x of type α that is free in the context:

$$\frac{\langle\ \dots\ \rangle,\ x : \alpha \vdash * :: \Box}{}$$

With the □-*form* rule, we can mark x as replaceable in $*$:

$$\frac{\langle\ \ldots\ \rangle,\ x : \alpha \vdash * :: \Box}{\langle\ \ldots\ \rangle \vdash (\Pi x : \alpha.*) :: \Box}$$

Of course, if we see a type α, then α must be a legal type derived already. So we need to add that as a second premise above the line:

$$\frac{\langle\ \ldots\ \rangle,\ x : \alpha \vdash * :: \Box \qquad \langle\ \ldots\ \rangle \vdash \alpha :: *}{\langle\ \ldots\ \rangle \vdash (\Pi x : \alpha.*) :: \Box}$$

Or, using the alternative notation:

$$\frac{\langle\ \ldots\ \rangle,\ x : \alpha \vdash * :: \Box \qquad \langle\ \ldots\ \rangle \vdash \alpha :: *}{\langle\ \ldots\ \rangle \vdash (\alpha \to *) :: \Box}$$

Here is another example. In this case, the term is y.

$$\frac{\langle\ \ldots\ \rangle,\ y : \alpha \vdash * :: \Box \qquad \langle\ \ldots\ \rangle \vdash \alpha :: *}{\langle\ \ldots\ \rangle \vdash (\Pi y : \alpha.*) :: \Box}$$

Or, using the alternative notation:

$$\frac{\langle\ \ldots\ \rangle,\ y : \alpha \vdash * :: \Box \qquad \langle\ \ldots\ \rangle \vdash \alpha :: *}{\langle\ \ldots\ \rangle \vdash (\alpha \to *) :: \Box}$$

A final example. In this case, the type is β and the term is z.

$$\frac{\langle\ \ldots\ \rangle,\ z : \beta \vdash * :: \Box \qquad \langle\ \ldots\ \rangle \vdash \beta :: *}{\langle\ \ldots\ \rangle \vdash (\Pi z : \beta.*) :: \Box}$$

Or, using the alternative notation:

$$\frac{\langle\ \ldots\ \rangle,\ z : \beta \vdash * :: \Box \qquad \langle\ \ldots\ \rangle \vdash \beta :: *}{\langle\ \ldots\ \rangle \vdash (\beta \to *) :: \Box}$$

59.3 The ∗-*Form* Rule

The ∗-*form* rule is nearly the same. First, suppose we have some context where we can derive a term, call it $T_2 :: *$:

$$\frac{\langle\ \ldots\ \rangle \vdash T_2 :: *}{}$$

Now suppose we have a term t of another type, call it T_1, that is free in the context:

$$\frac{\langle\ \ldots\ \rangle,\ t : T_1 \vdash T_2 :: *}{}$$

The ∗-*form* rule says we can mark $t : T_1$ as replaceable in T_2:

$$\frac{\langle\ \dots\ \rangle,\ t : T_1 \vdash T_2 :: *}{\langle\ \dots\ \rangle \vdash (\Pi t : T_1 . T_2) :: *}$$

Of course, if we see a type T_1, then it must be a legal type derived already. So let's add that as a second premise above the line. That gives us the full **$*$-form** rule:

$$*\text{-form}\ \frac{\langle\ \dots\ \rangle,\ t : T_1 \vdash T_2 :: * \qquad \langle\ \dots\ \rangle \vdash T_1 :: *}{\langle\ \dots\ \rangle \vdash (\Pi t : T_1 . T_2) :: *}$$

Or, using the alternative notation for Π types:

$$*\text{-form}\ \frac{\langle\ \dots\ \rangle,\ t : T_1 \vdash T_2 :: * \qquad \langle\ \dots\ \rangle \vdash T_1 :: *}{\langle\ \dots\ \rangle \vdash (T_1 \to T_2) :: *}$$

Notice that in this case, when we use the $*$-*form* rule, we generate a Π-*type*, not a Π-*kind*. We use the $\Box$-*form* rule to generate Π-kinds, and we use the $*$-*form* rule to generate Π-types.

Chapter 60

Formation Starter Rules

Let us take the rules we've learned so far and derive some further rules that are useful for introducing Π-bindings when we are building derivations.

60.1 The Π-*kind R* Rule

Suppose we have used the *term-var L* rule to introduce a judgment with the following form:

$$\frac{\text{Term-var L}}{T :: *, t : T \vdash * :: \Box}$$

The $\Box$-*form* rule tells us we can mark $t : T$ as replaceable in $*$:

$$\frac{\dfrac{\text{Term-var L}}{T :: *, t : T \vdash * :: \Box}}{T :: * \vdash (\Pi t : T.*) :: \Box}$$

Of course, T must be a valid type that we have derived, so let's add that as a second premise above the line:

$$\Box\text{-form}\ \frac{\dfrac{\text{Term-var L}}{T :: *, t : T \vdash * :: \Box} \qquad \dfrac{}{T :: * \vdash T :: *}}{T :: * \vdash (\Pi t : T.*) :: \Box}$$

How do we derive that? We can use the *type-var LR* rule. And that gives us the full **Π-kind R** rule:

$$\Box\text{-form}\ \frac{\dfrac{\text{Term-var L}}{T :: *, t : T \vdash * :: \Box} \qquad \dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}}{T :: * \vdash (\Pi t : T.*) :: \Box}$$

Or, in the alternative notation for Π-bindings:

$$\square\text{-form}\;\dfrac{\dfrac{\text{Term-var L}}{T :: *, t : T \vdash * :: \square} \qquad \dfrac{\text{Type-var LR}}{T :: * \vdash T :: *}}{T :: * \vdash (T \to *) :: \square}$$

60.1.1 Abbreviation

Let's abbreviate this. We'll keep the bottom line, and write "Π-kind R" above the line:

$$\dfrac{\text{Π-kind R}}{T :: * \vdash (\Pi t : T.*) :: \square}$$

Or, using the alternative notation for Π-bindings:

$$\dfrac{\text{Π-kind R}}{T :: * \vdash (T \to *) :: \square}$$

60.1.2 Examples

Suppose we have used the *term-var L* rule to introduce the following judgment:

$$\dfrac{\text{Term-var L}}{\alpha :: *, x : \alpha \vdash * :: \square}$$

The $\square$-*form* rule tells us we can mark $x : \alpha$ as replaceable in $*$:

$$\dfrac{\dfrac{\text{Term-var L}}{\alpha :: *, x : \alpha \vdash * :: \square}}{\alpha :: * \vdash (\Pi x : \alpha.*) :: \square}$$

Of course, α must be a valid type that we have derived (which we can do with the *type-var LR* rule), so let's add that as a second premise above the line:

$$\square\text{-form}\;\dfrac{\dfrac{\text{Term-var L}}{\alpha :: *, x : \alpha \vdash * :: \square} \qquad \dfrac{\text{Type-var LR}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: * \vdash (\Pi x : \alpha.*) :: \square}$$

Or, with the alternative notation for Π-bindings:

$$\square\text{-form}\;\dfrac{\dfrac{\text{Term-var L}}{\alpha :: *, x : \alpha \vdash * :: \square} \qquad \dfrac{\text{Type-var LR}}{\alpha :: * \vdash \alpha :: *}}{\alpha :: * \vdash (\alpha \to *) :: \square}$$

In the abbreviated format:

$$\frac{\text{Π-kind R}}{\alpha :: * \vdash (\Pi x : \alpha.*) :: \Box}$$

Or, using the alternative notation for Π-bindings:

$$\frac{\text{Π-kind R}}{\alpha :: * \vdash (\alpha \to *) :: \Box}$$

Here is another example, but with y as the term, instead of x:

$$\frac{\text{Π-kind R}}{\alpha :: * \vdash (\Pi y : \alpha.*) :: \Box}$$

Or, using the alternative notation for Π-bindings:

$$\frac{\text{Π-kind R}}{\alpha :: * \vdash (\alpha \to *) :: \Box}$$

Here is a final example, with z as the term and β as the type:

$$\frac{\text{Π-kind R}}{\beta :: * \vdash (\Pi z : \beta.*) :: \Box}$$

Or, using the alternative notation for Π-bindings:

$$\frac{\text{Π-kind R}}{\beta :: * \vdash (\beta \to *) :: \Box}$$

60.2 The □ Π-*kind L* Rule

Suppose I have used the *type-var L* rule to derive a judgment like this:

$$\frac{\text{Type-var L}}{T_1 :: * \vdash * :: \Box}$$

Now suppose I want to weaken this by introducing an instance of a Π-kind into the context. To weaken, we carry the judgment down below the line:

$$\frac{\dfrac{\text{Type-var L}}{T_1 :: * \vdash * :: \Box}}{T_1 :: * \vdash * :: \Box}$$

Then we put in a slot where we want to weaken the context:

$$\frac{\dfrac{\text{Type-var L}}{T_1 :: * \vdash * :: \Box}}{T_1 :: *, \boxed{??} \vdash * :: \Box}$$

Now I can fill in the question marks with an instance of a Π-kind. An instance of a kind is a *type*. So, let's call this instance T_2, and let's say that it is an instance of the kind $\Pi t : T_1.*$:

$$\dfrac{\dfrac{\text{Type-var L}}{T_1 :: * \vdash * :: \Box}}{T_1 :: *,\ \boxed{T_2 :: (\Pi t : T_1.*)} \vdash * :: \Box}$$

Of course, if we see a Π-kind in our derivation, then that must be a legal Π-kind we've derived already. So let's add that as a second premise above the line:

$$\text{Type-weaken}\ \dfrac{\dfrac{\text{Type-var L}}{T_1 :: * \vdash * :: \Box} \qquad \dfrac{}{T_1 :: * \vdash \boxed{(\Pi t : T_1.*) :: \Box}}}{T_1 :: *,\ \boxed{T_2 :: (\Pi t : T_1.*)} \vdash * :: \Box}$$

How do we derive that? We can do that with the Π-*kind R* rule:

$$\text{Type-weaken}\ \dfrac{\dfrac{\text{Type-var L}}{T_1 :: * \vdash * :: \Box} \qquad \dfrac{\Pi\text{-kind R}}{T_1 :: * \vdash (\Pi t : T_1.*) :: \Box}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash * :: \Box}$$

If we drop the labels, we have the full derivation for the □ **Π-kind L** rule:

$$\dfrac{\dfrac{\text{Type-var L}}{T_1 :: * \vdash * :: \Box} \qquad \dfrac{\Pi\text{-kind R}}{T_1 :: * \vdash (\Pi t : T_1.*) :: \Box}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash * :: \Box}$$

Or, using the alternative notation for Π-bindings:

$$\dfrac{\dfrac{\text{Type-var L}}{T_1 :: * \vdash * :: \Box} \qquad \dfrac{\Pi\text{-kind R}}{T_1 :: * \vdash (T_1 \to *) :: \Box}}{T_1 :: *,\ T_2 :: (T_1 \to *) \vdash * :: \Box}$$

60.2.1 Abbreviation

Let's abbreviate it. We'll keep the bottom line of the derivation, and write "□ Π-kind L" above the line.

$$\dfrac{\Box\ \Pi\text{-kind L}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash * :: \Box}$$

Or, using the alternative notation for Π-bindings:

$$\dfrac{\Box\ \Pi\text{-kind L}}{T_1 :: *,\ T_2 :: (T_1 \to *) \vdash * :: \Box}$$

60.2.2 Examples

Suppose we used the *type-var L* rule to derive this judgment:

$$\frac{\text{Type-var L}}{\alpha :: * \vdash * :: \Box}$$

Now suppose we want to weaken this by introducing a Π-kind into the context. To weaken, we carry the judgment down below the line:

$$\frac{\dfrac{\text{Type-var L}}{\alpha :: * \vdash * :: \Box}}{\alpha :: * \vdash * :: \Box}$$

Then we put in a slot where we want to weaken the context:

$$\frac{\dfrac{\text{Type-var L}}{\alpha :: * \vdash * :: \Box}}{\alpha :: *,\ \boxed{??\Pi} :: K \vdash * :: \Box}$$

Let's fill in the question marks with an instance of a Π-kind, for instance $\beta :: (\Pi x : \alpha.*)$.

$$\frac{\dfrac{\text{Type-var L}}{\alpha :: * \vdash * :: \Box}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash * :: \Box}$$

Of course, if we see a Π-kind in our derivation, then that must be a legal Π-kind we've derived already. So let's add that as a second premise above the line:

$$\frac{\dfrac{\text{Type-var L}}{\alpha :: * \vdash * :: \Box} \qquad \dfrac{\Pi\text{-kind R}}{\alpha :: * \vdash (\Pi x : \alpha.*) :: \Box}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash * :: \Box}$$

Or, using the alternative notation for Π-bindings:

$$\frac{\dfrac{\text{Type-var L}}{\alpha :: * \vdash * :: \Box} \qquad \dfrac{\Pi\text{-kind R}}{\alpha :: * \vdash (\alpha \to *) :: \Box}}{\alpha :: *,\ \beta :: (\alpha \to *) \vdash * :: \Box}$$

In the abbreviated style:

$$\frac{\Box\ \Pi\text{-kind L}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash * :: \Box}$$

Or, using the alternative notation for Π-bindings:

$$\frac{\Box\ \Pi\text{-kind L}}{\alpha :: *,\ \beta :: (\alpha \to *) \vdash * :: \Box}$$

Here is another example, using z for the term, and β and γ for the types:

$$\frac{\Box\ \Pi\text{-kind L}}{\beta :: *,\ \gamma :: (\Pi z : \beta.*) \vdash * :: \Box}$$

Or, using the alternative notation for Π-bindings:

$$\frac{\Box\ \Pi\text{-kind L}}{\beta :: *,\ \gamma :: (\beta \to *) \vdash * :: \Box}$$

60.3 The $*$ Π-*kind L* Rule

Notice what we can do with the $\Box$ Π-*Kind L* rule. We can use that rule to add a Π-kind into the context on the left, when we have the judgment $* :: \Box$ on the right.

We can use a similar technique to add a Π-kind into the context on the left, when we have the judgment $T :: *$ on the right. Let's call this the $*$ **Π-kind L** rule. Here it is:

$$\frac{\dfrac{\text{Type-var LR}}{T_1 :: * \vdash T_1 :: *} \qquad \dfrac{\Pi\text{-kind R}}{T_1 :: * \vdash (\Pi t : T_1.*) :: \Box}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash T_1 :: *}$$

Or, using the alternative notation for Π-bindings:

$$\frac{\dfrac{\text{Type-var LR}}{T_1 :: * \vdash T_1 :: *} \qquad \dfrac{\Pi\text{-kind R}}{T_1 :: * \vdash (\Pi t : T_1.*) :: \Box}}{T_1 :: *,\ T_2 :: (T_1 \to *) \vdash T_1 :: *}$$

The only difference between this derivation and the $\Box$ Π-*kind L* derivation is that in this case, we start with the *type-var LR* rule on the top left, rather than the *type-var L* rule.

60.3.1 Abbreviation

Let's abbreviate it. We'll keep the bottom line of the derivation, and write "$*$ Π-kind L" above the line.

$$\frac{*\ \Pi\text{-kind L}}{T_1 :: *, T_2 :: (\Pi t : T_1.*) \vdash T_1 :: *}$$

Or, using the alternative notation for Π-bindings:

$$\frac{*\ \Pi\text{-kind L}}{T_1 :: *, T_2 :: (T_1 \to *) \vdash T_1 :: *}$$

60.3.2 Examples

Here we use the rule with α as T and x as t:

$$\frac{}{\alpha :: *, \beta :: (\Pi x : \alpha . *) \vdash \alpha :: *} * \ \Pi\text{-kind L}$$

Or, using the alternative notation for Π-bindings:

$$\frac{}{\alpha :: *, \beta :: (\alpha \rightarrow *) \vdash \alpha :: *} * \ \Pi\text{-kind L}$$

And here we use the rule with β and γ as types, and z as the term:

$$\frac{}{\beta :: *, \gamma :: (\Pi z : \beta . *) \vdash \beta :: *} * \ \Pi\text{-kind L}$$

Or, using the alternative notation for Π-bindings:

$$\frac{}{\beta :: *, \gamma :: (\beta \rightarrow *) \vdash \beta :: *} * \ \Pi\text{-kind L}$$

Chapter 61

Input Rules

In λP, we can use formation rules to generate Π-kinds, which mark a *term* replaceable in a *kind*. For instance:

$$(\Pi x : \alpha.*) :: \Box \tag{61.1}$$

So, Π-kinds are like functions that go from terms to kinds. They are arrows. That is why the alternative notation is helpful:

$$\alpha \to * \tag{61.2}$$

If we want to reduce these expressions, they need the right kind of input — that is, terms that have the right type. In this example here, the right kind of input is a term of type α.

We can use the rules we've already learned to derive a few more rules that are useful for generating input terms.

61.1 The Π-*kind input LR* Rule

Suppose we have used the $*$ Π-*kind L* rule to generate a judgment of this form:

$$\frac{* \ \Pi\text{-kind L}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash T_1 :: *}$$

We can use the *term-var* rule to introduce an instance of the type T_1 on both sides of the $\vdash$ symbol:

$$\text{Term-var}\ \frac{\dfrac{* \ \Pi\text{-kind L}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash T_1 :: *}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*), t : T_1 \vdash t : T_1}$$

If we drop the "Term-var" label, we get the full derivation for the **Π-kind input LR** rule:

$$\dfrac{\dfrac{*\ \Pi\text{-kind L}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash T_1 :: *}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*), t : T_1 \vdash t : T_1}$$

Or, in the alternative notation for Π-bindings:

$$\dfrac{\dfrac{*\ \Pi\text{-kind L}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash T_1 :: *}}{T_1 :: *,\ T_2 :: (T_1 \to *), t : T_1 \vdash t : T_1}$$

61.1.1 Abbreviation

Let's abbreviate this:

$$\dfrac{\Pi\text{-kind input LR}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*), t : T_1 \vdash t : T_1}$$

Or, in the alternative notation for Π-bindings:

$$\dfrac{\Pi\text{-kind input LR}}{T_1 :: *,\ T_2 :: (T_1 \to *), t : T_1 \vdash t : T_1}$$

61.1.2 Examples

Suppose we have used the $*$ Π-*kind L* rule to generate this judgment:

$$\dfrac{*\ \Pi\text{-kind L}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \alpha :: *}$$

We can use the *term-var* rule to introduce a term — let's say, y — of type α on both sides of the $\vdash$ symbol:

$$\dfrac{\dfrac{*\ \Pi\text{-kind L}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \alpha :: *}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*), y : \alpha \vdash y : \alpha}$$

Or, in the alternative notation for Π-bindings:

$$\dfrac{\dfrac{*\ \Pi\text{-kind L}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \alpha :: *}}{\alpha :: *,\ \beta :: (\alpha \to *), y : \alpha \vdash y : \alpha}$$

Here is the same derivation, in the abbreviated style:

$$\dfrac{\Pi\text{-kind input LR}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*), y : \alpha \vdash y : \alpha}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\text{Π-kind input LR}}{\alpha :: *,\ \beta :: (\alpha \to *), y : \alpha \vdash y : \alpha}$$

Here is another example, where we introduce $z :: \alpha$ instead of $y :: \alpha$:

$$\frac{\text{Π-kind input LR}}{\alpha :: *,\ \beta :: (\Pi x : \alpha . *), z : \alpha \vdash z : \alpha}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\text{Π-kind input LR}}{\alpha :: *,\ \beta :: (\alpha \to *), z : \alpha \vdash z : \alpha}$$

61.2 The Π-*kind input L* Rule

Suppose we have used the Π-*kind LR* rule to form a judgment of this form:

$$\frac{\text{Π-kind LR}}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *) \vdash T_2 :: (\Pi t : T_1 . *)}$$

Suppose now that we want to weaken this with an instance t of T_1. To weaken, we carry the judgment down below the line:

$$\frac{\dfrac{\text{Π-kind LR}}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *) \vdash T_2 :: (\Pi t : T_1 . *)}}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *) \vdash T_2 :: (\Pi t : T_1 . *)}$$

And we put in a slot where we want to weaken it:

$$\frac{\dfrac{\text{Π-kind LR}}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *) \vdash T_2 :: (\Pi t : T_1 . *)}}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *),\ \boxed{??} \vdash T_2 :: (\Pi t : T_1 . *)}$$

Now we can put the term we want to weaken the context with — it is $t : T_1$ — in place of the question marks:

$$\frac{\dfrac{\text{Π-kind LR}}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *) \vdash T_2 :: (\Pi t : T_1 . *)}}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *),\ \boxed{t : T_1} \vdash T_2 :: (\Pi t : T_1 . *)}$$

Of course, the type T_1 must be a legal type that we've already derived, so let's add that as a second premise on the right, above the line:

$$\frac{\dfrac{\text{Π-kind LR}}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *) \vdash T_2 :: (\Pi t : T_1 . *)} \qquad \dfrac{}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *) \vdash \boxed{T_1 :: *}}}{T_1 :: *,\ T_2 :: (\Pi t : T_1 . *),\ \boxed{t : T_1} \vdash T_2 :: (\Pi t : T_1 . *)}$$

How do we derive that judgment on the top right? We can use the $*$ Π-*kind L* rule:

$$\dfrac{\dfrac{\Pi\text{-kind LR}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash T_2 :: (\Pi t : T_1.*)} \qquad \dfrac{\boxed{*\ \Pi\text{-kind L}}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash T_1 :: *}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*),\ t : T_1 \vdash T_2 :: (\Pi t : T_1.*)}$$

That gives us the full Π**-kind input L** rule:

$$\dfrac{\dfrac{\Pi\text{-kind LR}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash T_2 :: (\Pi t : T_1.*)} \qquad \dfrac{*\ \Pi\text{-kind L}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*) \vdash T_1 :: *}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*),\ t : T_1 \vdash T_2 :: (\Pi t : T_1.*)}$$

Or, in the alternative notation for Π-bindings:

$$\dfrac{\dfrac{\Pi\text{-kind LR}}{T_1 :: *,\ T_2 :: (T_1 \to *) \vdash T_2 :: (\Pi t : T_1.*)} \qquad \dfrac{*\ \Pi\text{-kind L}}{T_1 :: *,\ T_2 :: (T_1 \to *) \vdash T_1 :: *}}{T_1 :: *,\ T_2 :: (T_1 \to *),\ t : T_1 \vdash T_2 :: (\Pi t : T_1.*)}$$

61.2.1 Abbreviation

Let's abbreviate it:

$$\dfrac{\Pi\text{-kind input L}}{T_1 :: *,\ T_2 :: (\Pi t : T_1.*),\ t : T_1 \vdash T_2 :: (\Pi t : T_1.*)}$$

Or, in the alternative notation for Π-bindings:

$$\dfrac{\Pi\text{-kind input L}}{T_1 :: *,\ T_2 :: (T_1 \to *),\ t : T_1 \vdash T_2 :: (T_1 \to *)}$$

61.2.2 Examples

Suppose we have used the Π-*kind LR* rule to form this judgment:

$$\dfrac{\Pi\text{-kind LR}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \beta :: (\Pi x : \alpha.*)}$$

Suppose now that we want to weaken this with a term — say y — of α. To weaken, we carry the judgment down below the line:

$$\dfrac{\dfrac{\Pi\text{-kind LR}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \beta :: (\Pi x : \alpha.*)}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \beta :: (\Pi x : \alpha.*)}$$

And we put in a slot where we want to weaken it:

$$\dfrac{\dfrac{\text{Π-kind LR}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \beta :: (\Pi x : \alpha.*)}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ \boxed{??} \vdash \beta :: (\Pi x : \alpha.*)}$$

Now we can put the term we want to weaken the context with — it is $y : \alpha$ — in place of the question marks:

$$\dfrac{\dfrac{\text{Π-kind LR}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \beta :: (\Pi x : \alpha.*)}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ \boxed{y : \alpha} \vdash \beta :: (\Pi x : \alpha.*)}$$

Of course, the type α must be a legal type that we've already derived, so let's add that as a second premise on the right, above the line:

$$\dfrac{\dfrac{\text{Π-kind LR}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \beta :: (\Pi x : \alpha.*)} \qquad \dfrac{\text{$*$ Π-kind L}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \boxed{\alpha :: *}}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ \boxed{y : \alpha} \vdash \beta :: (\Pi x : \alpha.*)}$$

Or, in the alternative notation for Π-bindings:

$$\dfrac{\dfrac{\text{Π-kind LR}}{\alpha :: *,\ \beta :: (\alpha \to *) \vdash \beta :: (\alpha \to *)} \qquad \dfrac{\text{$*$ Π-kind L}}{\alpha :: *,\ \beta :: (\alpha \to *) \vdash \boxed{\alpha :: *}}}{\alpha :: *,\ \beta :: (\alpha \to *),\ \boxed{y : \alpha} \vdash \beta :: (\alpha \to *)}$$

Here is the same derivation, in the abbreviated style:

$$\dfrac{\text{Π-kind input L}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash \beta :: (\Pi x : \alpha.*)}$$

Or, in the alternative notation for Π-bindings:

$$\dfrac{\text{Π-kind input L}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta :: (\alpha \to *)}$$

Here is another example, with β and γ for types, and z for the term we introduce:

$$\dfrac{\text{Π-kind input L}}{\beta :: *,\ \gamma :: (\Pi x : \beta.*),\ z : \beta \vdash \gamma :: (\Pi x : \beta.*)}$$

Or, in the alternative notation for Π-bindings:

$$\dfrac{\text{Π-kind input L}}{\beta :: *,\ \gamma :: (\beta \to *),\ z : \beta \vdash \gamma :: (\beta \to *)}$$

Chapter 62

Π-Kind Application

62.1 Setting Up

Suppose we have derived a Π-kind that says you can replace a term t_1 of type T_1 in the kind K:

$$\langle\ \ldots\ \rangle \vdash (\Pi t_1 : T_1 . K) :: \Box \tag{62.1}$$

In the alternative notation for Π-bindings, we can say this is an arrow from type T_1 to K:

$$\langle\ \ldots\ \rangle \vdash (T_1 \to K) :: \Box \tag{62.2}$$

Now suppose we have derived a type that inhabits this Π-kind. Let us call this type T_2, to keep it distinct from T_1.

$$\langle\ \ldots\ \rangle \vdash T_2 :: (T_1 \to K) \tag{62.3}$$

Or, in the Π-style notation:

$$\langle\ \ldots\ \rangle \vdash T_2 :: (\Pi t_1 : T_1 . K) \tag{62.4}$$

At this point, we have a type whose kind is an arrow. Another way to say that is: an arrow is an abstraction — it has a term that is marked as replaceable.

This means we can stipulate something to replace the marked term with. And that, of course, is application. The replacement must have the correct type, of course: it must be a term (call it t_2), of type T_1:

$$\langle\ \ldots\ \rangle \vdash t_2 : T_1 \tag{62.5}$$

So, if we have those two, we can apply the arrow to the input.

62.2 The Application Rule

Remember that applications in general have this form:

$$\frac{\langle\ \dots\ \rangle \vdash M \qquad \langle\ \dots\ \rangle \vdash N}{\langle\ \dots\ \rangle \vdash MN}$$

But once we add types, applications can only happen if (i) the first term is an arrow $A \to B$, and (ii) the second term has the right type of input, namely A. But if (i) and (ii) are met, then the two terms can be combined, and the resulting application MN ends up with the output of the arrow, namely B.

$$\frac{\langle\ \dots\ \rangle \vdash M : A \to B \qquad \langle\ \dots\ \rangle \vdash N : A}{\langle\ \dots\ \rangle \vdash MN : B}$$

That is what we have here too, in λP. First, we have an arrow on the left:

$$\frac{\langle\ \dots\ \rangle \vdash T_2 :: (\Pi t_1 : T_1.K) \qquad ??}{??}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\langle\ \dots\ \rangle \vdash T_2 :: (T_1 \to K) \qquad ??}{??}$$

Next, we have a term on the right that has the right input type:

$$\frac{\langle\ \dots\ \rangle \vdash T_2 :: (\Pi t_1 : T_1.K) \qquad \langle\ \dots\ \rangle \vdash t_2 : T_1}{??}$$

In the alternative notation for Π-bindings:

$$\frac{\langle\ \dots\ \rangle \vdash T_2 :: (T_1 \to K) \qquad \langle\ \dots\ \rangle \vdash t_2 : T_1}{??}$$

So, at this point we have an arrow on the left, and a term on the right which has the right kind of input.

$$\frac{\langle\ \dots\ \rangle \vdash T_2 :: (\boxed{T_1} \to K) \qquad \langle\ \dots\ \rangle \vdash t_2 : \boxed{T_1}}{??}$$

That means we can put them together into an application:

$$\frac{\langle\ \dots\ \rangle \vdash \boxed{T_2} :: (T_1 \to K) \qquad \langle\ \dots\ \rangle \vdash \boxed{t_2} : T_1}{\langle\ \dots\ \rangle \vdash \boxed{T_2}\boxed{t_2} :: ??}$$

What type or kind does the application get? It gets the output of the arrow, with every t_1 in it replaced by t_2. In this case, the output of the arrow is K, so we want $K[t_1 := t_2]$.

$$\frac{\langle\ \dots\ \rangle \vdash T_2 :: (T_1 \to \boxed{K}) \qquad \langle\ \dots\ \rangle \vdash t_2 : T_1}{\langle\ \dots\ \rangle \vdash T_2 t_2 :: \boxed{K[t_1 := t_2]}}$$

In the alternative notation:

$$\frac{\langle\ \dots\ \rangle \vdash T_2 :: (\Pi t_1 : T_1 . \boxed{K}) \qquad \langle\ \dots\ \rangle \vdash t_2 : T_1}{\langle\ \dots\ \rangle \vdash T_2 t_2 :: \boxed{K[t_1 := t_2]}}$$

With that, we have the complete **Π-kind appl** rule. Here it is, without the boxes:

$$\frac{\langle\ \dots\ \rangle \vdash T_2 :: (\Pi t_1 : T_1 . K) \qquad \langle\ \dots\ \rangle \vdash t_2 : T_1}{\langle\ \dots\ \rangle \vdash T_2 t_2 :: K[t_1 := t_2]}$$

And in the alternative notation:

$$\frac{\langle\ \dots\ \rangle \vdash T_2 :: (T_1 \to K) \qquad \langle\ \dots\ \rangle \vdash t_2 : T_1}{\langle\ \dots\ \rangle \vdash T_2 t_2 :: K[t_1 := t_2]}$$

You can see that the term t_1 is inessential to the application. What matters here is its *type*. Whatever type it has, the input term must have the matching type. So the alternative notation for Π-bindings conveys loses no information.

62.3 Examples

Suppose we have used the Π-*kind input L* rule to derive the following:

$$\frac{\text{Π-kind input L}}{\alpha :: *,\ \beta :: (\Pi x : \alpha . *),\ y : \alpha \vdash \beta :: (\Pi x : \alpha . *)}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\text{Π-kind input L}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta :: (\alpha \to *)}$$

Now suppose that we have used the Π-*kind input LR* rule to derive this:

$$\frac{\text{Π-kind input LR}}{\alpha :: *,\ \beta :: (\Pi x : \alpha . *), y : \alpha \vdash y : \alpha}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\text{Π-kind input LR}}{\alpha :: *,\ \beta :: (\alpha \to *), y : \alpha \vdash y : \alpha}$$

These two judgments can be used to form an application. Let's build it up piece by piece. First, let's put the two next to each other:

$$\frac{\dfrac{\text{Π-kind input LR}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta :: (\alpha \to *)} \qquad \dfrac{\text{Π-kind input LR}}{\alpha :: *,\ \beta :: (\alpha \to *), y : \alpha \vdash y : \alpha}}{??}$$

Now let's drop the labels on the top, so we can focus only on the relevant details:

$$\frac{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta :: (\alpha \to *) \qquad \alpha :: *,\ \beta :: (\alpha \to *), y : \alpha \vdash y : \alpha}{??}$$

Notice that both of these have the same context:

$$\frac{\boxed{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha} \vdash \beta :: (\alpha \to *) \qquad \boxed{\alpha :: *,\ \beta :: (\alpha \to *), y : \alpha} \vdash y : \alpha}{??}$$

Let's bring the context down underneath the line:

$$\frac{\boxed{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha} \vdash \beta :: (\alpha \to *) \qquad \boxed{\alpha :: *,\ \beta :: (\alpha \to *), y : \alpha} \vdash y : \alpha}{\boxed{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha} \vdash ??}$$

And, in fact, since the context is the same, let's just write $\langle\ \dots\ \rangle$ instead, again so we can focus on the relevant details:

$$\frac{\boxed{\langle\ \dots\ \rangle} \vdash \beta :: (\alpha \to *) \qquad \boxed{\langle\ \dots\ \rangle} \vdash y : \alpha}{\boxed{\langle\ \dots\ \rangle} \vdash ??}$$

Now we can focus on the application itself. On the left we have an arrow from α to $*$.

$$\frac{\langle\ \dots\ \rangle \vdash \beta :: \boxed{(\alpha \to *)} \qquad \langle\ \dots\ \rangle \vdash y : \alpha}{\langle\ \dots\ \rangle \vdash ??}$$

And we can see the correct input type on the right:

$$\frac{\langle\ \dots\ \rangle \vdash \beta :: (\boxed{\alpha} \to *) \qquad \langle\ \dots\ \rangle \vdash y : \boxed{\alpha}}{\langle\ \dots\ \rangle \vdash ??}$$

So, we can combine them to form an application:

$$\frac{\langle\ \dots\ \rangle \vdash \boxed{\beta} :: (\alpha \to *) \qquad \langle\ \dots\ \rangle \vdash \boxed{y} : \alpha}{\langle\ \dots\ \rangle \vdash \boxed{\beta}\boxed{y} :: ??}$$

What type or kind does the application get? It gets the output of the arrow, with every x replaced by y. The output of the arrow is $*$, so we want $*[x := y]$.

$$\frac{\langle\ \dots\ \rangle \vdash \beta :: (\alpha \to \boxed{*}) \qquad \langle\ \dots\ \rangle \vdash y : \alpha}{\langle\ \dots\ \rangle \vdash \beta y :: \boxed{*[x := y]}}$$

Of course, there is no x to replace in $*$, so the result is just $*$:

$$\frac{\langle\ \dots\ \rangle \vdash \beta :: (\alpha \to *) \qquad \langle\ \dots\ \rangle \vdash y : \alpha}{\langle\ \dots\ \rangle \vdash \beta y :: \boxed{*}}$$

There we have it — a complete Π-kind application. Here it is, with the context filled back in:

$$\frac{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta :: (\alpha \to *) \qquad \alpha :: *,\ \beta :: (\alpha \to *), y : \alpha \vdash y : \alpha}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta y :: *}$$

Here is another example, with different types and terms:

$$\frac{\sigma :: *,\ \tau :: (\sigma \to *),\ x : \sigma \vdash \tau :: (\sigma \to *) \qquad \sigma :: *,\ \tau :: (\sigma \to *), x : \sigma \vdash x : \sigma}{\sigma :: *,\ \tau :: (\sigma \to *),\ x : \sigma \vdash \tau x :: *}$$

Chapter 63

Dependent Arrow Types

With the Π-*kind appl* rule, we can form types composed of both a type and a term, like this:

$$\alpha x :: * \tag{63.1}$$

$$\alpha y :: * \tag{63.2}$$

$$\sigma z :: * \tag{63.3}$$

These are dependent types in the sense that they are types which depend on a term.

Once we have constructed one or more dependent types, we can go a step further: we can use the Π-*type form* rule to form further arrow types (Π-types) built from dependent types.

63.1 Dependent Arrow Type Formation

Recall some of the examples from the chapter on Π-kind applications. Here is one of the judgments we derived:

$$\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta y :: * \tag{63.4}$$

Notice the structure of this judgment. On the left, we have a term y of type α, and on the right we have a dependent type βy:

$$\alpha :: *,\ \beta :: (\alpha \to *),\ \boxed{y : \alpha} \vdash \boxed{\beta y :: *} \tag{63.5}$$

Let's put $\langle\ \ldots\ \rangle$ in place of the rest of the context, so we can focus only on the details we care about. That gives us this:

$$\langle\ \ldots\ \rangle,\ y : \alpha \vdash \beta y :: * \tag{63.6}$$

We can form a Π-type from this, using the Π-*type form* rule. The conditions are right: we have a term y of type α free in the context, and we have a type $\beta y :: *$ on the right side of the $\vdash$ symbol.

$$\frac{\langle\ \ldots\ \rangle,\ \boxed{y:\alpha} \vdash \boxed{\beta y :: *}}{}$$

So, we carry the context down (minus the free $y:\alpha$):

$$\frac{\boxed{\langle\ \ldots\ \rangle},\ y:\alpha \vdash \beta y :: *}{\boxed{\langle\ \ldots\ \rangle} \vdash ??}$$

And then we mark the y as replaceable in βy:

$$\frac{\langle\ \ldots\ \rangle,\ y:\alpha \vdash \beta y :: *}{\langle\ \ldots\ \rangle \vdash (\Pi y:\alpha.\beta y) :: *}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\langle\ \ldots\ \rangle,\ y:\alpha \vdash \beta y :: *}{\langle\ \ldots\ \rangle \vdash (\alpha \rightarrow \beta y) :: *}$$

Of course, if we have a term $y:\alpha$ that appears in the derivation, then it's type must be a legal term derived already. And indeed, we can derive that using the $*$ Π-*kind L* rule. Let's put that as a second premise above the line:

$$\frac{\langle\ \ldots\ \rangle,\ y:\alpha \vdash \beta y :: * \qquad \langle\ \ldots\ \rangle \vdash \boxed{\alpha :: *}}{\langle\ \ldots\ \rangle \vdash (\Pi y:\alpha.\beta y) :: *}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\langle\ \ldots\ \rangle,\ y:\alpha \vdash \beta y :: * \qquad \langle\ \ldots\ \rangle \vdash \boxed{\alpha :: *}}{\langle\ \ldots\ \rangle \vdash (\alpha \rightarrow \beta y) :: *}$$

With that, we have formed an arrow type that has a dependent type (βy) as one of its parts. It is an arrow that takes as input a term of type y, and it return as output a type βy, with y replaced by the input.

63.1.1 Formulating the Rule

Let's replace y, α, β, and some of the $*$ symbols with more generic symbols, so we can turn it into a derivation schema. First, let's replace y with t:

$$\frac{\langle\ \ldots\ \rangle,\ \boxed{t}:\alpha \vdash \beta\boxed{t} :: * \qquad \langle\ \ldots\ \rangle \vdash \alpha :: *}{\langle\ \ldots\ \rangle \vdash (\Pi\boxed{t}:\alpha.\beta\boxed{t}) :: *}$$

Now let's replace α by T_1:

$$\frac{\langle\ \dots\ \rangle,\ t : \boxed{T_1} \vdash \beta t :: * \qquad \langle\ \dots\ \rangle \vdash \boxed{T_1} :: *}{\langle\ \dots\ \rangle \vdash (\Pi t : \boxed{T_1}.\beta t) :: *}$$

Next, let's replace β with T_2:

$$\frac{\langle\ \dots\ \rangle,\ t : T_1 \vdash \boxed{T_2}t :: * \qquad \langle\ \dots\ \rangle \vdash T_1 :: *}{\langle\ \dots\ \rangle \vdash (\Pi t : T_1.\boxed{T_2}t) :: *}$$

Finally, let's replace the two $*$ symbols on the left with K (we don't need to replace the $*$ symbol in the top right, because in any instance of this derivation it will always be a single $*$):

$$\frac{\langle\ \dots\ \rangle,\ t : T_1 \vdash T_2 t :: \boxed{K} \qquad \langle\ \dots\ \rangle \vdash T_1 :: *}{\langle\ \dots\ \rangle \vdash (\Pi t : T_1.T_2 t) :: \boxed{K}}$$

With that, we have a derivation schema that tells us how to derive a dependent arrow type. Let's call this derivation the **dependent arrow type** rule. Here it is, without any boxes:

$$\frac{\langle\ \dots\ \rangle,\ t : T_1 \vdash T_2 t :: K \qquad \langle\ \dots\ \rangle \vdash T_1 :: *}{\langle\ \dots\ \rangle \vdash (\Pi t : T_1.T_2 t) :: K}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\langle\ \dots\ \rangle,\ t : T_1 \vdash T_2 t :: K \qquad \langle\ \dots\ \rangle \vdash T_1 :: *}{\langle\ \dots\ \rangle \vdash (T_1 \to T_2 t) :: K}$$

63.1.2 Examples

Let's take the example we started with. Suppose we use Π-kind application to derive that first judgment:

$$\frac{\begin{array}{c}\Pi\text{-kind appl}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash \beta y :: *}$$

And suppose we use the $*$ Π-*kind L* rule to derive this:

$$\frac{\begin{array}{c}*\ \Pi\text{-kind L}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \alpha :: *}$$

Now let's put these together:

$$\dfrac{\dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash \beta y :: *} \qquad \dfrac{\overset{*\ \text{Π-kind L}}{\vdots}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \alpha :: *}}{??}$$

And then we can use the *dependent arrow type* rule to get the conclusion:

$$\dfrac{\dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash \beta y :: *} \qquad \dfrac{\overset{*\ \text{Π-kind L}}{\vdots}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \alpha :: *}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash (\Pi y : \alpha.\beta y) :: *}$$

63.2 Double Dependent Arrow Type

Suppose we have used Π-*kind appl* to construct a judgment of this form:

$$\dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\langle\ \dots\ \rangle \vdash Tt_1 :: K}$$

Make a copy of it, beside it:

$$\dfrac{\dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\langle\ \dots\ \rangle \vdash Tt_1 :: K} \qquad \dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\langle\ \dots\ \rangle \vdash Tt_1 :: K}}{??}$$

Now we can use weakening to declare an inhabitant of $Tt_1 :: K$. To weaken, we carry the judgment from the left down under the line, and put in a slot where we can weaken the context:

$$\dfrac{\dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\langle\ \dots\ \rangle \vdash Tt_1 :: K} \qquad \dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\langle\ \dots\ \rangle \vdash Tt_1 :: K}}{\langle\ \dots\ \rangle,\ \boxed{??} \vdash Tt_1 :: K}$$

Then we put in an inhabitant of the type from the right side:

$$\dfrac{\dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\langle\ \dots\ \rangle \vdash Tt_1 :: K} \qquad \dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\langle\ \dots\ \rangle \vdash \boxed{Tt_1} :: K}}{\langle\ \dots\ \rangle,\ \boxed{t_2 : Tt_1} \vdash Tt_1 :: K}$$

This is just another instance of our *term-var LR* rule. So let's remove the parts above the bottom line, and label it:

$$\dfrac{\overset{\text{Term-var LR}}{\vdots}}{\langle\ \dots\ \rangle,\ t_2 : Tt_1 \vdash Tt_1 :: K}$$

Notice that we have a term on the left, and a type on the right:

$$\dfrac{\overset{\text{Term-var LR}}{\vdots}}{\langle\ \dots\ \rangle,\ \boxed{t_2} : Tt_1 \vdash \boxed{Tt_1} :: K}$$

We can use the Π-*type form* rule to construct an arrow.

$$\dfrac{\dfrac{\overset{\text{Term-var LR}}{\vdots}}{\langle\ \dots\ \rangle,\ \boxed{t_2} : Tt_1 \vdash \boxed{Tt_1} :: K}}{\langle\ \dots\ \rangle \vdash (\Pi\boxed{t_2} : Tt_1.\boxed{Tt_1}) :: K}$$

Of course, if we have a term $t_2 : Tt_1$, then its type must be derived as a legal type. Let's add that as a premise on the right:

$$\dfrac{\dfrac{\overset{\text{Term-var LR}}{\vdots}}{\langle\ \dots\ \rangle,\ t_2 : Tt_1 \vdash Tt_1 :: K} \qquad \dfrac{\overset{??}{\vdots}}{\langle\ \dots\ \rangle, \vdash \boxed{Tt_1} :: K}}{\langle\ \dots\ \rangle \vdash (\Pi t_2 : \boxed{Tt_1}.Tt_1) :: K}$$

How do we derive the premise on the right? With the Π-*kind appl* rule again:

$$\dfrac{\dfrac{\overset{\text{Term-var LR}}{\vdots}}{\langle\ \dots\ \rangle,\ t_2 : Tt_1 \vdash Tt_1 :: K} \qquad \dfrac{\overset{\text{Π-kind appl}}{\vdots}}{\langle\ \dots\ \rangle, \vdash Tt_1 :: K}}{\langle\ \dots\ \rangle \vdash (\Pi t_2 : Tt_1.Tt_1) :: K}$$

Here it is in the alternative notation for Π-bindings:

$$\dfrac{\dfrac{\substack{\text{Term-var LR}\\ \vdots}}{\langle\ \dots\ \rangle,\ t_2 : Tt_1 \vdash Tt_1 :: K} \qquad \dfrac{\substack{\Pi\text{-kind appl}\\ \vdots}}{\langle\ \dots\ \rangle, \vdash Tt_1 :: K}}{\langle\ \dots\ \rangle \vdash (Tt_1 \to Tt_1) :: K}$$

Let us call this the **2x dependent type arrow** rule. Let's abbreviate it:

$$\dfrac{\substack{\text{2x dep. type arrow}\\ \vdots}}{\langle\ \dots\ \rangle \vdash (\Pi t_2 : Tt_1 \to Tt_1) :: K}$$

Or, in the alternative notation:

$$\dfrac{\substack{\text{2x dep. type arrow}\\ \vdots}}{\langle\ \dots\ \rangle \vdash (Tt_1 \to Tt_1) :: K}$$

63.2.1 Example

Take the application judgment we started the chapter with:

$$\dfrac{\substack{\Pi\text{-kind appl}\\ \vdots}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta y :: *}$$

Repeat it on the right:

$$\dfrac{\dfrac{\substack{\Pi\text{-kind appl}\\ \vdots}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta y :: *} \qquad \dfrac{\substack{\Pi\text{-kind appl}\\ \vdots}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta y :: *}}{??}$$

Weaken it:

$$\dfrac{\dfrac{\substack{\Pi\text{-kind appl}\\ \vdots}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta y :: *} \qquad \dfrac{\substack{\Pi\text{-kind appl}\\ \vdots}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta y :: *}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha, \boxed{z : \beta y} \vdash \beta y :: *}$$

This is just the *term-var LR* rule, so let's remove the details and just label it:

$$\frac{\begin{array}{c}\text{Term-var LR}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha, z : \beta y \vdash \beta y :: *}$$

Copy Π-*kind appl* again, on the right:

$$\frac{\dfrac{\begin{array}{c}\text{Term-var LR}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha, z : \beta y \vdash \beta y :: *} \qquad \dfrac{\begin{array}{c}\Pi\text{-kind appl}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta y :: *}}{??}$$

Form the Π-type:

$$\frac{\dfrac{\begin{array}{c}\text{Term-var LR}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha, z : \beta y \vdash \beta y :: *} \qquad \dfrac{\begin{array}{c}\Pi\text{-kind appl}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash \beta y :: *}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash (\beta y \to \beta y) :: *}$$

Or, abbreviated:

$$\frac{\begin{array}{c}\text{2x dep. type arrow}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash (\beta y \to \beta y) :: *}$$

63.3 Triple Dependent Type Arrows

Consider the example we just derived:

$$\frac{\begin{array}{c}\text{2x dep. type arrow}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *),\ y : \alpha \vdash (\beta y \to \beta y) :: *}$$

Notice that we have the right conditions for yet another Π-type formation. We have a term on the left, and a type on the right:

$$\frac{\begin{array}{c}\text{2x dep. type arrow}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *),\ \boxed{y : \alpha} \vdash \boxed{(\beta y \to \beta y)} :: *}$$

Let's form the Π-type:

$$\dfrac{\begin{array}{c}\text{2x dep. type arrow}\\ \vdots\end{array}}{\dfrac{\alpha :: *,\ \beta :: (\alpha \to *),\ \boxed{y : \alpha} \vdash \boxed{(\beta y \to \beta y)} :: *}{\alpha :: *,\ \beta :: (\alpha \to *), \vdash (\boxed{\alpha} \to \boxed{(\beta y \to \beta y)} :: *}}$$

As always, we need to declare α as a legal type with a second premise on the right. We derived that before with the $*$ Π-*kind L* rule:

$$\dfrac{\dfrac{\begin{array}{c}\text{2x dep. type arrow}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *),\ \boxed{y : \alpha} \vdash (\beta y \to \beta y) :: *} \qquad \dfrac{\begin{array}{c}* \ \Pi\text{-kind L}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\alpha \to *) \vdash \ \boxed{\alpha} :: *}}{\alpha :: *,\ \beta :: (\alpha \to *), \vdash (\boxed{\alpha} \to (\beta y \to \beta y) :: *}$$

And that gives us yet another Π-type. We can turn this into a general schema by ignoring things in the context that don't matter, replacing y with t, α with T_1, β with T_2, and $*$ with K:

$$\dfrac{\dfrac{\begin{array}{c}\text{2x dep. type arrow}\\ \vdots\end{array}}{\langle\ \dots\ \rangle,\ t : T_1 \vdash (T_2 t \to T_2 t) :: K} \qquad \dfrac{\begin{array}{c}* \ \Pi\text{-kind L}\\ \vdots\end{array}}{\langle\ \dots\ \rangle \vdash T_1 :: K}}{\langle\ \dots\ \rangle \vdash (T_1 \to (T_2 t \to T_2 t) :: K}$$

And we can abbreviate the whole thing as the **3x dependent type arrow** rule:

$$\dfrac{\begin{array}{c}\text{3x dep. type arrow}\\ \vdots\end{array}}{\langle\ \dots\ \rangle \vdash (T_1 \to (T_2 t \to T_2 t) :: K}$$

Chapter 64

Abstraction and Conversion

In λP, abstraction works with Π-kinds and Π-types. λP also has a conversion rule, which is identical to the conversion rule from $\lambda\underline{\omega}$.

64.1 The Abstraction Rule

The abstraction rule is often stated in a generic form. Let's build it up piece by piece. To start suppose we have derived an inhabitant b of some B.

$$\Gamma \vdash b : B$$

Let's also suppose we have a free x of type A in the context:

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

We can perform abstraction on this. That is, we can mark the x as replaceable in b:

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

What type does this abstraction get? Since this is an abstraction, it's an arrow that goes from the type of the bound variable to the type of the body. The type of the bound variable is A, and the type of the body variable is B. Hence, it is a Π type (in the alternative notation for Π-bindings, it is: $A \to B$).

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

Of course, the type $(\Pi x : A.B)$ must be a legal type that we have derived. So that goes in a second premise (where s is either $*$ or $\square$):

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

Or, in the alternative notation for Π-bindings:

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

That is the **abstraction rule** for λP, in the general form that it is often stated.

64.2 Term Abstraction

Let's restate the above rule, using our notation. Suppose we have a judgment where we have derived an inhabitant t_2 of a type T_2, and we have a free term t_1 of type T_1 in the context:

$$\Gamma, t_1 : T_1 \vdash t_2 : T_2$$

We can perform abstraction on this. That is, we can mark t_1 as replaceable in t_2:

$$\frac{\Gamma, t_1 : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda t_1 : T_1.t_2}$$

What type does this abstraction get? It's a Π-type that goes from T_1 to T_2:

$$\frac{\Gamma, t_1 : T_1 \vdash t_2 : T_2}{(\Gamma \vdash \lambda t_1 : T_1.t_2) : (\Pi t_1 : T_1.T_2)}$$

Of course, that Π-type must be a legal type that we have derived. So that goes in a second premise:

$$\frac{\Gamma, t_1 : T_1 \vdash t_2 : T_2 \qquad (\Gamma \vdash \Pi t_1 : T_1.T_2) :: K}{(\Gamma \vdash \lambda t_1 : T_1.t_2) : (\Pi t_1 : T_1.T_2)}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\Gamma, t_1 : T_1 \vdash t_2 : T_2 \qquad (\Gamma \vdash \Pi t_1 : T_1.T_2) :: K}{(\Gamma \vdash \lambda t_1 : T_1.t_2) : (T_1 \to T_2)}$$

That is the **abstraction** rule, in our notation (it is equivalent to the rule above, it is just written with different symbols).

64.3 Example

Suppose we derive a judgment like this:

$$\dfrac{\begin{array}{c}\Pi\text{-kind appl}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash \beta y :: *}$$

We can then use the *var* rule to introduce a term of type βy on each side of the $\vdash$ symbol:

$$\dfrac{\dfrac{\begin{array}{c}\Pi\text{-kind appl}\\ \vdots\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash \beta y :: *}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, z : \beta y \vdash z : \beta y}$$

Let's focus just on the bottom line (we'll replace everything above the bottom line with dots):

$$\begin{array}{c}\vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, z : \beta y \vdash z : \beta y\end{array}$$

Notice that we have a term on the left, and a term on the right:

$$\begin{array}{c}\vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, \boxed{z : \beta y} \vdash \boxed{z : \beta y}\end{array}$$

We can perform abstraction on that. Let's mark the z as replaceable in z:

$$\dfrac{\begin{array}{c}\vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, z : \beta y \vdash z : \beta y\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, \vdash \boxed{(\lambda z : \beta y.z)}}$$

What type does it get?

$$\dfrac{\begin{array}{c}\vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, z : \beta y \vdash z : \beta y\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, \vdash (\lambda z : \beta y.z) : \boxed{??}}$$

The abstraction's type is a Π-type that goes from the type of the bound term to the type of the body. So it's an arrow from βy to βy:

$$\frac{\begin{array}{c}\vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, z : \beta y \vdash z : \beta y\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, \vdash (\lambda z : \beta y.z) : \boxed{(\Pi z : \beta y.\beta y)}}$$

Or, in the alternative notation for Π-bindings:

$$\frac{\begin{array}{c}\vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, z : \beta y \vdash z : \beta y\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha, \vdash (\lambda z : \beta y.z) : \boxed{\beta y \to \beta y}}$$

Of course, that Π-type must be a legal type that we've derived, so we need a second premise. This all won't fit on the page, so let's abbreviate context on the left:

$$\frac{\begin{array}{c}\vdots\\ \langle\ \dots\ \rangle, z : \beta y \vdash z : \beta y\end{array} \qquad \begin{array}{c}\vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash (\Pi z : \beta y.\beta y) :: *\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash (\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y)}$$

How do we derive the premise on the right?

$$\frac{\begin{array}{c}\vdots\\ \langle\ \dots\ \rangle, z : \beta y \vdash z : \beta y\end{array} \qquad \begin{array}{c}??\\ \vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash (\Pi z : \beta y.\beta y) :: *\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash (\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y)}$$

We use the *2x dep. type arrow* rule:

$$\frac{\begin{array}{c}\vdots\\ \langle\ \dots\ \rangle, z : \beta y \vdash z : \beta y\end{array} \qquad \begin{array}{c}\text{2x dep. type arrow}\\ \\ \vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash (\Pi z : \beta y.\beta y) :: *\end{array}}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash (\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y)}$$

64.4 Abstracting Again

Take the judgment we just derived:

$$\begin{array}{c}\vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ y : \alpha \vdash (\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y)\end{array}$$

Notice that we have a term on the left and term on the right:

$$\dfrac{\vdots}{\alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ \boxed{y : \alpha} \vdash \boxed{(\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y)}}$$

We can perform an abstraction on this too. We can mark the y from the left as replaceable in the term on the right:

$$\begin{array}{c}\vdots\\ \alpha :: *,\ \beta :: (\Pi x : \alpha.*),\ \boxed{y : \alpha} \vdash \boxed{(\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y)} \\ \hline \alpha :: *,\ \beta :: (\Pi x : \alpha.*) \vdash \lambda\boxed{y : \alpha}.(\boxed{(\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y))}\end{array}$$

Let's abbreviate the context on the left to make room:

$$\begin{array}{c}\vdots\\ \langle\ \ldots\ \rangle,\ y : \alpha \vdash (\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y) \\ \hline \langle\ \ldots\ \rangle \vdash \lambda y : \alpha.((\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y))\end{array}$$

What type does this new abstraction get?

$$\begin{array}{c}\vdots\\ \langle\ \ldots\ \rangle,\ y : \alpha \vdash (\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y) \\ \hline \langle\ \ldots\ \rangle \vdash (\lambda y : \alpha.((\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y))) :: \boxed{??}\end{array}$$

The type will be an arrow from the type of the bound term to the type of the body. The bound term is y, which has type α, and the body is itself an abstraction which has the type $\Pi z : \beta y.\beta y$. So:

$$\begin{array}{c}\vdots\\ \langle\ \ldots\ \rangle,\ y : \alpha \vdash (\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y) \\ \hline \langle\ \ldots\ \rangle \vdash (\lambda y : \alpha.((\lambda z : \beta y.z) : (\Pi z : \beta y.\beta y))) :: \boxed{(\Pi y : \alpha.(\Pi z : \beta y.\beta y))}\end{array}$$

Or, in the alternative notation for Π-bindings:

$$\begin{array}{c}\vdots\\ \langle\ \ldots\ \rangle,\ y : \alpha \vdash (\lambda z : \beta y.z) : (\beta y \to \beta y) \\ \hline \langle\ \ldots\ \rangle \vdash (\lambda y : \alpha.((\lambda z : \beta y.z) : (\beta y \to \beta y))) :: \boxed{(\alpha \to (\beta y \to \beta y))}\end{array}$$

64.5 Conversion

As I noted above, in λP we have a conversion rule. It is no different from the conversion rule in $\lambda\underline{\omega}$:

$$\frac{\Gamma \vdash A : B_1 \qquad \Gamma \vdash B_2 : s \qquad B_1 =_\beta B_2}{\Gamma \vdash A : B_2}$$

This says that if A is an inhabitant of the type (or kind) B_1, and B_2 is a legal type (or kind) s, and if a β-reduction can reduce B_1 to B_2, then we can say that A is also an inhabitant of B_2.

We can break this rule up into two separate rules, one for types and one for kinds. Here is the one for kinds:

$$\frac{\langle\ \ldots\ \rangle \vdash T :: K_1 \qquad \langle\ \ldots\ \rangle \vdash K_2 :: \Box \qquad K_1 =_\beta K_2}{\langle\ \ldots\ \rangle \vdash T :: K_2}$$

This says that if T is a type that inhabits kind K_1, and K_1 can be reduced to another legal kind K_2, then T inhabits K_2 as well.

Here is the conversion rule for types:

$$\frac{\langle\ \ldots\ \rangle \vdash t : T_1 \qquad \langle\ \ldots\ \rangle \vdash T_2 : K \qquad T_1 =_\beta T_2}{\langle\ \ldots\ \rangle \vdash t : T_2}$$

This says that if t is a term that inhabits type T_1, and T_1 can be reduced to another legal type T_2, then t inhabits T_2 as well.