Type Theory for Logic and Mathematics

A philosophical and mathematical introduction to type theory

Kevin Davey

July 9, 2023

Contents

Part I

Logic

Chapter 1

Introduction to the λ-calculus.

1.1 Preamble.

This chapter is an introduction to the λ-calculus, which is a basic tool needed for much of the rest of this book. We begin by presenting the λ-calculus as a language for talking clearly about functions and their application to arguments, and go on to present the λ-calculus as something more like a programming language. We develop some of the basic mathematical ideas of both the typed and untyped λ-calculus. More sophisticated versions of the λ-calculus are presented in subsequent chapters, but the core ideas of these more sophisticated versions remain largely the same as those of the simple version presented in this chapter.

1.2 Functions and λ-abstraction.

Almost all of mathematics revolves around the manipulation of *functions*. Mathematicians constantly apply functions to inputs to obtain outputs, and combine simple functions in various ways to obtain more complicated functions. Mathematics contains familiar notations for expressing such things. Given a unary function f and an input i, we write $f(i)$ to indicate the value of the function f at i. Given two unary functions f and g, we write $f(g(i))$ to indicate the function in which g is applied to the input i first, and then f is applied to the result. Generalizing, combining, and iterating these notations, mathematicians end up writing down complex expressions such as $f(g(g(x,y),h(z)))$ to denote functions. This notation is perfectly adequate for most ordinary purposes.

Still, in certain situations this sort of notation can be a bit ambiguous. Consider a binary function $f(x,y)$. One can think of this function as some sort of process that on two inputs i and j produces an output $f(i,j)$. But a different point of view is also possible. When a computer applies a binary function $f(x,y)$ to inputs i and j, it might *first* replace x with i, and *then* y with j. This ordered replacement of x with i and then y with j is actually a somewhat subtle process. Replacing x with i, we end up with a unary function $f(i,y)$ of y. We may then replace y with j

in this expression to obtain the final value $f(i,j)$. What we are doing here is treating $f(x,y)$ as a *unary* function of x, that takes an input i for x, and outputs the unary function $f(i,y)$ of y. We can then of course evaluate this function $f(i,y)$ for some value j of y.

Consider for example the function $f(x,y) = x + y$ on natural numbers. We may think of this as a function of x that, when input with the number 3, outputs the function $3 + y$ of y. Instead of thinking of addition a *binary* function, we are thus thinking of it as a *unary* function that given any number x as input, outputs the unary function $x + y$ of y. If we so desire, we can in this way redescribe any binary function as a unary function.

Of course, we could equally well think of $x + y$ as a unary function of y that takes a natural number y as input, and outputs the unary function $x + y$ of x. On this point of view, when input with the number 3, f outputs the function $x + 3$ of x. We thus have two different ways of treating $x+y$ as a unary function. They will both give us the same result for any ordinary addition problem such as $2+3$, but these two different ways of treating $x+y$ as a unary function nevertheless describe two quite distinct processes.

An expression such as $x+y$ can thus be viewed in several different ways as a unary function. It can be viewed as a unary function of x that always outputs a function of y, or as a unary function of y that always outputs a function of x. In ordinary mathematical situations this ambiguity does not really matter, and we can just casually think of f as a binary function of x and y. But in other contexts (for example, in describing the step-by-step operation of a computer program) we sometimes must be more careful.

A notation that is very useful for resolving such ambiguities is that of *λ-abstraction*. To indicate that $x+y$ is to be viewed as a unary function of x, we use the notation $\lambda x(x+y)$, and to indicate that $x+y$ is to be viewed as a unary function of y, we use the notation $\lambda y(x+y)$. More generally, given any expression E and a variable z, $\lambda z(E)$ refers to the expression E viewed as a unary function of z. (If z does not occur free in E, then this will just be a constant function, as we will see later.) Thus the λ operator 'abstracts' from an expression, allowing us to view it as a unary function of a particular, single variable.

Definition 1.1: λ-abstraction.

If E is an expression and z a variable, $\lambda z(E)$ is the expression E viewed as a function of z.

Using this notation, the expression $\lambda x(x + y)$ is then the expression $x + y$ viewed as a (unary) function of x, which on any input i returns the function $i + y$ of y. By contrast, the expression $\lambda y(x + y)$ is the expression $x + y$ viewed as a (unary) function of y, which on any input j returns the function $x + j$ of x.

Most importantly, we can iterate the λ notation to write expressions such as

$$\lambda x(\lambda y(x + y)). \tag{1}$$

Because $\lambda y(x + y)$ is the expression $x + y$ viewed as a function of y, we can then think of (1) as just

$$\lambda x \text{ } (x + y \text{ viewed as a function of } y)$$

which we can in turn think of as

($x + y$ viewed as a function of y) viewed as a function of x.

This is a function of x, which on input i, returns $i + y$ viewed as a function of y. This of course is just the way of thinking about addition described earlier, in which $x + y$ is viewed as a unary function of x that returns a unary function of y. The expression

$$\lambda y(\lambda x(x + y)) \tag{2}$$

is, on the other hand, a way of thinking about addition in which $x+y$ is viewed as a unary function of y that returns a unary function of x. In a sense, (1) and (2) describe the same function implemented in two different ways. Thus, although (1) and (2) are in some sense the same function, our new notation allows us to distinguish a point of view from which they are different. This will be useful in what follows.

To improve readability, we will often drop parentheses and just write $\lambda x\lambda y(x+y)$ or $\lambda y\lambda x(x+y)$ instead of (1) and (2) where there is no risk of ambiguity.

Some further terminology will also be useful. Just as in ordinary quantificational logic we talk about free and bound variables, we also say that in the expression $\lambda y(x + y)$, the variable x is *free* and the variable y *bound*. More specifically, we say that the bound variable y that occurs in $\lambda y(x + y)$ has been *bound by the operator* λy. An expression with no free variables is *closed*, and an expression with free variables is *open*. So for example, $\lambda x\lambda y(x + y)$ is closed, while $\lambda x(x + y)$, $\lambda y(x + y)$ and $x + y$ are open. Although the notions of free and bound variables can be given more rigorous definitions, an intuitive understanding of them will suffice in what follows.

Exercises for Section 1.2

1. The expression $\lambda x(x \times y)$ has the form $\lambda x(E)$, and is thus a unary function of x. What is its output when 7 is substituted for x? (Here '$\times$' refers to multiplication.)

2. What is the output of the function $\lambda y(x \times y)$ when given the input 7?

3. Using λ-notation, write a unary function that when given an input x, outputs $x \times 2$.

4. Using λ-notation, write a unary function that when given an input x, outputs a unary function that when given an input y, outputs $x \times y$.

5. Using λ-notation, write a unary function that when given an input x, outputs a unary function that when given an input y, outputs a unary function that when given an input z, outputs $x + (y \times z)$.

6. Are the following terms open or closed? What are their free variables, if they have any?

(i) $\lambda y(x \times y)$
(ii) $\lambda x\lambda y\lambda z(x + y + z)$
(iii) $\lambda x\lambda y\lambda z(x + y)$
(iv) $\lambda x\lambda y(x + y + z)$

1.3 The untyped λ-calculus

In this section, we introduce the simplest version of the λ-calculus. The λ-calculus comes in many different versions, and we consider several throughout this book. The first version we will consider is an example of a so-called *untyped λ-calculus*. (What the word 'untyped' is doing here will only become clear later on when we introduce the 'typed' lambda-calculus with which it can be contrasted.) The first version of the untyped lambda calculus we will consider consists of expressions such as

$$\begin{array}{c} \lambda x(\lambda y(x(x(y)))) \\ x(\lambda x(y(x))) \\ \lambda y(x(\lambda x(y(x)))) \end{array} \tag{3}$$

These expressions are all known as λ-terms. Assuming we have some list $x, y, ...$ of variables, the rules of formation for λ-terms are as follows:

Definition 1.2: Simple untyped λ-terms.

(i) Any variable $x, y, z, ...$ is a λ-term.
(ii) If M and N are λ-terms, then so is $M(N)$.
(iii) If M is a λ-term and x a variable, then $\lambda x(M)$ is a λ-term.

You should note how these rules can be combined to generate terms such as those in (3). For example, the variables x and y are λ-terms by **(i)**. Applying **(ii)** gives that $y(x)$ is a λ-term. Applying **(iii)** then gives that $\lambda x(y(x))$ is a λ-term, and applying **(ii)** again gives that $x(\lambda x(y(x)))$ is a λ-term.

Note that operations like '+' and '$\times$' that appeared in the previous section are not officially part of our language, and thus do not appear in simple untyped λ-terms. While for explanatory purposes it was useful earlier to write expressions such as $\lambda x(\lambda y(x + y))$, such terms are not officially part of the simplest version of the untyped λ-calculus we are considering here.

Those who have seen the λ-calculus before will note that the conventions for parentheses given here are different from some other texts. We will discuss this in a little more detail shortly.

What do our λ-terms refer to? We will think of all λ-terms as *unary functions*. They are unary functions which, when given an input – which will also always be a unary function – output a unary function. This unary function can in turn be applied to another input – which will itself be a unary function – yielding yet another unary function as output.

More specifically, given two λ-terms M and N, we think of $M(N)$ as the result of applying the unary function M to the unary function N. Rule **(ii)** above states that whenever we have two λ-terms M and N, we can apply the function M to N, obtaining $M(N)$. Now $M(N)$ is itself a λ-term and thus a unary function, and so using rule **(ii)** again it can be applied to any λ-term O to obtain $M(N)(O)$. This λ-term can likewise be applied to any λ-term P to obtain $M(N)(O)(P)$, and so on.

This sort of behavior might seem odd. One might expect such a process of applying functions to inputs to eventually 'bottom out' with some sort of value or result which is *not* itself a function.

But in the simplest version of the λ-calculus that we are considering here, computations never 'bottom out' in anything that is not a function – all λ-terms are unary functions, the outputs of which are unary functions, and the outputs of *those* unary functions are further unary functions, and so on and so forth, ad infinitum.

On this point of view, even our variables $x, y, z, ...$ will be viewed as unary functions. So if x and y are variables, then they are both λ-terms and thus unary functions, and the result of applying the unary function x to the input y is just the λ-term $x(y)$. We can iterate this sort of process using rules **(i)** and **(ii)** to obtain λ-terms $x(y)$, $x(x)$, $x(y)(z)$, $x(y(z))$, $x(x(y)(y))$, each of which denotes a distinct unary function.

Let us turn our attention to rule **(iii)** and the associated concept of λ-abstraction. As an example, consider a term like $x(y)$. Using rule **(iii)**, we can perform two different types of 'abstraction' over this term, by viewing it is a function of x, and writing $\lambda x(x(y))$, or by viewing it as a function of y, and writing $\lambda y(x(y))$. As an exercise, let us consider the meaning of each of these expressions in turn.

We begin with the expression $\lambda x(x(y))$, in which $x(y)$ is viewed as a function of x. This function takes an input f for x, and outputs $f(y)$. That is to say, the function $\lambda x(x(y))$ transforms a unary function f into its value $f(y)$ at y. So $\lambda x(x(y))$ may be described as the function 'evaluate the function on input y'. The process of evaluating a function at y may thus be redescribed as the application of the λ-term $\lambda x(x(y))$ to that function.

Consider next the expression $\lambda y(x(y))$, in which $x(y)$ is viewed as a function of y. This function takes an input c for y, and outputs $x(c)$. That is to say, the function $\lambda y(x(y))$ transforms an input c into $x(c)$. So $\lambda y(x(y))$ may be described as the function 'apply the function x to the input'. This is of course just a fancy way of redescribing the function x itself. The activity of applying a function x to an input may thus be redescribed as applying the λ-term $\lambda y(x(y))$ to that input.

Given any λ-term M and variable x, rule **(iii)** allows us to form the term $\lambda x(M)$, in which M is viewed as a (unary) function of x. Typically, this will be most useful when M contains a free occurrence of the variable x. However, we do not require x to appear free in M, and so rule **(iii)** allows us to construct λ-terms such as $\lambda x(y(y))$. It also allows us to abstract over the same variable twice, as in the λ-term $\lambda x(x(\lambda x(x(y))))$. We discuss the meaning of these terms shortly.

Because too many parentheses can sometimes impair readability, we will sometimes drop parentheses where there is no risk of ambiguity or the intended reading is obvious from context. So for example, the expression xy will be an abbreviation for $x(y)$ (i.e., x applied to y), and the expression λvE will be an abbreviation for the function $\lambda v(E)$. To minimize the number of parentheses, we will also assume association to the left - so for example, the term $wxyz$ will denote $w(x)(y)(z)$, and *not* $w(x(y(z)))$. The difference between these two sorts of expressions is important. The expression $w(x)(y)(z)$ is the function w, applied to x, the output of which is a function which is then applied to y, the output of which is a function which is then applied to z. By contrast, the expression $w(x(y(z)))$ is the function w, applied to the result of applying the function x to the result of applying the function y to the function z. You should think about the difference between these two terms and the way in which it is expressed in the placement of parentheses.

As mentioned already, we will sometimes omit parentheses between adjacent λ-abstractions,

and so instead of writing $\lambda x(\lambda y(E))$ we will sometimes just write $\lambda x \lambda y(E)$. Combining these conventions, instead of writing

$$\lambda x(\lambda y(x(x)(y)))$$

we will generally just write

$$\lambda x \lambda y(xxy)$$

For further compactness, we will sometimes express two (or more) consecutive λ abstractions $\lambda x \lambda y(...)$ as just $\lambda xy(...)$. Thus, our term can also be written

$$\lambda xy(xxy).$$

If you read other texts on the λ-calculus, you will quickly discover that different textbooks use different conventions for parentheses. Older books often also introduce a complex set of conventions involving dots in order to eliminate the need for parentheses altogether. Such conventions are perhaps useful for designing lean, efficient programs for manipulating λ-terms, but as that is not our goal we will let ourselves be guided instead by a desire to produce the most readable expressions possible. In this regard, note that our notational conventions overlap somewhat with ordinary mathematical practice. In particular, when a λ-term f is applied to an object c (which is of course another λ-term), the result in our notation is $f(c)$, which is a very familiar way of expressing the evaluation of a function.

At first, the λ-calculus might seem bizarre, and one might wonder what sorts of mathematical objects λ-terms really are. Some practice is required to get the hang of how the λ-calculus works – this will be provided in the next sections and the exercises. At the beginning, it is useful just to think of the untyped λ-calculus as a type of highly abstract formalism, and worry later about what sorts of mathematical objects λ-terms denote.

In order to talk a little more abstractly about the λ-calculus, it will help to introduce yet further notation. A λ-term like $\lambda x(xxyx)$ is $xxyx$ viewed as a unary function of x, and so one would expect that applying $\lambda x(xxyx)$ to a term t should produce $ttyt$. More generally, for any λ-term c, the function $\lambda x(M)$ applied to input c should have the value M', where M' is obtained by substituting c for every free occurrence of x in M. We will use the notation $M[c/x]$ to indicate the expression in which every free occurrence of x in M is replaced by c. (Note: a minority of texts use the notation $M[x/c]$ for this.) So for example, $xxyx[t/x]$ is $ttyt$, and $xxyx[t/y]$ is $xxtx$. The function $\lambda x(M)$ applied to input c will then have the value $M[c/x]$. For some more examples, the function $\lambda x(y(x))$ on input c is $y(x)[c/x]$ which is $y(c)$, and the function $\lambda x(x(x))$ on input c is $x(x)[c/x]$ which is $c(c)$. In summary:

Definition 1.3: Substitution notation.

The function $\lambda x(M)$ applied to c has the the value $M[c/x]$, where $M[c/x]$ is the expression in which every free occurrence of x in M is replaced by c.

There are actually some subtleties involved in this process of substitution, but these will be dealt with in the next sections.

It is important to emphasize that the substitution notation $M[c/x]$ is just a way for us to indirectly describe λ-terms built up using Definition 1.2. We are *not* here adding a clause to Definition 1.2 that says that if M and c are λ-terms and x is a variable, then $M[c/x]$ is a λ-term. The notation $M[c/x]$ is simply notation we use in talking about λ terms, but it is not part of the official syntactic structure of λ-terms themselves.

Manipulating some simple examples of λ-terms can help give a feel for their meaning. Let us begin with a very simple but central example; the term

$$\lambda x(x).$$

On any input c, this evaluates to $x[c/x]$, which is of course c. Thus $\lambda x(x)$ is a function whose output is always identical to its input, and so is just the identity function. Note however that this is a very abstract and powerful sort of identity function, which returns itself on literally *any* input. In particular, consider the term:

$$\lambda x(x)(\lambda x(x)). \tag{4}$$

This is the identity function applied to the identity function, which using our substitution rule is easily seen to simplify to the identity function $\lambda x(x)$ itself. In more detail, because (4) is the function $\lambda x(x)$ applied to $\lambda x(x)$, it has the value

$$x[\lambda x(x)/x]$$

i.e., the expression x with every free occurrence of x replaced with $\lambda x(x)$. This is just $\lambda x(x)$.

Of course, the idea of applying the identity function to itself is problematic from a set theoretic point of view, because insofar as we view a function as a set of ordered pairs, no function can take itself as an input. There is a sense then in which the functions we can define in the untyped λ-calculus are more general than anything that can be defined from a set-theoretic point of view.

For another example, consider the closed term

$$\lambda x(xx).$$

If one includes all parentheses, this term is $\lambda x(x(x))$. This term describes a function that takes x to $x(x)$ – that is, this term describes a function that takes an expression into that expression composed with itself. To see this more carefully, note that we have:

$$\lambda x(x(x))(f) = x(x)[f/x]$$

$$= f(f)$$

Note however that this is quite different from composing a function with itself in the traditional sense. Given a function $f : \mathbb{R} \to \mathbb{R}$ such as $f : x \to \sin(x)$, we can of course consider the function $g : x \to \sin(\sin(x))$, which is, in a sense, f composed with itself. However, this sort of self-application is *not* what we mean when we write a λ-term like $f(f)$. When we evaluate $\sin(\sin(x))$ in traditional mathematics, we apply the $\sin$ function to some *number* $\sin(x)$, where we are thinking of x as having being given. We are *not* applying the $\sin$ function to the $\sin$ function itself - that would be meaningless, as the $\sin$ function acts only on elements of $\mathbb{R}$, and not on functions. In contrast, when we write $f(f)$ in the λ-calculus, we are thinking of f as a function, and are applying this function to the function f itself. This is a very different (and less familiar) type of self-application. This must be kept in mind to avoid confusion.

Of course, for λ-terms f and x, the λ-term $f(f(x))$ does denote the function f applied to the output of the function f on input x. In this way, we can represent the more familiar type of function composition and iteration in the λ-calculus. We can write the λ-term

$$\lambda x(f(f(x))) \tag{5}$$

to denote the function that takes an input c into $f(f(c))$. To see in more detail that on input c the expression (5) gives output $f(f(c))$, note that we have

$$\begin{aligned} \lambda x(f(f(x)))(c) &= f(f(x))[c/x] \\ &= f(f(c)) \end{aligned}$$

We can also write

$$\lambda yx\ (y(y(x))) \tag{6}$$

to denote the function which, when provided with the two inputs f and c (in that order) produces the output $f(f(c))$. To see in more detail that on inputs f and c the expression (6) gives output $f(f(c))$, note that we have

$$\begin{aligned} \lambda yx\ (y(y(x)))(f) &= \lambda x(y(y(x)))[f/y] \\ &= \lambda x(f(f(x))) \end{aligned}$$

That on input c the function $\lambda x(f(f(x)))$ outputs $f(f(c))$ has already been shown. Thus, the closed λ-term (6) may be viewed as an abstract description of the process of applying a function to an argument twice. This more familiar type of iteration of a function f expressed in both (5) and (6) is quite different from the kind of iteration given by $f(f)$.

What the example of the unusual λ-term $f(f)$ brings out nicely is the fact that λ-terms are expressions that may be legally applied to *any* other expression. In this sense, they are quite unlike functions such as $\sin$ that are only defined on a limited domain. We will later put some restrictions on the λ-calculus to incorporate functions with limited domains, but until we do so,

we must accept that λ-terms are quite unlike the ordinary mathematical functions with which we are familiar.

For a final example, consider the λ-term

$$\lambda x(y(y)). \tag{7}$$

The free variable x does not appear inside $y(y)$, and so this term is a little unusual. If we input some c into this function, the output is $y(y)[c/x]$. This is $y(y)$ with every free occurrence of x replaced by c. Because there are no free occurrences of x in $y(y)$, this is just $y(y)$. So the output $y(y)$ of this function is the same for every input x. The function (7) is therefore simply the constant function whose output is always $y(y)$.

Before moving on, one further definition will be useful. We often want to talk about the *subterms* of a given λ-term. For example, the subterms of

$$x(\lambda x(x(y)))$$

are $x, y, x(y), \lambda x(x(y))$ and the term $x(\lambda x(x(y)))$ itself. Although an intuitive understanding of this notion will generally suffice, it is nevertheless useful to define it formally. To this end, we have the following definition

Definition 1.4: Subterms.

The notion of a *subterm* of a λ-term is defined inductively as follows

(i) If x is a variable, the only subterm of x is x.

(ii) If M and N are λ-terms, the subterms of $M(N)$ are $M(N)$, the subterms of M, and the subterms of N.

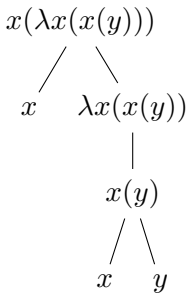
(iii) If M is a λ-term and x a variable, the subterms of $\lambda x(M)$ are $\lambda x(M)$ and the subterms of M.

In the λ-term $x(y(z))$, note that the subterm y is properly contained in the subterm $y(z)$. The subterm x is however disjoint from the subterm $y(z)$. In fact, these are in a sense the only possibilities. More specifically, it is impossible for two subterms to only partially overlap. Formally, we have the following:

Lemma 1.5: Subterm Lemma.

If t_1 and t_2 are distinct subterms of a λ-term s, then either t_1 is properly contained in t_2, t_2 is properly contained in t_1, or t_1 and t_2 are disjoint.

The proof is left to the exercises. As a consequence of this lemma, all the subterms of a given λ-term form a treelike structure under the relation of proper containment. For example, the subterms of $x(\lambda x(x(y)))$ may be represented by the following tree



where a subterm is located beneath another subterm in this tree iff it is properly contained in it. Note that there are two distinct subterms x of $x(\lambda x(x(y)))$. In such a situation, to be fully clear we talk about distinct *occurences* of the subterm x. Where there is no risk of confusion we simply talk about subterms simpliciter.

Exercises for Section 1.3

1. What is the value of the closed λ-term

$$\lambda x(\lambda y(x(x(x(y))))),$$

when applied to $x = f$ and $y = g$? Justify your answer in the way shown above, using the $M[c/x]$ style of notation.

2. What closed λ-term takes expressions f, g and h (in that order) into the expression

$$(f(g))(f(h))?$$

3. What function does the λ-term $\lambda x(x)(\lambda y(y))$ describe?

4. (a) Do the closed λ-terms

$$\lambda x(\lambda y(x(y)))$$

and

$$\lambda x(\lambda y(y(x)))$$

describe the same function? (Hint: apply both of these functions to an arbitrary f, and then apply the result to an arbitrary g. Are the results the same?)
(b) Do the closed λ-terms

$$\lambda x(\lambda y(x(y)))$$

and

$$\lambda y(\lambda x(y(x)))$$

describe the same function?

5. Are the following true or false? If false, give a counterexample. If true, simply state so.
(a) For any λ-term M, $\lambda x(M)$ is a closed λ-term.
(b) If $\lambda x(M)$ is a closed λ-term and N an arbitrary λ-term, then $\lambda x(M)(N)$ is a closed λ-term.
(c) If M and N are closed λ-terms, then $M(N)$ is a closed λ-term.

6. Prove Lemma 1.5. (Hint: use induction on the construction of s.)

1.4 α-equivalence

We argued earlier that $\lambda x(x)$ represents the identity function. But there is nothing special about x here. If $\lambda x(x)$ represents the identity function, then surely $\lambda y(y)$ or $\lambda z(z)$ also represents the identity function. Renaming bound variables should not change a λ-term in any meaningful way. In this and the next section we make this idea more general and precise. In the course of doing so some subtleties about the nature of substitution will arise that require careful attention.

The idea we would like to capture is that when a λ-term Y can be obtained from another λ-term X by renaming bound variables, there is some sense in which the λ-terms X and Y are the same. In this case, we will say that these terms are α-equivalent, and write

$$X =_\alpha Y.$$

This of course is not yet a fully precise or adequate definition of α-equivalence. By considering various problematic and unproblematic examples we will move towards a more rigorous definition of α-equivalence that we will write down shortly.

As a straightforward example, we will have

$$\lambda x(x) =_\alpha \lambda y(y). \tag{8}$$

Another simple example is

$$\lambda x(zxw) =_\alpha \lambda y(zyw) \tag{9}$$

and a further example is

$$\lambda x(z(x(\lambda y(y)))) =_\alpha \lambda u(z(u(\lambda v(v)))) \tag{10}$$

where in (10), two bound variables have been renamed.

We must however be a little careful. Consider the λ-term

$$\lambda x(y) \tag{11}$$

If we replace the bound variable x with y, we get the term

$$\lambda y(y) \tag{12}$$

The term (11) is the constant function which always returns the value y, and (12) is the identity function. We do *not* want to say that the terms (11) and (12) are the same, and thus do *not* want to say that these terms are α-equivalent.

The problem here is of course that starting with the term (11), we are trying to replace a bound variable x with a variable y that already appears free in the original term (11). Nothing like this happens in examples (8), (9) or (10), which are entirely unproblematic.

With this in mind, we define α-equivalence more carefully as follows. Given a term $\lambda u(E)$ and a variable v *which does not appear free in* E, we say that the terms $\lambda u(E)$ and $\lambda v(E[v/u])$ are α-equivalent. So for example, we have

$$\lambda x(x) =_\alpha \lambda y(x[y/x]) = \lambda y(y)$$

which gives us (8), and

$$\lambda x(zxw) =_\alpha \lambda y(zxw[y/x]) = \lambda y(zyw)$$

which gives us (9).

Let us make two further stipulations. First, we stipulate that the relation $=_\alpha$ is transitive. That is to say, we stipulate that for any λ-terms x, y and z, if $x =_\alpha y$ and $y =_\alpha z$, then $x =_\alpha z$.

Second, we stipulate that replacing a subexpression of a term with an α-equivalent subexpression gives an α-equivalent term. More precisely, if a term t contains a term s as a subexpression and $s =_\alpha s'$, then if t' is the result of replacing the subexpression s of t with s', then $t =_\alpha t'$. So for example, because $\lambda x(x) =_\alpha \lambda y(y)$, we have

$$z(\lambda x(x)) =_\alpha z(\lambda y(y))$$

(where we have let $t = z(\lambda x(x)), s = \lambda x(x), s' = \lambda y(y)$ and $t' = z(\lambda y(y))$), and

$$\lambda v(z(\lambda x(x))(x(y))) =_\alpha \lambda v(z(\lambda y(y))(x(y)))$$

(where we have let $t = \lambda v(z(\lambda x(x))(x(y))), s = \lambda x(x), s' = \lambda y(y)$ and $t' = \lambda v(z(\lambda y(y))(x(y)))$.)

Armed with all this, we then have for example

$$\lambda x(z(x(\lambda y(y)))) =_\alpha \lambda x(z(x(\lambda v(v))))$$

and

$$\lambda x(z(x(\lambda v(v)))) =_\alpha \lambda u(z(u(\lambda v(v)))).$$

From transitivity we then have (10).

We thus define α-equivalence rigorously as follows

Definition 1.6: α-equivalence.

Two λ-terms t and t' are α-equivalent iff there is a sequence of λ-terms $s_0, ..., s_n$ such that

(i) $t = s_0$ and $t' = s_n$, and
(ii) each s_{i+1} may be obtained from s_i by replacing some subexpression $\lambda v(E)$ of s_i with the expression $\lambda w(E[w/v])$, where w does not appear free in E.

Note that we regard every term as a subexpression of itself, so that for any λ-term E, E is a subexpression of E. Also note that the transitivity of $=_\alpha$ is not built into this definition, but is an easy proven consequence of it. In fact, it can easily be shown that $=_\alpha$ is an equivalence relation – that is, it can easily be shown that for any λ-terms x, y and z, (i) $x =_\alpha x$, (ii) if $x =_\alpha y$ then $y =_\alpha x$, and (iii) if $x =_\alpha y$ and $y =_\alpha z$, then $x =_\alpha z$. The argument for this is left to the exercises.

Using this definition we can show (10) by setting

$$s_0 = \lambda x(z(x(\lambda y(y))))$$
$$s_1 = \lambda x(z(x(\lambda v(v))))$$
$$s_2 = \lambda u(z(u(\lambda v(v))))$$

It is conventional to regard α-equivalent terms as notational variants of each other. Thus, it is conventional to think of $\lambda x(x)$ and $\lambda y(y)$ as the same λ-term, just written in two different ways. When we talk about λ-terms, we will therefore technically be talking about an equivalence class of terms or expressions, each of which is just a notational variant of the other in which bound variables are renamed. So for example, when we talk about the identity function, we may well write down $\lambda x(x)$ to notationally represent it, but what we really mean is the *class* of mutually α-equivalent λ-terms

$$\{\lambda x(x),\ \lambda y(y),\ \lambda z(z), ...\}$$

Because talking about classes of α-equivalent λ-terms is cumbersome, we will follow tradition by abusing notation, and will generally state our results in terms of individual λ-terms. This cuts down on what would otherwise be an excess of notation. The reader concerned with such details should easily be able to distinguish when we are talking about individual λ-terms and when we are really talking about equivalence classes of λ-terms under the relation of α-equivalence.

Because we regard $\lambda x(x)$ and $\lambda y(y)$ as the same term, we will also regard things out of which they are built – e.g., $z(\lambda x(x))$ and $z(\lambda y(y))$ – as the same term. Thus we will have

$$z(\lambda x(x)) =_\alpha z(\lambda y(y))$$

in accordance with Definition 1.6.

Exercises for Section 1.4

1. Are the following true or false?
(a) $x(x) =_\alpha y(y)$
(b) $x(\lambda x(x)) =_\alpha x(\lambda y(y))$
(c) $x(\lambda x(x)) =_\alpha y(\lambda y(y))$

2. Which of the following six expressions are α-equivalent?

$$\begin{array}{lll} \lambda x(x\lambda x(x)) & \lambda x(x\lambda y(y)) & \lambda x(y\lambda y(y)) \\ \lambda z(x\lambda z(x)) & \lambda z(x\lambda z(z)) & \lambda z(x\lambda x(z)) \end{array}$$

Where two or more expressions are α-equivalent, justify your answer using Definition 1.6.

3. Show that $=_\alpha$ is an equivalence relation. That is, show that for any λ-terms x, y and z

(i) $x =_\alpha x$
(ii) if $x =_\alpha y$ then $y =_\alpha x$, and
(iii) if $x =_\alpha y$ and $y =_\alpha z$, then $x =_\alpha z$.

4. Show that if E is a λ-term and v is a variable that does not appear free in E, then there is a λ-term E' such that $E =_\alpha E'$ and v does not appear at all in E. (Hint: use induction on the construction of E.)

5. Consider the following alternative to Definition 1.6

> Two λ-terms t and t' are α-equivalent iff there is a sequence of λ-terms $s_0, ..., s_n$ such that
>
> **(i)** $t = s_0$ and $t' = s_n$, and
> **(ii)** each s_{i+1} may be obtained from s_i by replacing some subexpression $\lambda v(E)$ of s_i with the expression $\lambda w(E[w/v])$, where w does not appear at all (either free or bound) in E.

Are there terms that are α-equivalent according to Definition 1.6, but not this definition, or vice versa? Either give an argument that these definitions agree in all cases, or give an example of a λ-term on which they differ.

1.5 Substitution Re-examined

One might have worries about the definition of α-equivalence given in the previous section. In particular, it looks like this definition entails the following

$$\lambda x(\lambda y(x)) =_\alpha \lambda y(\lambda y(x)[y/x]) = \lambda y(\lambda y(y)) \quad ? \tag{13}$$

Note that y does not appear free in $\lambda y(x)$, so this renaming of bound variables is legitimate. But $\lambda x(\lambda y(x))$ is a function which, on any input x, returns the constant function that always outputs x. On the other hand, $\lambda y(\lambda y(y))$ is a function that, on any input y, always outputs the identity function. These are different functions, and so should not be α-equivalent.

To resolve this problem, we must examine the nature of substitution more carefully. This will be the task of this section.

Let us start by considering the λ-term $M = \lambda x(xy)$. This has y as a free variable. Let us consider what happens when we substitute x for y; that is, let us calculate $M[x/y]$. Proceeding naively and replacing every free occurrence of y with x, it looks like we should get $M[x/y] = \lambda x(xx)$. This is a closed λ-term representing a function that takes a function x to xx - i.e., the application of x to itself.

However, $M = \lambda x(xy)$ is the same term as $M' = \lambda z(zy)$, as these terms are α-equivalent. But note that when we substitute x for the free variable y in M', we find that $M'[x/y]$ is $\lambda z(zx)$. This is not even a closed λ-term. We are left with the very awkward situation in which terms M and M' that we want to regard as the same (because they are α-equivalent) return different values when the same thing (namely, x) is substituted for the same free variable (namely, y) in each. This is unpleasant. How should we resolve this problem?

Note that there is something peculiar about the expression $\lambda x(xy)[x/y]$ insofar as x plays two different roles in it - first, it is something that we are substituting for y, and second, it is a variable that is already bound by λx in $\lambda x(xy)$. Because of this, the variable x that we substitute, while free on its own, is 'captured' and becomes a bound variable in the final expression $\lambda x(xx)$. More generally, when evaluating $\lambda x(xy)[E/y]$ in a case in which E contains x as a free variable, when E is substituted for y in $\lambda x(xy)$ the free variable x in E is 'captured' by the λx and becomes a bound variable. This seems undesirable, and is the source of the unpleasantness described in the previous paragraph. The natural thing to do is to prohibit this, and insist that in evaluating $\lambda x(xy)[E/y]$ we substitute E for all free ocurrences of y in a representation of $\lambda x(xy)$ in which this sort of capturing does not happen. If for example we rewrite $\lambda x(xy)$ as the α-equivalent term $\lambda z(zy)$ and *then* do the substitution in question, we get $\lambda z(zy)[x/y]$, which becomes $\lambda z(zx)$, and x remains free.

More generally, we will say that when evaluating $N[E/y]$ where y is a free variable in N, we must substitute E for y in a term N' α-equivalent to N such that no 'capturing' of free variables occurs. In particular, we must choose some N' which is α-equivalent to N such that no free variable of E becomes bound when E is substituted for every free occurrence of y in N'. The term $N[E/y]$ is then the result of substituting E for every free occurrence of y in such an N'. Even though different choices of N' are possible, it is easily seen that the final results are all α-equivalent. So for example, we could have chosen $\lambda w(wy)$ instead of $\lambda z(zy)$ for N' in our above example, in which case $\lambda x(xy)[x/y]$ would become $\lambda w(wx)$ instead of $\lambda z(zx)$. That is fine, as $\lambda w(wx)$ and $\lambda z(zx)$ are α-equivalent, and are thus merely different representations of the same term.

Definition 1.7: Capturing.

When performing the substitution $N[E/y]$, a free variable x occurring in E is *captured* if some free occurrence of y in N occurs inside the scope of a λx operator – that is, if N has a subexpression of the form $\lambda x(...y...)$, where this occurrence of y is free in N.

Definition 1.8: Substitution.

The λ-term $N[E/y]$ is the term N' with every free occurrence of y replaced by E, where N' is a λ-term α-equivalent to N such that no free variable of E is captured during this substitution.

That such an appropriate term N' always exists is something you will demonstrate in the exercises.

Of course, in many cases where there is no risk of capturing we may calculate $N[E/y]$ naively by substituting E for y without carefully finding an appropriate N'. It is nevertheless important to know how to proceed when this is not the case.

Let us return now to (13). Note that in the term $\lambda y(x)[y/x]$, if we naively substitute y for every free occurrence of x, we get $\lambda y(y)$, and the variable y has been captured. Replacing $\lambda y(x)$ with the α-equivalent term $\lambda z(x)$, the substitution $\lambda z(x)[y/x]$ gives the result $\lambda z(y)$, and there is no capturing. So $\lambda y(x)[y/x]$ is $\lambda z(y)$ (up to α-equivalence.) Thus, equation (13) is incorrect, and must be replaced by

$$\lambda x(\lambda y(x)) =_\alpha \lambda y(\lambda y(x)[y/x]) = \lambda y(\lambda z(y)) \tag{14}$$

which is unproblematic. A correct understanding of substitution therefore saves our definition of α-equivalence from the unpleasant example in question.

Another unusual case worth discussing is that of terms in which the same variable is bound multiple times. Consider for example the term

$$\lambda x(x\lambda x(\underset{*}{\underset{\uparrow}{x}}y)) \tag{15}$$

This term contains λx twice. Consider the x labelled by $*$ in the above expression. Which λx should we regard this as bound by? By applying Definition 1.6, we see that (15) is α-equivalent to

$$\lambda x(x\lambda z(zy)),$$

and the x in question has become a z. This shows us that in (15), we must regard the x in question as bound by the rightmost λx, and not the leftmost λx. (Note that we *cannot* apply Definition 1.6 to argue that (15) is α-equivalent to $\lambda z(z\lambda x(zy))$, and so there is no analogous argument for regarding the x in question as bound by the leftmost λx.) In general then, any bound variable x which occurs in the scope of multiple λx abstractions will be bound by the λx that occurs closest

to the left of it. Note then that on input c, the function (15) outputs the value $(x\lambda x(xy))[c/x]$, which is $c\lambda x(xy)$, and not $c\lambda x(cy)$ or $x\lambda x(cy)$.

Finally, while a more intuitive grasp of substitution suffices for the ordinary manipulations of λ-terms encountered in much of this book, on occasions it is useful to work with a more rigorous definition of substitution, which we now present. This can safely be skipped on a first reading. In this rigorous definition, $N[E/v]$ is defined inductively as follows:

(i) $v[E/v]$ is E
(ii) if x is a variable distinct from v, then $x[E/v]$ is x
(iii) $N(N')[E/v]$ is $N[E/v](N'[E/v])$
(iv) $\lambda v(N)[E/v]$ is $\lambda v(N)$
(v) if x is a variable distinct from v, and either x does not occur free in E or v does not appear free in N, then $\lambda x(N)[E/v]$ is $\lambda x(N[E/v])$
(vi) if x is a variable distinct from v, and x occurs free in E and v occurs free in N, then $\lambda x(N)[E/v]$ is $\lambda z(N[z/x][E/v])$, where z is a variable that does not appear free or bound in N or E.

Of course, this definition only generates one of the set of α-equivalent terms representing $N[E/v]$.

Note that by relabeling the bound variable x of $\lambda x(N)$, case (vi) can always be avoided. However, if it seems circular to make use of the notion of replacement of bound variables in the definition of substitution, clause (vi) may be included, and no reference to the notion of relabeling of bound variables is then necessary.

As an example, we evaluate $\lambda r(r(\lambda s(t)))[r/t]$ using these rules.

Starting with	$\lambda r(r(\lambda s(t)))[r/t]$
Applying rule (vi), we get	$\lambda z(r(\lambda s(t))[z/r][r/t])$
Applying rule (iii), we get	$\lambda z(r[z/r](\lambda s(t)[z/r])[r/t])$
Applying rules (i) and (v), we get	$\lambda z(z(\lambda s(t[z/r]))[r/t])$
Applying rule (ii), we get	$\lambda z(z(\lambda s(t))[r/t])$
Applying rule (iii), we get	$\lambda z(z[r/t](\lambda s(t)[r/t]))$
Applying rule (ii) and (v), we get	$\lambda z(z(\lambda s(t[r/t])))$
Applying rule (i), we get	$\lambda z(z(\lambda s(r)))$

Thus $\lambda r(r(\lambda s(t)))[r/t]$ evaluates to $\lambda z(z(\lambda s(r)))$, as expected.

Using this definition of substitution, facts such as the following can be rigorously proven:

Theorem 1.9: Substitution Theorem.

If x and y are distinct variables and x does not occur free in P, then

$$N[M/x][P/y] \text{ is identical with } N[P/y][M[P/y]/x]$$

The proof is left to the exercises.

Exercises for Section 1.5

1. Simplify the following expressions as far as possible, making sure to respect the requirement that free variables not be captured in substitutions. Here x, y and z are to be understood as distinct variables. (You do not have to evaluate these expressions using the more rigorous definition of substitution presented at the end of the section; instead argue more informally.)

(i) $x(x)[y/x]$
(ii) $y(x)[y/x]$
(iii) $\lambda x(y)[y/x]$
(iv) $\lambda x(yx)[y/x]$
(v) $\lambda x(yx)[x/y]$
(vi) $x\lambda x(yx)[x/y]$
(vii) $x(\lambda y(xy))[\lambda x(xy)/x]$
(viii) $x(\lambda y(xy))[\lambda y(xy)/x]$

2. Given any λ-terms N, E and a variable y, show that there is a term N' α-equivalent to N such that no free variable of E is captured in the substitution $N'[E/y]$.

3. Shown that $=\alpha$ is an equivalence relation. That is, show that for any λ-terms x, y and z, (i) $x =_\alpha x$, (ii) if $x =_\alpha y$ then $y =_\alpha x$, and (iii) if $x =_\alpha y$ and $y =_\alpha z$, then $x =_\alpha z$.

4. Perform the substitutions of problem **1. (iv), (v), (vi)** and **(vii)** using the rigorous definition of substitution presented at the end of the section.

5. Prove Theorem 1.9. (Hint: use induction on the construction on N. When considering the case in which N has the form $\lambda z(\overline{N})$, you may assume by relabelling of bound variables that z is distinct from x and y, and that z does not appear free in M or P.)

6. Show that if x and y are distinct variables and x is allowed to occur free in P, then we do not necessarily have that $N[M/x][P/y]$ is identical with $N[P/y][M[P/y]/x]$.

1.6 β-reduction and Equivalence

In the λ-calculus, $\lambda x(M)$ is the expression M viewed as a function of x. This means that it takes the input N to the output $M[N/x]$. But the application of the function $\lambda x(M)$ to N can also be written as the expression $\lambda x(M)(N)$. The expression $\lambda x(M)(N)$ should therefore simplify (in some sense) to $M[N/x]$. We call the act of simplifying

$$\lambda x(M)(N)$$

to

$$M[N/x]$$

β-reduction, and say that $\lambda x(M)(N)$ *β-reduces* to $M[N/x]$. The expression $\lambda x(M)(N)$ is sometimes called a *β-redex*, or just a *redex* (i.e., something that can be *reduced*.) The act of β-reducing a redex is really nothing more than the act of evaluating a function on a given input.

For example, consider the term

$$\lambda x(x(x))(\lambda z(z)).$$

This has the form $\lambda x(M)(N)$ with M the expression $x(x)$ and N the expression $\lambda z(z)$. It therefore β-reduces to $x(x)[\lambda z(z)/x]$, which is just $\lambda z(z)(\lambda z(z))$. This resulting term also has the form $\lambda z(M)(N)$ with M the expression z and N the expression $\lambda z(z)$. It therefore β-reduces to $z[\lambda z(z)/z]$, which is just $\lambda z(z)$. When A β-reduces to B and B β-reduces to C, we also say that A β-reduces to C. Thus $\lambda x(x(x))(\lambda z(z))$ β-reduces to $\lambda z(z)$.

We will allow β-reduction to be performed on a redex inside a term. Thus, the term

$$\tau_1\ \lambda x(M)(N)\ \tau_2$$

β-reduces to

$$\tau_1\ M[N/x]\ \tau_2.$$

where τ_1 and τ_2 are sequences of symbols. For example, consider the term

$$\lambda y(\lambda x(x(x))(\lambda z(z))(y))$$

This has the form $\tau_1\ \lambda x(M)(N)\ \tau_2$ with τ_1 the sequence of symbols $\lambda y($, and M the expression $x(x)$, and N the expression $\lambda z(z)$, and τ_2 the sequence of symbols $(y))$ – i.e.,

$$\underbrace{\lambda y(}_{\tau_1}\ \lambda x\underbrace{(x(x))}_{\text{M}}\underbrace{(\lambda z(z))}_{\text{N}}\ \underbrace{(y))}_{\tau_2}$$

It therefore β-reduces to

$$\lambda y(\ x(x)[\lambda z(z)/x]\ (y))$$

which is

$$\lambda y(\lambda z(z)(\lambda z(z))(y)).$$

This in turn has the form $\tau_1\ \lambda z(M)(N)\ \tau_2$ with τ_1 the expression $\lambda y($ and M the expression z and N the expression $\lambda z(z)$ and τ_2 the expression $(y))$ – i.e.,

$$\underbrace{\lambda y(}_{\tau_1}\ \lambda z\underbrace{(z)}_{\text{M}}\underbrace{(\lambda z(z))}_{\text{N}}\ \underbrace{(y))}_{\tau_2}$$

It therefore β-reduces to

$$\lambda y(\ z[\lambda z(z)/z]\ (y))$$

which is

$$\lambda y(\lambda z(z)(y)).$$

You should be able to see that by similar reasoning, this β-reduces to $\lambda y(y)$, which cannot be β-reduced any further. Thus the original expression $\lambda y(\lambda x(x(x))(\lambda z(z))(y))$ β-reduces to $\lambda y(y)$.

One must be careful here and pay very close attention to the presence or absence of parentheses. In order to apply β-reduction within a term, the overall term just have the *exact* form

$$\tau_1\ \lambda x(M)(N)\ \tau_2.$$

In particular, there must a set of parentheses immediately surrounding both M and N as depicted, and *nothing* between (M) and (N). With this is mind, consider the exaxmple:

$$y(\lambda x(x))(z).$$

We *cannot* apply the function $\lambda x(x)$ to z and β-reduce the whole expression to yz, letting τ_1 be $y($ and M be the expression x and N be the expression z and τ_2 the empty expression – i.e.,

$$\underbrace{\lambda y(}_{\tau_1}\ \lambda x(\underset{\text{M}}{\underbrace{x}})(\underset{\text{N}}{\underbrace{z}})$$

because (M) does not sit *immediately* next to (N), due to the intervening). The λ-term $y(\lambda x(x))(z)$ in fact contains no redexes, and thus cannot be β-reduced in any way.

Determining whether it is correct to perform a β-reduction can sometimes be confusing when have used our conventions to abbreviate expressions and remove parentheses. To check whether one can apply β-reduction within a term, it is often best to restore any missing parentheses first, just to be sure. Paying attention to possibly suppressed parentheses is especially important also because of the failure of associativity of the λ-calculus: the λ-terms $x(yz)$ and $(xy)z$ are not in general the same in the λ-calculus, and so the placement of parentheses can completely change the meaning of a λ-term. You will explore this point in the exercises.

When a λ-term A reduces to B by a single β-reduction, we write $A \to_\beta B$. When A reduces to B by a *sequence* of β-reductions (this includes the possibility that the sequence consists of just a single β-reduction), we write $A \twoheadrightarrow_\beta B$. So we have just seen that

$$\lambda y(\lambda x(x(x))(\lambda z(z))(y)) \twoheadrightarrow_\beta \lambda y(y).$$

We also write $A \twoheadrightarrow_\beta A$ for any term A, thinking of there as being a '0 step' β-reduction of A to A.

We summarize these definitions as follows.

Definition 1.10: β-reduction.

An expression of the form $\lambda x(M)(N)$ (where x is a variable and M, N are λ-terms) is called a *β-redex* (or just a *redex*). We have:

(i) $A \to_\beta B$ iff A has the form $\tau_1\ \lambda x(M)(N)\ \tau_2$ and B has the form $\tau_1\ M[N/x]\ \tau_2$.

(ii) $A \twoheadrightarrow_\beta B$ (A β-reduces to B) iff for some sequence $X_0, X_1, ..., X_n$ of λ-terms we have that

$A = X_0$, $B = X_n$, and $X_0 \to_\beta X_1 \to_\beta ... \to_\beta X_n$. This includes the degenerate case in which A and B are identical.

The set of rules of formation of λ-terms along with the rule of β-reduction just given can be thought of as a system of rules for manipulating terms constructed in a certain sort of way. This system is typically called the *untyped λ-calculus*. We will also call it λ_0, to easily distinguish it from expansions of the untyped λ-calculus that we consider later.

Definition 1.11: Rules of the Untyped λ-calculus λ_0.

Rules of λ-term formation

(i) Any variable $x, y, z, ...$ is a λ-term.
(ii) If M and N are λ-terms, then so is $M(N)$.
(iii) If M is a λ-term and x a variable, then $\lambda x(M)$ is a λ-term.

Rule of β-reduction

(i) $\lambda x(M)(N) \to_\beta M[N/x]$

Note that for compactness, in the above definition we state only the simplest form of the rule of β-reduction, with it being understood that β-reduction may be performed on redexes inside terms.

We now have the following important definition

Definition 1.12: β-normal form.

A term X is in *β-normal form* just in case no β-reduction may be performed on it – that is, just in case it contains no subexpression of the form $\lambda x(M)(N)$.

For example, $\lambda y(y)$ is in β-normal form. By performing a sequence of β-reductions, we reduced $\lambda y(\lambda x(x(x))(\lambda z(z))(y))$ to the β-normal form term $\lambda y(y)$. One might hope that starting with any term and continually β-reducing it, one will eventually arrive at a term in β-normal form. Unfortunately this is not the case. For example, consider the term

$$\lambda x(x(x))(\lambda x(x(x))).$$

We have

$$\begin{aligned} \lambda x(x(x))(\lambda x(x(x))) &\to_\beta x(x)[\lambda x(x(x))/x] \\ &= \lambda x(x(x))(\lambda x(x(x))) \end{aligned}$$

Although the term $\lambda x(x(x))(\lambda x(x(x)))$ is not in β-normal form, the only act of β-reduction that may be performed on it does not simplify the original term, but rather just returns the term itself.

There is thus no sequence of β-reductions that begins with this term and ends with a term in β-normal form.

A yet further complication that needs to be considered is that there are λ-terms on which there are *multiple* β-reductions that can be performed, because they contain multiple redexes. For example, consider the term

$$\lambda x(x)(y(y))(\lambda z(z)(y))$$

If we perform a β-reduction on the redex $\lambda x(x)(y(y))$, this redex becomes $y(y)$, and so the whole term β-reduces to $y(y)(\lambda z(z)(y))$. This in turn β-reduces to $y(y)(y)$ (i.e., yyy), which is in β-normal form.

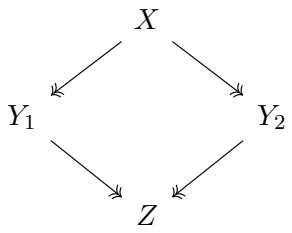
However, we can also perform a β-reduction on the redex $\lambda z(z)(y)$ first. Doing so, the entire term is β-reduced to $\lambda x(x)(y(y))(y)$, which in turn β-reduces to $y(y)(y)$ (i.e. yyy.)

In this case, even though there are different 'orders' in which we can β-reduce our given term, different β-reductions ultimately give the same β-normal term. Is this true in general? It turns out that it is. This is a consequence of the following important general fact:

Theorem 1.13: The Church-Rosser Theorem for β-reductions in λ_0.

In the untyped λ-calculus λ_0, if $X \twoheadrightarrow_\beta Y_1$ and $X \twoheadrightarrow_\beta Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$.

This theorem is often depicted as follows:



This theorem tells us that if different sequences of β-reductions take a term X to different terms Y_1, Y_2, then *further* β-reductions can take Y_1 and Y_2 back to a common term Z. So any distinct sequences of β-reductions can be 'reunited', forming the diamond shape shown above.

It follows from this that when a term X *can* be reduced to a β-normal form Y, this Y is unique (up to α-equivalence). That is to say, it follows that different chains of β-reductions cannot take us from an X to two distinct β-normal forms Y_1 and Y_2. To see why, suppose to the contrary that two different chains of β-reductions take us from a term X to two β-normal forms Y_1 and Y_2. It follows from the Church-Rosser Theorem just stated there is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$. But if Y_1 and Y_2 are in β-normal form then they cannot be reduced any further, and so we must have that Y_1 and Z are the same term (up to α-equivalence) and Y_2 and Z are the same term (up to α-equivalence). By transitivity of α-equivalence, it follows that Y_1 and Y_2 are also α-equivalent.

So if different chains of β-reductions take us from X to distinct β-normal forms Y_1 and Y_2, then Y_1 and Y_2 are α-equivalent.

The proof of the Church-Rosser Theorem is perhaps surprisingly complex. We include it in the appendix to this chapter.

Intuitively, β-reduction should be thought of as a process taking a λ-term into another λ-term referring to the same entity. Consider then the following 'symmetrical' version of one-step β-reduction:

Definition 1.14: The relation $\sim_\beta$.

$X \sim_\beta Y$ if either $X \to_\beta Y$ or $Y \to_\beta X$

We have $X \sim_\beta Y$ iff there is a one-step β-reduction from one of X or Y to the other. If $X \sim_\beta Y$, then X and Y should refer to the same entity. Because 'referring to the same entity' is surely a transitive relation, we should then have that if $X_0 \sim_\beta X_1$, $X_1 \sim_\beta X_2$, ..., $X_{n-1} \sim_\beta X_n$, then X_0 and X_n 'refer to the same entity'. With this in mind, we define the relation of *β-equivalence* as follows:

Definition 1.15: The relation $U =_\beta V$.

U and V are *β-equivalent* (in symbols, $U =_\beta V$) iff one of the following holds:

(a) U and V are the same term (up to α-equivalence), or
(b) for some sequence of terms $X_0, X_1, ..., X_{n-1}, X_n$ with $U = X_0$ and $V = X_n$,

$$X_0 \sim_\beta X_1,\ X_1 \sim_\beta X_2, ...,\ X_{n-1} \sim_\beta X_n.$$

Although when U and V are β-equivalent we regard them as referring to the same thing, this does *not* mean that we regard them as the same expression. (This is a common feature of any language. In English the terms 'Prince Charles' mother' and 'Queen Elizabeth II' refer to the same person, even though they are different expressions.) In this sense, what is being said here is different from what was said about α-equivalence, where certain distinct lexical items were regarded as literally the same expression. We do not want to make that move here, as we want it to be possible for syntactically distinct terms to have a common referent.

The following is a useful corollary of the Church-Rosser Theorem

Corollary 1.16

In λ_0, $U =_\beta V$ iff there is a term Z such that $U \twoheadrightarrow_\beta Z$ and $V \twoheadrightarrow_\beta Z$.

You will prove this in an exercise. This corollary tells us that two λ-terms are β-equivalent iff they can be β-reduced to a common term.

A final fact about β-reduction that is occasionally useful is the following

Lemma 1.17:

For any λ_0-terms U, V, t and variable x, if $U \to_\beta V$ then $U[t/x] \twoheadrightarrow_\beta V[t/x]$

The proof is also left to the exercises.

Exercises for Section 1.6

1. Reduce each of the following lambda terms into β-normal form. In all cases, if an expression is already in β-normal form, you simply need to point this out and do nothing further. Here, x, y, z, u and v are to be understood as distinct variables.

(i). $\lambda x(x)(y)$
(ii). $\lambda x(E)(z)$ (Here E is a term that contains no free occurrence of x.)
(iii). $\lambda x(x)(\lambda y(y(y)))$
(iv). $\lambda x(x)(\lambda y(y(y))(\lambda z(z)))$
(v). $\lambda x(y(\lambda y(z)))$
(vi). $\lambda x(\lambda y(xy))(u)$
(vii). $\lambda x(\lambda y(xy))(\lambda u(u(v)))$
(viii). $\lambda x(\lambda y(xy))(\lambda u(\lambda v(u(v))))$

2. In this exercise, we show that the application of λ-terms is not associative. In particular, we show that it is *not* the case that for all λ-terms a, b and c, $a(b)(c)$ and $a(b(c))$ are β-equivalent. Let x and y be distinct variables, and let a be $\lambda x(y)$, b be y, and c be y. Show that $a(b)(c)$ β-reduces to $y(y)$ and $a(b(c))$ β-reduces to y. Because both y and $y(y)$ are in β-normal form and are different terms (i.e., not α equivalent), it follows that y and $y(y)$ are not β-equivalent terms.

3. In spite of the previous exercise, the λ-calculus captures the fact that function composition is associative. Given functions (i.e., λ-terms) f and g, define $[f \circ g]$ to be $\lambda x(f(g(x))$. Show that the terms $[f \circ g](h(x))$ and $f([g \circ h](x))$ are β-equivalent. How are the results of this and the previous exercise compatible? Doesn't one exercise show that function composition is associative, and the other show that it is not?

4. Argue that if s is a closed λ-term and $s \twoheadrightarrow_\beta t$, then t is a closed λ-term. (A closed term is one with no free variables.)

5. Argue that every β-normal closed term can be written in the form

$$\lambda x_1 ... \lambda x_n (x(M_1)...(M_m))$$

where $n > 0, m \geq 0$, x and the x_i are (not necessarily distinct) variables, and the M_i are all in β-normal form. (Hint: prove this by induction on the construction of λ-terms.)

6. Using the Church-Rosser Theorem, prove Corollary 1.16.

7. Show that there is a λ-term τ which has a non-terminating β-reduction sequence - i.e., an infinite sequence $\tau \to_\beta \tau_1 \to_\beta \tau_2 \to_\beta \tau_3 \to_\beta \cdots$ - as well as a terminating β-reduction sequence - i.e., a finite sequence $\tau \to_\beta \sigma_1 \to_\beta \sigma_2 \to_\beta \cdots \to_\beta \sigma_i$ for some finite i, where σ_i is in β-normal form.

8. Find terms X and Y such that neither X nor Y has a β-normal form, but $X(Y)$ has a β-normal form.

9. Let Y be the term $\lambda x(\lambda y(x(yy))(\lambda y(x(yy))))$. For every term E, show that

$$E(Y(E)) =_\beta Y(E).$$

Thus, $Y(E)$ is a 'fixed point' of E.

10. Show that the term Y of the previous problem does not have a β-normal form. Show that in fact for any term Y' such that for every term E, $E(Y'(E)) =_\beta Y'(E)$, Y' does not have a β-normal form. (Hint: let x be a variable that does not appear free in Y', and consider the fact that $x(Y'(x)) =_\beta Y'(x)$.)

11. Prove Lemma 1.17. (Hint: by α-equivalence, you may assume that neither x nor any free variable of t is bound in V.)

12. Is it true that for any λ-terms U, V, t and variable x, if $U \to_\beta V$ then $U[t/x] \to_\beta V[t/x]$?

1.7 η-reduction and Extensionality

Suppose E is a term in the λ-calculus that does not contain x as a free variable. Consider the term $E^* = \lambda x(E(x))$. On any input c, we have

$$\begin{aligned} E^*(c) &= \lambda x(E(x))(c) \\ &=_\beta E(x)[c/x] \\ &= E(c) \end{aligned}$$

As functions, E^* and E produce the same outputs on the same inputs. It is thus tempting to think of them as the same function.

Note that if we do not require that E does not contain x as a free variable, then $E^*(c)$ is not necessarily β-equivalent to $E(c)$. To see this, let x and y be distinct variables, and let E just be the variable x. Then $E(y)$ is $x(y)$, but $E^*(y)$ is $\lambda x(x(x))(y)$, which β-reduces to $y(y)$, and so $E(y)$ and $E^*(y)$ are not β-equivalent.

Suppose then that E does not contain x as a free variable, so that E^* and E produce the same outputs on the same inputs. In spite of this, nothing we have said so far forces us to identify E^* and E in any sense. Consider for example the special case in which E is just the variable y. In this case, E^* is $\lambda x(yx)$. Both y and $\lambda x(yx)$ are terms in β-normal form, and are not α-equivalent. Thus, they are not β-equivalent either. So E^* and E are not equivalent in any formal sense we have identified yet.

In the λ-calculus as we have developed it thus far, we do not necessarily treat a pair of functions as identical just because they always give the same outputs on the same inputs. Is this a defect or a virtue? Consider the following principle

Definition 1.18: Principle of Extensionality.

If two terms λ-terms U and V have the property that for every λ-term c, $U(c)$ and $V(c)$ are the same, then U and V are the same.

This principle is not fully precise, insofar as the term 'same' has not yet been defined. However, if by 'same' we mean β-equivalent, then we have seen that this principle fails.

Perhaps this is not worrisome. One might, after all, have reservations about the Principle of Extensionality. Must we really identify two functions or algorithms that each return the same output on the same inputs? Two algorithms that always return the same output on the same inputs might still be different in many other respects - one algorithm might require significantly more time or memory to produce an output than the other, for example. Whether this sort of distinction matters will depend on our purposes. If we are interested in implementations of functions on real physical computers, such distinctions may well be important. If by contrast we are only interested in functions from a very abstract mathematical point of view, such distinctions may well not matter. There is no right or wrong answer as to whether we must endorse the Principle of Extensionality. It simply depends on whether doing so is useful to our ends, whatever those are.

Suppose however that we are atttracted to the Principle of Extensionality, and are interested in articulating a notion of sameness according to which it holds. How might we proceed? To start with, for any term E we will want to identify $\lambda x(E(x))$ (where x is not free in E) with E. To this end, we introduce a new reduction rule - the rule of *η-reduction* - according to which the term $\lambda x(E(x))$ (with x not free in E) can be reduced to E, as follows

If x not free in E, then $\lambda x(E(x)) \rightarrow_\eta E$.

In this definition, we are of course free to use any variable in place of x. The expression $\lambda x(E(x))$ is sometimes called an *η-redex*, or just a *redex*. As with β-reduction, we allow ourselves to apply

η-reductions to η-redexes within terms. In this way, the term

$$\tau_1 \ \lambda x(E(x)) \ \tau_2$$

also η-reduces to

$$\tau_1 E \tau_2$$

(where τ_1 and τ_2 are sequences of symbols, and x is not free in E.)

We thus regard both β-reduction and η-reduction as valid ways of simplifying terms. In this spirit, we define $X \rightarrow_{\beta\eta} Y$ iff $X \rightarrow_\beta Y$ or $X \rightarrow_\eta Y$, and $X \twoheadrightarrow_{\beta\eta} Y$ iff some sequence of β or η reductions takes X to Y (where again, this includes the degenerate '0 step' reduction in which X and Y are α-equivalent.) More formally, we have the following definitions:

Definition 1.19: η-reduction, $\beta\eta$-reduction, and $\beta\eta$-equivalence

An expression of the form $\lambda x(E(x))$ (where x is a variable and E a λ-term such that x does not appear free in E) is called an η-redex (or just a *redex*). We have:

(a) $A \rightarrow_\eta B$ iff A has the form $\tau_1 \ \lambda x(E(x)) \ \tau_2$ and B has the form $\tau_1 \ E \ \tau_2$ for expressions τ_1, E, τ_2, and x is not free in E.

(b) $A \rightarrow_{\beta\eta} B$ iff $A \rightarrow_\beta B$ or $A \rightarrow_\eta B$.

(c) $A \twoheadrightarrow_{\beta\eta} B$ (A $\beta\eta$-reduces to B) iff for some sequence $X_0, X_1, ..., X_n$ of λ-terms we have that $A = X_0$, $B = X_n$, and $X_0 \rightarrow_{\beta\eta} X_1 ... \rightarrow_{\beta\eta} X_n$. This includes the degenerate case in which A and B are identical.

(d) X and Y are *$\beta\eta$-equivalent* (in symbols, $X =_{\beta\eta} Y$) iff one of the following holds:

(i) X and Y are the same term (up to α-equivalence), or
(ii) for some sequence of terms $X_0, X_1, ..., X_{n-1}, X_n$ with $U = X_0$ and $V = X_n$,

$$X_0 \sim_{\beta\eta} X_1, \ X_1 \sim_{\beta\eta} X_2, ..., \ X_{n-1} \sim_{\beta\eta} X_n.$$

In fact, $\beta\eta$-reduction and $\beta\eta$-equivalence have many of the same important properties that β-reduction and β-equivalence have. For example

Theorem 1.20: The Church-Rosser Theorem for $\beta\eta$-reductions in λ_0.

In the untyped λ-calculus λ_0, if $X \twoheadrightarrow_{\beta\eta} Y_1$ and $X \twoheadrightarrow_{\beta\eta} Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_{\beta\eta} Z$ and $Y_2 \twoheadrightarrow_{\beta\eta} Z$.

Again, the proof of this result is left to the appendix. As before, the following corollary is easy:

Corollary 1.21

In λ_0, $U =_{\beta\eta} V$ iff there is a term Z such that $U \twoheadrightarrow_{\beta\eta} Z$ and $V \twoheadrightarrow_{\beta\eta} Z$.

We also have the following definition

Definition 1.22: $\beta\eta$-normal form.

A term X is in *$\beta\eta$-normal form* just in case no β-reduction or η-reduction may be performed on it – i.e., just in case it contains no subexpression of the form $\lambda x(M)(N)$ or of the form $\lambda x(E(x))$, where x is not free in E.

Corollary 1.21 then shows that $\beta\eta$-equivalent terms can be $\beta\eta$-reduced to a common term. It also follows from Theorem 1.20 as before that when a term can be reduced to a term in $\beta\eta$-normal form, this term in $\beta\eta$-normal form is unique (up to α-equivalence.)

In analogy with Lemma 1.17, a further useful fact about η-reduction is the following

Lemma 1.23:

For any λ_0-terms U, V, t and variable x, if $U \rightarrow_\eta V$ then $U[t/x] \twoheadrightarrow_\eta V[t/x[$

The proof is also left to the exercises.

An interesting technical property of $\beta\eta$-reductions is that whenever an η-reduction is followed by a β-reduction, there is a sense in which the order of these reductions can be reversed, so that the β-reduction comes before the η-reduction. More precisely, we have the following lemma

Lemma 1.24: η-reduction postponement in λ_0.

For all λ-terms X, Y, Z, if $X \rightarrow_\eta Y \rightarrow_\beta Z$ in λ_0, then there is a λ-term Y' such that $X \twoheadrightarrow_\beta Y' \twoheadrightarrow_\eta Z$ (where $\twoheadrightarrow_\eta$ is a sequence of η-reductions.)

You will prove this in the exercises. As a consequence, it can be shown that if a term can be reduced to $\beta\eta$-normal form, then there is a way of reducing it to $\beta\eta$-normal form that involves performing a sequence of β-reductions followed by a sequence of η-reductions.

All things considered, the notion of $\beta\eta$-equivalence is a natural notion with many pleasing properties. In fact, the notion of $\beta\eta$-equivalence provides us with a notion of 'sameness' which it turns out is sufficient to demonstrate the Principle of Extensionality. In particular, we have the following

Theorem 1.25: Principle of Extensionality for $\beta\eta$-equivalence.

In the untyped λ-calculus λ_0, if two terms λ-terms U and V have the property that for every λ-term c, $U(c) =_{\beta\eta} V(c)$, then $U =_{\beta\eta} V$.

Proof

Suppose we have λ-terms U and V such that for every λ-term c, $U(c) =_{\beta\eta} V(c)$. Let z be a variable that does not appear free in U or V. Then $U(z) =_{\beta\eta} V(z)$. In an exercise, you will show that if E_1 and E_2 are λ-terms such that $E_1 =_{\beta\eta} E_2$ and z is a variable, then $\lambda z(E_1) =_{\beta\eta} \lambda z(E_2)$. We therefore have that $\lambda z(U(z)) =_{\beta\eta} \lambda z(V(z))$. Because $\lambda z(U(z)) =_\eta U$ and $\lambda z(V(z)) =_\eta V$, combining these and using the symmetry and transitivity of $=_{\beta\eta}$ gives $U =_{\beta\eta} V$. $\square$

The relation $=_{\beta\eta}$ is therefore a notion of sameness of λ-terms for which the Principle of Extensionality holds.

There is room to wonder whether the Principle of Extensionality as stated in Definition 1.18 adequately captures the intuitive idea that functions which give the same outputs on the same inputs are the same function. For instance, it might be suggested that we should be willing to identify λ-terms U and V for which $U(c)$ and $V(c)$ are the same for every *closed* λ-term c. Or perhaps we should identify λ-terms U and V for which $U(c)$ and $V(c)$ are the same merely for every closed λ-term c in β-normal form. This would lead to various modifications of the Principle of Extensionality. Whether new reduction rules can be found that render the corresponding modified versions of Theorem 1.25 true is not clear. More generally, whether we should be content with Definition 1.18 and Theorem 1.25 as a statement of the 'extensionality' of our system is a question we shall have to leave open.

At this point, we have two independent notions of reduction available to us: β-reduction and η-reduction. We also have the notion of reduction that consists of their combination, namely, $\beta\eta$-reduction. All three of these notions of reduction satisfy the Church-Rosser Theorem, and so they have much in common. In most texts, β- and $\beta\eta$-reduction are the most closely studied. The notion of β-reduction is especially natural given the interpretation of λ-terms as functions which can operate on arguments, and the notion of $\beta\eta$-reduction is natural because of the way it ensures a type of extensionality. Both notions will be important in what follows. We will expand and explore these notions of reduction further in following chapters.

Exercises for Section 1.7

1. Reduce each of the following terms to $\beta\eta$-normal form, or argue that they cannot be reduced to $\beta\eta$-normal form. Here, x and y are distinct variables.

(i). $\lambda x(x(x))(\lambda x(y(x)))$
(ii). $\lambda x(y(x))(\lambda x(x(x)))$
(iii). $\lambda x(y(x))(\lambda x(y(x)))$
(iv). $\lambda x(x(x))(\lambda x(x(x)))$

2. Prove that if E_1 and E_2 are λ-terms such that $E_1 =_{\beta\eta} E_2$ and v is a variable, then $\lambda v(E_1) =_{\beta\eta} \lambda v(E_2)$. (Hint: use induction on n, where $X_0, X_1, ..., X_{n-1}, X_n$ is a sequence of λ-terms with $U = E_1$ and $V = E_2$ such that $X_0 \sim_{\beta\eta} X_1,\ X_1 \sim_{\beta\eta} X_2, ...,\ X_{n-1} \sim_{\beta\eta} X_n$.)

3. Prove that no λ-term has an infinite η-reduction sequence. That is to say, prove that there is no sequence of terms $X_0, X_1, X_2, ...$ such that $X_0 \to_\eta X_1 \to_\eta X_2 \to_\eta ....$

4. Prove Lemma 1.24. (Hint: suppose the term X contains an η-redex $\lambda x(E(x))$ which is η-reduced to E in the expression Y. We are supposing that Y then contains a β-redex $\lambda y(F)(G)$ which when β-reduced gives $Y \to_\beta Z$. From the Subterm Lemma (Lemma 1.5), in the term Y either the subterms E and $\lambda y(F)(G)$ are identical, one is properly contained in the other, or they are disjoint.

We thus have the following cases to consider:
Case (i): E and $\lambda y(F)(G)$ are identical subterms of Y. This means that X has the form

$$X = ... \ \lambda x(\lambda y(F)(G)(x)) \ ...$$

Case (ii): In the term Y, the subterm E is properly contained in the subterm $\lambda y(F)(G)$. This means that in the term Y, (a) E is the term $\lambda y(F)$, (b) E is contained in F, or (c) E is contained in G. In subcase (a), this means that X has the form

$$X = ... \ \lambda x(\lambda y(F)(x))(G) \ ...$$

In subcase (b) this means that Y has the form

$$Y = ...\lambda y(...E...)(G) \ ...$$

and so X has the form

$$X = ...\lambda y(...\lambda x(E(x))...)(G) \ ...$$

and in subcase (c), this means that X has the form

$$X = ...\lambda y(F)(...\lambda x(E(x))...) \ ...$$

Case (iii): In the term Y, the subterm $\lambda y(F)(G)$ is properly contained in the subterm E. This means that X has the form

$$X = ... \ \lambda x(...\lambda y(F)(G)...(x)) \ ...$$

Case (iv): E and $\lambda y(F)(G)$ are disjoint subterms of Y. Then it is easily argued that $\lambda x(E(x))$ must also be disjoint from $\lambda y(F)(G)$ in X, and so X has the form

$$X = ... \lambda x(E(x)) ... \lambda y(F)(G) ...$$

Consider all these possible forms for X and show that the result holds in each case.)

5. Using Lemma 1.24, prove the following generalization of Lemma 1.24

For all λ-terms X and Z, if $X \to_{\beta\eta} Z$, then there is a λ-term Y such that $X \twoheadrightarrow_{\beta} Y \twoheadrightarrow_{\eta} Z$.

6. Prove that if a term has an infinite $\beta\eta$-reduction sequence, then it also has an infinite β-reduction sequence. (Hint: use the result of the previous exercise.)

7. (i) Show that if $X \to_{\eta} Y$ and X is in β-normal form, then Y is in β-normal form. (ii) Give an example of $X \to_{\eta} Y$ in which Y is in β-normal form, but X is not in β-normal form.

8. In contrast with (ii) of the previous exercise, show that if $X \to_{\eta} Y$ and Y is in β-normal form, then X has a β-normal form.

9. Prove that if a term has a β-normal form, then it has a $\beta\eta$-normal form.

10. Prove that if a term has a $\beta\eta$-normal form, then it has a β-normal form.

11. Prove Lemma 1.23.

1.8 The untyped λ-calculus as a model of computation - I

By now you should have some confidence with the manipulation of λ-terms. Still, at this point all these manipulations might seem like pure formalism. λ-terms themselves are functions, but they are very strange sorts of functions that take *any* other function as a possible input, and so are quite different from the sorts of functions of everyday mathematical practice. What is then the mathematical point of this strange system?

For our purposes, one of the main points of the λ-calculus is that it is a very powerful *programming language.* In this section, we will show how certain traditional calculations or computations with which you are familiar can be interpreted as λ-calculus manipulations. In the next section, we will talk more abstractly about the λ-calculus as a programming language or framework for computation.

As our first example of the utility of the λ-calculus, let us focus on the manipulation of truth values. We begin by noting that both the terms $\lambda x(\lambda y(x))$ and $\lambda x(\lambda y(y))$ are in β-normal form and so cannot be simplified any further by β-reduction. Let us think of the former as denoting the truth value **T** and the later as denoting the truth value **F**:

$$\mathbf{T} \equiv \lambda x(\lambda y(x)), \quad \mathbf{F} \equiv \lambda x(\lambda y(y)). \tag{16}$$

Consider then the function $\lambda s(\lambda t(st\mathbf{F}))$, which we denote **AND**:

$$\mathbf{AND} \equiv \lambda s(\lambda t(st\mathbf{F})).$$

One can then verify the following:

$$\mathbf{AND}\ (\mathbf{T})(\mathbf{T}) \twoheadrightarrow_\beta \mathbf{T} \qquad \mathbf{AND}\ (\mathbf{T})(\mathbf{F}) \twoheadrightarrow_\beta \mathbf{F}$$
$$\mathbf{AND}\ (\mathbf{F})(\mathbf{T}) \twoheadrightarrow_\beta \mathbf{F} \qquad \mathbf{AND}\ (\mathbf{F})(\mathbf{F}) \twoheadrightarrow_\beta \mathbf{F}$$

For example,

$$\begin{aligned}\mathbf{AND}\ (\mathbf{T})(\mathbf{F}) &= \lambda s(\lambda t(st\mathbf{F}))(\mathbf{T})(\mathbf{F})\\ &\to_\beta \lambda t(\mathbf{T}t\mathbf{F})(\mathbf{F})\\ &\to_\beta \mathbf{TFF} \text{ (i.e., } \mathbf{T}(\mathbf{F})(\mathbf{F})\text{.)}\\ &= \lambda x(\lambda y(x))(\mathbf{F})(\mathbf{F})\\ &\to_\beta \lambda y\ (\mathbf{F})(\mathbf{F})\\ &\to_\beta \mathbf{F} \text{ (because } \mathbf{F} \text{ has no free variables, and so } \lambda y(\mathbf{F})(t) \to_\beta \mathbf{F} \text{ for any } t\text{.)}\end{aligned}$$

As an exercise in λ-calculus manipulations, you should verify the remaining three claims.

What this shows is that the λ-term **AND** 'computes' conjunction - that is to say, given two truth values v_1 and v_2, the term $\mathbf{AND}(v_1)(v_2)$ (more typically written $\mathbf{AND}\ v_1 v_2$) β-reduces to the truth value of the conjunction of v_1 with v_2. One can define similar λ-terms for other logical connectives - for example:

$$\mathbf{OR} \equiv \lambda s(\lambda t(s\mathbf{T}t)) \qquad \mathbf{NOT} \equiv \lambda s(s\mathbf{FT})$$

By combining these λ-terms in various ways we can of course construct more complicated connectives. What this means is that the manipulation of truth values can be interpreted as a type of manipulation (more specifically, β-reduction) of λ-terms.

The λ-calculus allows us to do far more than manipulate truth values. We can also encode the manipulation of natural numbers in the λ-calculus. Just as we began our discussion of the manipulation of truth values by encoding the truth values **T** and **F** with closed λ-terms, so too we will begin our discussion of the manipulation of natural numbers by encoding the natural numbers with closed λ-terms. Just as in the case of truth values, the coding is somewhat arbitrary, and there is no unique choice. The following coding of the natural numbers is most common:

$$\mathbf{0} \equiv \lambda x(\lambda y(y)),\ \mathbf{1} \equiv \lambda x(\lambda y(x(y))),\ \mathbf{2} \equiv \lambda x(\lambda y(x(x(y)))), ... \tag{17}$$

The terms in (17) are sometimes called the *Church numerals*.

In general, we will represent the natural number n as:

$$\mathbf{n} \equiv \lambda x(\lambda y(x^n y))$$

where $x^n y$ is $x(x(...(x(y))...))$ with n occurrences of x. (Note the placement of parentheses.) Using this notation, you should be able to see that $x^n(x^m y)$ is just $x^{n+m}y$.

With this convention, it is possible to define all sorts of complicated functions on the natural numbers. Consider, for instance, the λ-term $\lambda u\lambda x\lambda y(x((ux)y))$, which we denote **succ**. This acts on natural numbers in the following way:

$$\begin{aligned}\mathbf{succ}\ (\mathbf{n}) &\equiv \lambda u\lambda x\lambda y(x((ux)y))(\mathbf{n})\\ &\to_\beta \lambda x\lambda y(x((\mathbf{n}x)y))\\ &\to_\beta \lambda x\lambda y(x(x^n y))\ \ (\text{using}(\mathbf{n}x)y \to_\beta x^n y)\\ &\to_\beta \lambda x\lambda y(x^{n+1}y)\\ &\equiv \mathbf{n+1}\end{aligned}$$

Thus we have $\mathbf{succ\,n} \to_\beta \mathbf{n+1}$, and so we may think of **succ** as a λ-term that 'computes' the successor function.

For another example, define **add** to be the λ-term $\lambda u\lambda v\lambda x\lambda y[u(x)(v(x)(y))]$. Then we have:

$$\begin{aligned}\mathbf{add}(\mathbf{n})(\mathbf{m}) &\equiv \lambda u\lambda v\lambda x\lambda y[u(x)(v(x)(y))](\mathbf{n})(\mathbf{m})\\ &\to_\beta \lambda x\lambda y[\mathbf{n}(x)(\mathbf{m}(x)(y))]\\ &\to_\beta \lambda x\lambda y[\mathbf{n}(x)(x^m y))]\ \ (\text{using } \mathbf{m}(x)(y) \to_\beta x^m y)\\ &\to_\beta \lambda x\lambda y[\lambda y(x^n y)(x^m y)]\ \ (\text{using } \mathbf{n}(x) \to_\beta \lambda y(x^n y))\\ &\to_\beta \lambda x\lambda y[x^n(x^m y)]\\ &\to_\beta \lambda x\lambda y[x^{n+m} y]\\ &\equiv \mathbf{n+m}\end{aligned}$$

Thus, **add** represents the function of addition. It is a relatively mechanical matter to generalize these ideas to construct λ-terms that compute multiplication and exponentiation. You will explore this in the exercises.

In fact, more general results are possible, though they require significant effort to prove. Call a function f defined on some subset of the natural numbers *partial recursive* if it corresponds to the output of some Turing machine – that is to say, if there is some Turing machine such that for any n, if this Turing machine running on input n eventually terminates with a numerical output m, then f is defined on n and $f(n) = m$, while if this Turing machine running on input n does not eventually terminate with a numerical output (either because it does not terminate at all, or because it terminates with a non-numerical output on its tape), then f is undefined on n. (For a more general introduction to Turing machines and the theory of recursive functions, see chapter 3 and onwards of [3].)

For each partial recursive function f it can be shown that there is a closed λ-term Θ_f such that for all natural numbers n, $\Theta_f(\mathbf{n})$ may be reduced to a term in β-normal form iff $f(n)$ is defined. Furthermore, in the case in which $f(n)$ is defined,

$$\Theta_f(\mathbf{n}) \twoheadrightarrow_\beta \mathbf{f}(\mathbf{n}).$$

(Here, $\mathbf{f}(\mathbf{n})$ is the Church numeral corresponding to the natural number $f(n)$.) Thus, not only are there ways of representing the successor function, addition, multiplication and exponentiation

in the untyped λ-calculus λ_0, but also ways of representing any 'algorithmic' numeric function whatsoever. This result may also be straightforwardly generalized to partial recursive functions of more than one variable. For a fuller discussion and proof of this result, see chapter 4 of [4].

Exercises for Section 1.8

1. Verify that $\mathbf{NOT}\ (\mathbf{T}) \twoheadrightarrow_\beta \mathbf{F}$ and $\mathbf{NOT}\ (\mathbf{F}) \twoheadrightarrow_\beta \mathbf{T}$.

2. Construct a λ-term $\mathbf{ENT}$ satisfying the following axioms for entailment:

1. For any λ-term s, $\mathbf{ENT}(s)(\mathbf{T}) \twoheadrightarrow_\beta \mathbf{T}$ (i.e., everything entails $\mathbf{T}$),
2. For any λ-term s, $\mathbf{ENT}\ (\mathbf{F})(s) \twoheadrightarrow_\beta \mathbf{T}$ (i.e., everything is entailed by $\mathbf{F}$),
3. $\mathbf{ENT}\ (\mathbf{T})(\mathbf{F}) \twoheadrightarrow_\beta \mathbf{F}$.

You should verify that the term you construct has these three properties.

3. Define **mult** to be the λ-term $\lambda u \lambda v \lambda x(u(vx))$. Verify that $\mathbf{mult}(\mathbf{n})(\mathbf{m}) = \mathbf{p}$ where $n \times m = p$.

4. Construct a λ-term $\mathbf{exp}$ such that $\mathbf{exp}(\mathbf{n})(\mathbf{m}) = \mathbf{p}$ where $n^m = p$. Verify that the term you have constructed has this property.

1.9 The untyped λ-calculus as a model of computation - II

The discussion of the previous section shows that the λ-calculus λ_0 may be thought of as a model of *computation*, as powerful as any other model of computation. Just as the way in which in a Turing machine computation is implemented by a device moving along a tape, printing and deleting symbols according to definite rules, or the way in which in a modern computer computation is implemented by the manipulation of bits by logic gates, in the λ-calculus computation is modeled by β-reduction. In this way, the λ-calculus gives us an alternative way of thinking about the fundamental building blocks of computer programs. In particular, we can now think of all computation as ultimately reducible to the manipulation of terms in the λ-calculus, with the process of computation itself just becoming the process of β-reduction (or $\beta\eta$-reduction, if one wishes.) This gives a different and powerful way of thinking about the theoretical basis of computer programming.

There are texts devoted to so-called 'programming in the λ-calculus', that show how lists, more complex data structures, binary tree searches, recursion and so on can all be implemented in the λ-calculus. Indeed, the λ-calculus is perhaps the simplest example of a so-called *functional programming language*, a powerful and distinctive way of thinking about computation different from that given in the perhaps more familiar *imperative programming languages*. We will not discuss this in detail here. The interested reader should look at [6] or section 5.2 of [8] for fuller discussions.

There are several important differences between the λ-calculus viewed as a programming language and other more traditional programming languages. We can of course view any λ-term E as a program, either by (i) thinking of E itself as the starting point of a computation that terminates once we have reduced E to β-normal form, or (ii) by thinking of E as a (partial) function which on

input c – where c is yet another λ-term – produces the output corresponding to the β-normal form of $E(c)$. Much like other models of computation, in each case there is of course no guarantee that the program will halt, as not all terms have β-normal forms. But putting this to the side, regardless of whether we adopt point of view (i) or (ii), a curious difference with more familiar computer programs is that the outputs of these computer programs are not so much definite values, but rather λ-terms which are themselves further computer programs. This is different from more familiar computer programs where the output is generally some specific object, such as a number or string of characters. Still, as we have seen in the previous section, we can recover more traditional computer programs by identifying certain basic λ-terms with specific mathematical objects such as the truth values or the natural numbers, as in (16) or (17).

There are however further even deeper differences between the λ-calculus viewed as a programming language and more traditional programming languages. In a traditional computer program, a computer executes a sequence of instructions step by step. When one instruction is executed, there is no question what the next instruction to execute is - it is simply the next instruction on the list, or the next instruction as determined by some sort of rule in the presence of loops or other such programming structures. The process of β-reduction is however not like this. We have seen, after all, that some λ-terms can be β-reduced in different ways (though as the Church-Rosser Theorem shows, this does not mean that the program can produce different results.) So we can think of λ-terms as programs such that we sometimes have multiple choices as to the direction in which the program can proceed. Sometimes this is called an 'indeterminstic' program. When we call λ-terms 'programs', they should be understood as programs in this sense.

We saw earlier that there are λ-terms such that some ways of β-reducing them lead to infinite loops, and other ways of β-reducing them terminate in a term in β-normal form. The Church-Rosser Theorem guarantees that all chains of β-reduction that terminate yield the same result, but that of course does not mean that all chains of β-reductions terminate. One might worry that in this sense the β-reduction of λ-terms is quite *unlike* traditional computing, in that we must be clever and β-reduce in just the right way, lest our program not terminate at all. But it turns out that this worry can be averted by following certain *protocols* when faced with the decision of how to β-reduce. For example, one can always perform the β-reduction for the redex $\lambda x(E_1)(E_2)$ for which the λ is leftmost out of all redexes. It turns out that if one follows this protocol, it will terminate if there is *any* terminating sequence of β-reductions at all. (Demonstrating that this is so is non-trivial. See chapter 13 of [1], and in particular section 13.2 for a detailed discussion and proof.) There are other such protocols as well with this remarkable property. So long as one follows such a protocol (which can be done mechanically), the term in question reduces to a β-normal term if there is any such reduction at all, and one has something more like a 'deterministic' model of computing. A curiosity worth noting is that there are also protocols which are *unterminating* so long as there is *some* unterminating sequent of β-reductions!

1.10 Typed Computation.

We may view terms in the λ-calculus λ_0 as functions that act on each other, including themselves. As we have noted already, this is somewhat different from the ordinary functions mathematicians or computer programs typically manipulate. An ordinary mathematical function like $f(x) = 2x$ takes natural numbers or perhaps real numbers as inputs, but does not take itself as input. We do not ordinarily talk about doubling f itself, except as an indirect way to refer to something quite different, namely the function $f(f(x)) = 4x$. Likewise, ordinary computer programs might take as an input some natural number or string of characters, but a typical computer program does not take itself as an input.

Mathematical functions or computer programs typically act on data of a specific form and output data of a specific form. We are all familiar with functions that take natural numbers into natural numbers, or real numbers into real numbers, or vectors into real numbers, or matrices into ordered pairs of natural numbers, and so on. In each of these cases the input of the function must be of a specific form in order for the function to be applicable, and so long as the input has that specific form, the output will also have some fixed (though possibly different) form.

Given this, it is very natural to want to distinguish functions based on the structure or form of the data they input and output. A function that takes a natural number as an input and outputs a natural number is a very different sort of thing from a function that takes some sort of function as an input, and produces an ordered triple of natural numbers as an output. The untyped λ-calculus has no principled way of drawing these sorts of distinctions. We seek a type of formalism in which such distinctions may be drawn easily.

We begin then with the notion of a *type*. Loosely speaking, a type is some sort of domain on which a function acts, corresponding to data with some particular structure. The natural numbers $\mathbb{N}$ is a very simple example of a type. Sometimes people refer to the type of individuals, where an individual is something which is not a function. Another very useful type is the type of truth values, which consists of the two objects **T** and **F**. This is sometimes denoted $\mathbb{B}$. One can also build types from other types - so we can consider the type of ordered pairs, the first element of which is an element of $\mathbb{N}$ and the second element of which is an element of $\mathbb{B}$, or the type of functions from elements of $\mathbb{N}$ to elements of $\mathbb{B}$. This somewhat informal understanding of the notion of a type will suffice for now.

The functions mathematicians and computer scientists work with typically not only have some sort of type as their domain, but also as their codomain. So a function from $\mathbb{N}$ to $\mathbb{B}$ has the type $\mathbb{N}$ as its domain, and the type $\mathbb{B}$ as its codomain. Consider a predicate such as 'x is an even number'. This may be viewed as a function that takes an element of $\mathbb{N}$ as an input, and returns an element of $\mathbb{B}$ as an output. Thus, it may be viewed as a function from the type $\mathbb{N}$ to the type $\mathbb{B}$. (For this reason, open sentences are sometimes called propositional *functions*.)

Typical computations then are *typed* – they take input of a certain type into outputs of a certain type. In addition, in typical computations quantities that appear as intermediate steps in determining the output are also typed. For example, in the course of evaluating some function f that takes $\mathbb{N}$ to $\mathbb{N}$ on some particular input, we might end up with some variable which always takes on

a value from $\mathbb{B}$, and some other variable which always takes on a value from $\mathbb{N}$, both of which end up being manipulated in a certain way to calculate the eventual output. The entire computation is then a manipulation of variables, each of which is typed. We call the sort of computation in which not only the input and output but also any intermediate quantities have types *typed computation*.

The untyped λ-calculus gives us a useful model for computation in general. Is there any similar formalism that can be used to give a model of typed computation? It turns out that there is indeed a variant of the untyped λ-calculus that can be used to give some sort of model of typed computation; it is unsurprisingly called the *typed* λ-calculus. However, perhaps unlike the case of computation in general, what counts as a typed computation and what algorithms can be captured by typed computations very much depends on how much structure one is willing to include in the class of types. In this chapter we will present the simplest version of the typed λ-calculus, but a sequence of richer versions of the typed λ-calculus will be developed in the following chapters, each version expressively richer than the previous. One might ask how well the typed λ-calculus and its variants serve as a model of typed computation in general, or whether there even is a most general model of typed computation in the way that Turing machines or the untyped λ-calculus λ_0 give the most general possible models of general (untyped) computation. But these questions are hard and we will only be able to partially reflect on them much later.

1.11 Church-style Typed λ-calculus

We begin by presenting the simplest and earliest version of the typed λ-calculus, in the form roughly originally developed by Church.

To start, we assume that we have some unspecified, infinite set $\tau_1, \tau_2, ...$ of fundamental types, out of which we will build other types. Perhaps our fundamental types consist of $\mathbb{N}$, $\mathbb{B}$, and some other other naturally occurring types - the exact details will not matter for now, and in what follows we will simply assume that $\tau_1, \tau_2, ...$ is a list of unspecified fundamental types. We will also assume that more complex types can be built from simpler types in a certain way. To this end, consider the definition

Definition 1.26: Function Types.

If α and β are types, then $\alpha \to \beta$ (sometimes called the 'function type from α to β') is the type of unary functions that are defined on all objects of type α, and take objects of type α to objects of type β.

So for example, $\mathbb{N} \to \mathbb{N}$ is the type of functions that take natural numbers to natural numbers, and $\mathbb{N} \to \mathbb{B}$ is the type of functions that take natural numbers to truth values.

We will assume that whenever we have types α and β, we may form the type $\alpha \to \beta$. We will eventually add other mechanisms for building complex types out of simpler types, but the creation of function types will suffice for now. We then have the following definition

Definition 1.27: Simple types.

The *simple types* are those that may be obtained from the fundamental types $\tau_1, \tau_2, ...$ under the operation that takes two types α and β to the type $\alpha \to \beta$.

For example, τ_1, $(\tau_1 \to \tau_3)$ and $(\tau_1 \to \tau_5) \to (\tau_1 \to \tau_3)$ are all simple types.

The nature of *variables* in the typed λ-calculus requires some discussion. In the untyped λ-calculus, each variable $x, y, z, ...$ could take *any* value. In the *typed λ-calculus*, by contrast, each variable will be associated with a *type*. So, for example, the variable x might represent an arbitrary element of $\mathbb{N}$, the variable y might represent an arbitrary element of $\mathbb{B}$, and so on. The easy way to implement this is to take some standard infinite list of variables $x, y, z, ...$ and stipulate that for each type τ, we have the variables $x^\tau, y^\tau, z^\tau, ..$, where these variables are understood as ranging over objects of type τ. So the variables $x^{\mathbb{N}}$, $x^{\mathbb{B}}$, and $x^{\mathbb{N}\to\mathbb{B}}$ are all quite different things - the first is a variable that ranges over elements of $\mathbb{N}$, the second is a variable that ranges over elements of $\mathbb{B}$, and the third is a variable that ranges over elements of $\mathbb{N} \to \mathbb{B}$. The fact that they all involve x does not connect them in any way – they should be thought of as three completely distinct variables that have nothing more in common than $u^{\mathbb{N}}$, $v^{\mathbb{B}}$ and $w^{\mathbb{N}\to\mathbb{B}}$ do. In the typed λ-calculus, all variables will be typed in this way.

Just as every variable in the typed λ-calculus is typed, so too in fact every expression in the typed λ-calculus will be typed. The type of an expression will be denoted by a superscript attached to the expression itself. For example, $[E]^\tau$ denotes the expression E, along with the stipulation that it has type τ. Sometimes for ease of reading we will omit the brackets and simply write E^τ. In full rigor we then define the set of terms of the typed λ-calculus as follows

Definition 1.28: Simple typed λ-terms.

The set of *simply typed λ-terms* are defined inductively as follows:

(i) For any variable v and any simple type τ, v^τ is a simply typed λ-term of type τ. (To reduce clutter, we will generally not write this as $[v]^\tau$).

(ii) For simple types α and β, if $[M]^{\alpha\to\beta}$ and $[N]^\alpha$ are simply typed λ-terms, then $[[M]^{\alpha\to\beta}([N]^\alpha)]^\beta$ is a simply typed λ-term.

(iii) For simple types α and β, if $[M]^\beta$ is a simply typed λ-term and x^α a variable, then $[\lambda x^\alpha([M]^\beta)]^{\alpha\to\beta}$ is a simply typed λ-term.

Terms of the typed λ-calculus then look very much like terms of the untyped λ-calculus, with superscripts added to each subexpression. Examples of terms in the typed λ-calculus include:

$$\begin{gathered}[x^{\alpha\to\beta}(y^\alpha)]^\beta \\ [[\lambda x^\alpha(x^\beta)]^{\alpha\to\beta}(y^\alpha)]^\beta \\ z^{(\alpha\to\beta)\to\gamma}([\lambda x^\alpha(x^\beta)]^{\alpha\to\beta})\end{gathered} \tag{18}$$

The clauses of Definition 1.28 might look complex, but actually express fairly simple ideas. Clause (i) simply gives us an inexhaustible list of variables of each type. Clause (ii) corresponds to the idea of *application*, this time in the context of the typed λ-calculus. In the untyped lambda calculus, we could apply any function to any input - that is to say, $M(N)$ was always a valid λ-term for any λ-terms M and N. That however is no longer the case. To apply M to N, M must be a function and N must be an object of the right type to act as an input to M. More specifically, for some types α and β, M must be a term of type $\alpha \to \beta$ and N must be a term of type α. In such a case, the result of applying M to N will have type β. Such a requirement is the idea behind clause (ii). So for example, if α and β are distinct simple types, the term $x^{\alpha\to\beta}(y^\beta)$ is *not* a legal simply typed λ-term, as it involves trying to apply a function from α to β to an input from β. The term $[x^{\alpha\to\beta}(y^\alpha)]^\beta$ is by contrast acceptable and has overall type β because it is the result of applying a function of type $\alpha \to \beta$ to an input of type α, which of course yields an output of type β.

Consider now clause (iii). In the untyped context, the term $\lambda x(M)$ corresponded to treating the expression M as a function of some variable x. This is a function which on input c yields $M[x/c]$. Suppose now that x has some type α and M has some type β. Then we can think of $M[x/c]$ as a function that takes an input of type α and produces an output of type β. Thus, in the typed context the λ-term $\lambda x(M)$ – written more fully as $\lambda x^\alpha([M]^\beta)$ – has the simple type $\alpha \to \beta$. That is the idea behind clause (iii).

We will often omit not only the brackets $[\cdot]$, but also type symbols when they are obvious or can be easily inferred from the context. So for instance, rather than writing $[[M]^{\alpha\to\beta}([N]^\alpha)]^\beta$, we will typically just write $M^{\alpha\to\beta}(N^\alpha)$ (or even just $M^{\alpha\to\beta}N^\alpha$), as the fact that this term has type β follows trivially from the fact that it involves applying a function of type $\alpha \to \beta$ to an object of type α. Likewise, rather than writing $[\lambda x^\alpha(M^\beta)]^{\alpha\to\beta}$, we will typically write $\lambda x^\alpha(M^\beta)$ (or even just $\lambda x^\alpha M^\beta$.) The terms (18) can therefore be written more simply:

$$x^{\alpha\to\beta}(y^\alpha)$$
$$\lambda x^\alpha(x^\beta)(y^\alpha)$$
$$z^{(\alpha\to\beta)\to\gamma}(\lambda x^\alpha(x^\beta))$$

In general, when a λ-term is written out in full it suffices to indicate only the type of each variable, as the type of any larger expression can then be easily deduced, as seen in our three examples.

In addition, for each bound variable it typically suffices to indicate the type of the variable only when it is introduced with the corresponding λ-operator. For example, the term $\lambda x^\alpha \lambda y^\beta(x^\alpha y^\beta)$ can be rewritten $\lambda x^\alpha \lambda y^\beta(xy)$, as the types of x and y have already been stipulated to be α and β respectively. There are, however, situations where this convention can lead to confusion. For example, if we were to start with the typed λ-term $\lambda x^\alpha \lambda x^\beta(x^\alpha x^\beta)$ and use the convention in question, we would end up with $\lambda x^\alpha \lambda x^\beta(xx)$, which is potentially confusing (what are the types of the unadorned xs?) This sort of situation can be avoided by renaming variables and working with α-equivalent terms. So $\lambda x^\alpha \lambda x^\beta(x^\alpha x^\beta)$ is α-equivalent to $\lambda x^\alpha \lambda y^\beta(x^\alpha y^\beta)$, which may be abbreviated $\lambda x^\alpha \lambda y^\beta(xy)$, and all ambiguity has been removed. In general, we will not omit information when it could potentially lead to ambiguity.

With all these conventions in mind, the following may be regarded as fully typed terms:

$$\lambda x^{\alpha\to\beta}(x(z^{\alpha}))$$
$$\lambda x^{\alpha}\lambda y^{\alpha\to\beta}(yx)$$
$$\lambda x^{\alpha\to\beta}\lambda z^{(\alpha\to\beta)\to\gamma}(z(\lambda y^{\alpha}(xy)))$$

In particular, given the type of each free and bound variable in a λ-term, the type of every subexpression (including the type of the overall term itself) can be determined.

In fact, sometimes even less information is necessary to determine the type of a λ-term and all its subexpressions. For instance, consider the term

$$\lambda x^{\alpha\to\beta}(xy).$$

Here the bound variable x is typed, but the open variable y has not been explicitly typed. However, from the fact that there is a subexpression of the form xy and x has type $\alpha \to \beta$, it follows that the type of y must be α. Thus the expression $\lambda x^{\alpha\to\beta}(xy)$ can be regarded as fully typed, as it contains enough information for us to deduce the type of every subexpression. Nevertheless, in order to make things as simple as possible for the reader, we will generally explicitly indicate the types of all bound and free variables in a typed term, writing for example

$$\lambda x^{\alpha\to\beta}(xy^{\alpha}).$$

As in the untyped λ-calculus, the λ-term $\lambda x^{\sigma}(M^{\tau})(N^{\sigma})$ simplifies to

$$M^{\tau}[N^{\sigma}/x^{\sigma}].$$

As before, we will call the act of simplifying $\lambda x^{\sigma}(M^{\tau})(N^{\sigma})$ to $M^{\tau}[N^{\sigma}/x^{\sigma}]$ *β-reduction*. We use the same notational conventions as before:

Definition 1.29: β-reduction.

An expression of the form $\lambda x^{\sigma}(M^{\tau})(N^{\sigma})$ (where x^{σ} is a variable and M^{τ}, N^{σ} are λ-terms) is called a *β-redex* (or just a *redex*). We have
(i) $A \to_{\beta} B$ iff A has the form $\tau_1\ \lambda x^{\sigma}(M^{\tau})(N^{\sigma})\ \tau_2$ and B has the form $\tau_1\ M^{\tau}[N^{\sigma}/x^{\sigma}]\ \tau_2$.
(ii) $A \twoheadrightarrow_{\beta} B$ (A β-reduces to B) iff for some sequence $X_0, X_1, ..., X_n$ of λ-terms we have that $A = X_0$, $B = X_n$, and $X_0 \to_{\beta} X_1 ... \to_{\beta} X_n$. This includes the degenerate case in which A and B are identical.
(iii) A term A is in *β-normal form* just in case no β-reduction may be performed on it.

Analogous terminology is used for η-reduction and $\beta\eta$-reduction.

We use the term λ_0^{type} to refer to this version of the λ-calculus, with the set of rules of formation of typed λ-terms and rules of reduction just presented. Sometimes it is also called a *Church style typed λ-calculus*. We present its rules together for convenience:

Definition 1.30: Rules of the typed λ-calculus λ_0^{type}.

Types

The *simple types* are those that may be obtained from the fundamental types $\tau_1, \tau_2, ...$ under the operation that takes two types α and β to the type $\alpha \to \beta$.

Terms

The set of *simply typed λ terms* are defined inductively as follows:

(i) For any variable v and any simple type τ, v^τ (or $[v]^\tau$) is a simply typed λ-term.
(ii) For simple types α and β, if $[M]^{\alpha\to\beta}$ and $[N]^\alpha$ are simply typed λ-terms, then $M^{\alpha\to\beta}(N^\alpha)$ (or more fully, $[[M]^{\alpha\to\beta}([N]^\alpha)]^\beta$) is a simply typed λ-term.
(iii) For simple types α and β, if $[M]^\beta$ is a simply typed λ-term and x^α a variable, then $\lambda x^\alpha(M^\beta)$ (or more fully, $[\lambda x^\alpha([M]^\beta)]^{\alpha\to\beta}$) is a simply typed λ-term.

Reduction Rules

(i) $\lambda x^\sigma([M]^\tau)([N]^\sigma) \to_\beta [M]^\tau[[N]^\sigma/x^\sigma]$
(ii) $\lambda x^\sigma([E]^\tau(x^\sigma)) \to_\eta [E]^\tau$, where x^σ does not appear free in $[E]^\tau$.

The typed λ-calculus is quite different from the untyped λ-calculus in several important ways. Unlike the untyped λ-calculus λ_0, it may be shown that every term of the typed λ-calculus λ_0^{type} can be β-reduced to a term in β-normal form (and moreover can be $\beta\eta$-reduced to a term in $\beta\eta$-normal form.) Recall the problematic *untyped* λ-term:

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

This *untyped* term has no β-normal form - when we β-reduce it, we simply get the original term back. However, there is no term like this in the *typed* λ-calculus. The sub-term $x(x)$ of our problematic untyped λ-term has nothing corresponding to it in the typed λ-calculus. In particular, for any type α, $x^\alpha(x^\alpha)$ is not a legal term, as it does not conform to the pattern in clause (ii) of the definition of λ-terms in Definition 1.30. In order for $x^\beta(x^\alpha)$ to be a legal term in the typed λ-calculus, β must have the form $\alpha \to \gamma$ for some γ. In order for $x^\alpha(x^\alpha)$ to be a legal term, α itself would then have to have the form $\alpha \to \gamma$, and no simple type (or any type we consider in this book) has this sort of self-referential character.

In the exercises, you will be walked through a proof that each term in the typed λ-calculus λ_0^{type} has a β-normal form. Actually, it turns out that something stronger is true – for each term in λ_0^{type} *every* series of β-reductions eventually terminates with a term in β-normal form. (This property is sometimes called *strong normalization*.) That is to say, not only is it the case that for every typed λ-term of λ_0^{type} there is *some* sequence of β-reductions taking it to a term in β-normal form, but in fact *every* sequence of β-reductions eventually leads to a term in β-normal form. That is, we have the following:

Theorem 1.31: Strong Normalizability of λ_0^{type} under β-reduction.

There is no infinite sequence $\tau_1, \tau_2, \ldots$ of terms in the typed λ-calculus λ_0^{type} for which

$$\tau_1 \to_\beta \tau_2 \to_\beta \tau_3 \to_\beta \ldots$$

The proof of this theorem is extremely delicate and is included in the appendix to this chapter. As before, we also have a version of the Church-Rosser theorem for the typed λ-calculus:

Theorem 1.32: The Church-Rosser Theorem for β-reductions in λ_0^{type}.

In the typed λ-calculus λ_0^{type}, if $X \twoheadrightarrow_\beta Y_1$ and $X \twoheadrightarrow_\beta Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$.

We also discuss this in the appendix. Analogs of these theorems also hold for $\beta\eta$-reductions in λ_0^{type}. In particular, we have the following

Theorem 1.33: Strong Normalizability of λ_0^{type} under $\beta\eta$-reduction.

There is no infinite sequence $\tau_1, \tau_2, \ldots$ of terms in the typed λ-calculus λ_0^{type} for which

$$\tau_1 \to_{\beta\eta} \tau_2 \to_{\beta\eta} \tau_3 \to_{\beta\eta} \ldots$$

(where $X \to_{\beta\eta} Y$ iff $X \to_\beta Y$ or $X \to_\eta Y$.)

Theorem 1.34: The Church-Rosser Theorem for $\beta\eta$-reductions in λ_0^{type}.

In the typed λ-calculus λ_0^{type}, if $X \twoheadrightarrow_{\beta\eta} Y_1$ and $X \twoheadrightarrow_{\beta\eta} Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_{\beta\eta} Z$ and $Y_2 \twoheadrightarrow_{\beta\eta} Z$.

Theorem 1.33 actually follows easily from Theorem 1.31, using the fact proved in an earlier exercise that if a term has an infinite $\beta\eta$-reduction sequence, then it also has an infinite β-reduction sequence. (Although the argument given for this was in the context of the untyped λ-calculus, the addition of types does not affect the argument.) The proof of Theorem 1.34 requires more attention, and is left to the appendix.

As before, it follows easily from the Church-Rosser Theorem(s) for λ_0^{type} that starting with the same λ_0^{type} term, different sequences of β-reductions (or $\beta\eta$-reductions) that terminate with

a term in β-normal form (or $\beta\eta$-normal form) must in fact terminate with the same term (up to α-equivalence.)

Putting all these results together we have that starting with a fixed term of the typed λ-calculus λ_0^{type}, every sequence of β-reductions (or $\beta\eta$-reductions) eventually terminates, and terminates with exactly the same β-normal form (or $\beta\eta$-normal form) term.

The fact that every sequence of β-reductions eventually terminates makes the typed λ-calculus λ_0^{type} a very convenient programming language, as we do not have to worry about programs falling into infinite loops, for example. But this comes at a cost. We cannot hope to simulate all Turing Machines in the typed λ-calculus, as not all Turing Machines halt. If one thinks that there is some intuitive sense in which the running of a Turing machine program is an example of typed computation, then one will have to concede that there are examples of typed computation that are *not* captured by the typed λ-calculus λ_0^{type} (or even any of the extensions we will consider in the next chapters.) Perhaps then the typed λ-calculus λ_0^{type} should be taken a model of something more like *convergent* (i.e., terminating) typed calculation. While there is something right about this, the typed λ-calculus does not seem to be a 'universal' model of even convergent typed calculation in the way the untyped λ-calculus is something like a universal model of general computation, as by adding further structure to the types of the untyped λ-calculus (as we will in the following chapters) we get increasing expressive power. This is unlike the case of the untyped λ-calculus in which we can already represent all computable functions, and so adding further computational machinery does not end up increasing expressive power. Indeed, it is not clear at present whether there is any universal model of typed computation or even typed convergent computation analogous to the universal model of general computation given by Church's Thesis.

The important point for now is just that the typed λ-calculus λ_0^{type}, in contrast with the untyped λ-calculus λ_0, is a somewhat limited programming language. The Curry-Howard Correspondence will reveal even further limitations. Still, the typed λ-calculus λ_0^{type} is not *hopelessly* limited, as there are many aspects of data manipulation that can be captured by it. Moreover, in further chapters we will add various resources to the typed λ-calculus λ_0^{type} to significantly expand its expressive power, while at the same time keeping everything fully 'typed'. So both in its own right and as a starting point for further expansion, the typed λ-calculus λ_0^{type} is both a programming language and mathematical object of great interest.

Exercises for Section 1.11

1. Say whether the following terms are grammatical or ungrammatical terms of the Church style typed λ-calculus. For each grammatical term, if it is not already in β-normal form you should (i) reduce it to β-normal form, and (ii) state what the type of the β-normal form term is. You should assume that A and B are distinct types.

(i). $x^A(y^B)$
(ii). $\lambda x^A(y^B)(z^B)$

(iii). $\lambda x^A(y^B)(z^A)$
(iv). $\lambda x^A(x^A)(\lambda y^B(y^B))$
(v). $\lambda x^A(x^A)(\lambda y^A(y^A))$
(vi). $\lambda x^{A\to A}(x^{A\to A})(\lambda y^A(y^A))$
(vii). $\lambda x^{A\to A}(x^{A\to A})(\lambda y^A(y^A))(z^A)$

2. (a) Prove that if the type of each free and bound variable in a λ-term M is given, that the type of every subexpression of M can be determined. (Hint: proceed by induction on the construction of λ-terms.)
(b) Is it true that if the type of a λ-term M as well as the type of each free variable is given, that the type of every subexpression of M can be determined? If so, prove this fact. If not, give a counterexample.
(c) Is it true that if the type of a λ-term M as well as the type of each bound variable is given, that the type of every subexpression of M can be determined? If so, prove this fact. If not, give a counterexample.

3. We prove in this exercise that there is a strategy for β-reducing terms in the typed λ-calculus λ_0^{type} that always terminates. Thus, every term in the typed λ-calculus has a normal form. (This is a weaker result than Theorem 1.31, but is easier to prove.)

First of all, we define the *complexity* $c(\tau)$ of a simple type inductively by stipulating that $c(\tau)$ is 0 if τ is one of the fundamental types, and $c(\tau)$ is $1 + \max\{c(\sigma_1), c(\sigma_2)\}$ if τ is a type of the form $\sigma_1 \to \sigma_2$.

We next associate with each λ-term s a pair of natural numbers $n(s)$ and $m(s)$, where $n(s)$ is the maximum value of $c(\tau)$ for any variable x^τ that appears bound in s, and $m(s)$ is the number of occurrences in s of sub-expressions of the form $\lambda x^\tau(...)$, where τ is a type with $c(\tau) = n(s)$. So loosely speaking, $n(s)$ is the highest complexity of the type of any bound variable in s, and $m(s)$ is the number of abstractions in s over a variable whose type has that maximum complexity.

Argue that if s is a typed λ-term which is not in β-normal form, then there is a term s' that may be obtained from s by a single β-reduction (i.e., $s \to_\beta s'$) such that either (i) $n(s') < n(s)$, or (ii) $n(s') = n(s)$ and $m(s') < m(s)$. Argue from this that for any typed λ-term s, there is *some* sequence of β-reductions beginning with s that terminates. You should in fact be able to describe an algorithm for the β-reduction of typed λ-calculus terms that always terminates.

4. Can the strategy of the previous problem be adapted to give a strategy for $\beta\eta$-reducing terms in the typed λ-calculus λ_0^{type} that always terminates?

1.12 Curry-style Typed λ-calculus

Somewhat confusingly, there are several distinct formalisms that go under the name of the typed λ-calculus. First, there is the *Church-style* typed λ-calculus λ_0^{type} that we have already seen. Second, there is the so-called *Curry-style* typed λ-calculus, of which several distinct versions exist. We

describe one version of the Curry-style typed λ-calculus this section and another in the next.

In the Church-style λ-calculus λ_0^{type}, each λ-term has a single, unique type. However, in the first version of the Curry style λ-calculus that we consider, the terms are ordinary terms of the *untyped* λ-calculus λ_0, each of which is assigned a *set* of possible types. To see why we might want to do this, consider the untyped λ-term $\lambda x(x)$. This of course is the identity function. It is built up from the variable x. If we assign x the type $\mathbb{N}$, then the untyped λ-term $\lambda x(x)$ is transformed into the typed λ-term $\lambda x^{\mathbb{N}}(x^{\mathbb{N}})$ (or $\lambda x^{\mathbb{N}}(x)$, using our abbreviations). This of course is just the identity function that takes an element of $\mathbb{N}$ to itself, and has type $\mathbb{N} \to \mathbb{N}$. It is a term of the Church-style typed λ-calculus λ_0^{type}. Likewse, if we think of x as having the type $\mathbb{B}$, then the untyped λ-term $\lambda x(x)$ is transformed into the typed λ-term $\lambda x^{\mathbb{B}}(x^{\mathbb{B}})$ (or $\lambda x^{\mathbb{B}}(x)$), which is the identity function that takes an element of $\mathbb{B}$ to itself, and has type $\mathbb{B} \to \mathbb{B}$. In general, for any type τ, by thinking of x as having type τ our original term $\lambda x(x)$ can be 'typed' into the term $\lambda x^{\tau}(x^{\tau})$ (or $\lambda x^{\tau}(x)$) of the Church-style typed λ-calculus λ_0^{type}, where it acquires the type $\tau \to \tau$.

Because it is possible to add superscripts to the term $\lambda x(x)$ in different ways to produce different terms of the Church-style typed λ-calculus λ_0^{type}, we can think of the untyped term $\lambda x(x)$ as having many possible types. The set of types it is possible for the term $\lambda x(x)$ to acquire in this way is of course just the set of types of the form $X \to X$.

Just as we can examine the set of possible types a closed λ-term like $\lambda x(x)$ can be given, so too we can examine the set of possible types an open λ-term can be given. Consider for example the open untyped λ-term $\lambda x(y)$. By assigning x the type σ and y the type τ, the term $\lambda x(y)$ is transformed into the term $\lambda x^{\sigma}(y^{\tau})$, a term of λ_0^{type}, which of course has the type $\sigma \to \tau$. In this way, the set of types it is possible for the term $\lambda x(y)$ to acquire is the set of types of the form $X \to Y$.

In determining the possible types of an expression, we are constrained by the fact that when a typed expression M^{σ} is applied to a typed expression N^{τ}, the resulting expression $M^{\sigma}(N^{\tau})$ makes grammatical sense only if σ is of the form $\tau \to \gamma$. As a result of this constraint, the set of possible types a given untyped term can assume can be empty. For example, if x is a variable, consider the open term $x(x)$. If we assign x the type σ, it has the form $x^{\sigma}(x^{\sigma})$. But this is ungrammatical; if the second x has type σ, then the first x needs to have type $\sigma \to \tau$ for some τ in order for the whole expression to make sense. So it is impossible to assign $x(x)$ any type at all in a consistent manner, and thus its set of possible types is empty. Any untyped term in which $x(x)$ occurs (such as the closed term $\lambda x(x(x))$) will also have an empty set of possible types for the same reason.

Let us consider another example. Take the λ-term $\lambda x(\lambda y(x))$. What types can this be assigned? Suppose x has type τ, and y has type σ. Then $\lambda y(x)$ is a function from objects of type σ to objects of type τ, and so is itself a term of type $\sigma \to \tau$. The entire expression $\lambda x(\lambda y(x))$ is thus a function which takes an object of type τ into an object of type $\sigma \to \tau$, and so has type $\tau \to (\sigma \to \tau)$. We have thus shown that the untyped term $\lambda x(\lambda y(x))$ can be 'typed' into the following term of λ_0^{type}

$$[\lambda x^{\tau}(\lambda y^{\sigma}(x^{\tau}))^{\sigma \to \tau}]^{\tau \to (\sigma \to \tau)}$$

or more simply,

$$\lambda x^{\tau}(\lambda y^{\sigma}(x)).$$

Although this might seem pedantic, we would like to formalize this sort of reasoning about which types an untyped λ-term can be assigned. Central to this sort of reasoning will be claims of the following form, for an untyped λ-term E

> Consistent with the constraint that the variables $v_1, v_2, ..., v_n$ (which include all the free variables of E) be assigned types $\tau_1, \tau_2, ..., \tau_n$ respectively, it is possible to type E so that it becomes a λ_0^{type} term of type σ.

We express this using the following notation

Definition 1.35: Type Theoretic Sequents.

For an untyped λ-term E and list of variables $v_1, v_2, ..., v_n$ that include (but is not necessarily limited to) all the free variables of E, we write

$$v_1 : \tau_1,\ v_2 : \tau_2,\ ...,\ v_n : \tau_n \vdash E : \sigma$$

to mean that consistent with the constraint that the free variables $v_1, v_2, ..., v_n$ of E are assigned types $\tau_1, \tau_2, ..., \tau_n$ respectively, it is possible to type E so that it becomes a λ_0^{type} term of type σ.

The claim

$$v_1 : \tau_1,\ v_2 : \tau_2,\ ...,\ v_n : \tau_n \vdash E : \sigma$$

is called a *type theoretic sequent* (or just a *sequent*), and its left hand side

$$v_1 : \tau_1,\ v_2 : \tau_2,\ ...,\ v_n : \tau_n$$

is its *context* or *environment*.

A sequent then just says that in a given context, a term may be assigned a given type. The sequent $\vdash E : \sigma$ with empty context then just says that E is a closed term that can be assigned the type σ – i.e., E has σ among its possible types.

Using this notation, the examples we have give so for in this section may be rewritten

$$\begin{aligned} &\vdash \lambda x(x) : \tau \to \tau \\ y : \tau &\vdash \lambda x(y) : \sigma \to \tau \\ y : \sigma &\vdash \lambda x(\lambda y(x)) : \tau \to (\sigma \to \tau) \end{aligned}$$

for any types σ and τ.

Within a context $v_1 : \tau_1,\ v_2 : \tau_2,\ ...,\ v_n : \tau_n$ we allow a variable to appear more than once. If it appears more than once, however, it must be assigned the same type each time. So for example,

$$x : \tau,\ x : \tau,\ y : \sigma$$

is a grammatical context, but

$$x : \tau,\ x : \sigma,\ y : \sigma$$

is not a grammatical context if σ and τ are distinct.

The context can be thought of as consisting of a sequence of 'variable declarations'. For example, we can think of the context:

$$x : \mathbb{N}, y : \mathbb{N} \to \mathbb{N}$$

as expressing:

> x is a natural number, and y is a function from natural numbers to natural numbers.

or, from a computer programmer's point of view

> let x be a natural number, and let y be a function from natural numbers to natural numbers.

With all this understood, we can then easily write down some 'laws' that sequents of the form $v_1 : \tau_1,\ v_2 : \tau_2,\ ...,\ v_n : \tau_n \vdash E : \sigma$ must obey. In particular, we have:

Definition 1.36: Rules of the typed λ-calculus $\mathbf{TR}_0$.

(i) $\Gamma, v : \tau \ \vdash\ v : \tau$ (Var)

(ii) $$\frac{\Gamma \ \vdash\ E_1 : \tau \to \sigma \qquad \Gamma \ \vdash\ E_2 : \tau}{\Gamma \ \vdash\ E_1(E_2) : \sigma}$$ (App)

(iii) $$\frac{\Gamma, v : \tau \ \vdash\ E : \sigma}{\Gamma \ \vdash\ \lambda v(E) : \tau \to \sigma}$$ (Abs)

where v is a variable, E_1, E_2 and E are terms of the untyped λ-calculus, σ, τ are types, and all contexts are assumed to be grammatical.

'Var' is here an abbreviation for *variable rule*, 'App' for *application rule*, and 'Abs' for *abstraction rule*. Sometimes these rules are called *typing rules*, and with this in mind we call the system of three rules just given $\mathbf{TR}_0$. (Other texts sometimes use the name $\lambda_\to$ for this set of rules.) Our first version of the Curry-style typed λ-calculus is constituted by this set of typing rules.

The (Var) rule states the obvious fact that in some context in which the variable v has been assigned type τ, τ is a possible type of v. The (App) rule states that if E_1 has possible type $\tau \to \sigma$ in some context Γ and E_2 has possible type τ *in the same context* Γ, then in that context $E_1(E_2)$ has possible type σ. The (Abs) rule states that if in some context $\Gamma, v : \tau$ the term E has the possible type σ, then in the context Γ, the term $\lambda v(E)$ has possible type $\tau \to \sigma$. These three rules may thus be read as statements about the structure of the set of possible types that each term can be assigned.

We prove that an untyped λ-term can be assigned a certain type by constructing a 'proof' of this fact using the above rules. More specifically, to show that E can be assigned the type X it suffices to construct a proof of

$$\Gamma \vdash E : X$$

in $\mathbf{TR_0}$ for some grammatical context Γ.

To take our most basic example, to show that the λ-term $\lambda x(x)$ can be assigned the type $\tau \to \tau$ for any type τ, we have the following argument:

$$\dfrac{x : \tau \ \vdash \ x : \tau \qquad \text{(Var)}}{\vdash \ \ \lambda x(x) : \tau \to \tau}\ \text{(Abs)}$$

Note that while this argument shows that every type of the form $\tau \to \tau$ is a possible type of $\lambda x(x)$, it does not show that every possible type of $\lambda x(x)$ has the form $\tau \to \tau$. While true, this latter fact requires a different sort of argument that we will not consider here.

For another example, to show that the expression $\lambda x(\lambda y(x))$ can be assigned the type $\tau \to (\sigma \to \tau)$ for any types τ and σ, we have the following argument:

$$\dfrac{\dfrac{x : \tau,\ y : \sigma \ \vdash \ x : \tau \qquad \text{(Var)}}{x : \tau \ \vdash \ \lambda y(x) : \sigma \to \tau}\ \text{(Abs)}}{\vdash \ \ \lambda x(\lambda y(x)) : \tau \to (\sigma \to \tau)}\ \text{(Abs)}$$

This argument shows that every type of the form $\tau \to (\sigma \to \tau)$ is a possible type of $\lambda x(\lambda y(x))$.

For a more complex example, to show that the expression $\lambda x(\lambda y(x(xy)))$ can be assigned any type of the form $(\sigma \to \sigma) \to (\sigma \to \sigma)$, we have the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{x : \sigma \to \sigma,\ y : \sigma \ \vdash \ x : \sigma \to \sigma \qquad \dfrac{x : \sigma \to \sigma,\ y : \sigma \ \vdash \ x : \sigma \to \sigma \qquad x : \sigma \to \sigma,\ y : \sigma \ \vdash \ y : \sigma}{x : \sigma \to \sigma,\ y : \sigma \ \vdash \ xy : \sigma}\ \text{(App)}}{x : \sigma \to \sigma,\ y : \sigma \ \vdash \ x(xy) : \sigma}\ \text{(App)}}{x : \sigma \to \sigma \ \vdash \ \lambda y(x(xy)) : \sigma \to \sigma}\ \text{(Abs)}}{\vdash \ \ \lambda x(\lambda y(x(xy))) : (\sigma \to \sigma) \to (\sigma \to \sigma)}\ \text{(Abs)}$$

Each 'leaf' of this tree is of the form (Var). We call arguments like these *typing derivations*.

In typing derivations, we allow ourselves to 'silently' replace untyped λ-terms by α-equivalent terms, without requiring any explicit step in which the relevant bound variables are renamed. So for example, consider the untyped λ-term

$$\lambda x(\lambda x(x)).$$

As discussed earlier, we regard the innermost (x) as bound by the second λx from the left rather than the first λx from the left. (In general, when there are multiple options for binding a variable, the variable is bound by the closest possible binding operator that is grammatically capable of binding it.) Given this, we should expect that for any types σ and τ this λ-term can be assigned the type $\sigma \to (\tau \to \tau)$. Proving this is easy:

$$\dfrac{\dfrac{x:\sigma,\ y:\tau\ \vdash\ y:\tau \qquad \text{(Var)}}{x:\sigma\ \vdash\ \lambda y(y):\tau\to\tau}\ \text{(Abs)}}{\vdash\ \ \lambda x(\lambda x(x)):\sigma\to(\tau\to\tau)}\ \text{(Abs)}$$

where in the last step the untyped λ-term $\lambda x(\lambda y(y))$ has been silently replaced by the α-equivalent term $\lambda x(\lambda x(x))$. You should be able to convince yourself that without silent renaming of this sort, no proof of the sequent $\vdash \lambda x(\lambda x(x)) : \sigma \to (\tau \to \tau)$ would be possible.

To avoid having to rename variables in the middle of a proof, it can be useful to start with λ-terms in which all bound variables have different names, and no variable is both free and bound. Sometimes this is called the *Barendregt variable convention*.

Definition 1.37: Barendregt Variable Convention.

Say that a λ-term E obeys the *Barendregt variable convention* iff all bound variables of E have different names, and no variable of E occurs both free and bound in E.

Note that sometimes this convention is defined to merely require that no variable of E occurs both free and bound in E, but leaves open the possibility that distinct bound variables have the same name, as in the terms $\lambda x(\lambda x(y))$ or $\lambda x(x)(\lambda x(x))$. We formulate the convention in a more strict way to exclude these possibilities.

Note also that in the Barendregt variable convention, the λ-term E is not being thought of as an equivalence class of λ-terms, but rather as a particular element from this class, i.e., a specific expression. This is an example of our casually moving between talk of λ-terms as specific strings of symbols (syntactic objects) and talk of λ-terms as equivalence classes of such syntactic objects.

The proof of the following theorem is straightforward and is left to the exercises.

Theorem 1.38

If E is a λ-term E obeying the Barendregt variable convention and E' is a subterm of E, then E' obeys the Barendregt variable convention.

Given any λ-term, there is always an α-equivalent λ-term that obeys the Barendregt variable convention. For instance, in $x(\lambda x(x))$ there are both free and bound occurrences of x, and in $\lambda x(\lambda x(zx))$ we have two declarations of the bound variable x. However, $x(\lambda x(x))$ is α-equivalent to $x(\lambda y(y))$, and $\lambda x(\lambda x(zx))$ is α-equivalent to $\lambda x(\lambda y(zy))$, where these later terms obey the Barendregt variable convention. The proof of the following theorem is also left to the exercises.

Theorem 1.39

For every λ-term E, there is a λ-term E' which is α-equivalent to E such that E' obeys the Barendegt variable convention.

If then instead of typing $\lambda x(\lambda x(x))$ we type the α-equivalent term $\lambda x(\lambda y(y))$ that obeys the Barendregt variable convention, we have the derivation:

$$\dfrac{\dfrac{x:\sigma,\ y:\tau\ \vdash\ y:\tau \qquad \text{(Var)}}{x:\sigma\ \vdash\ \lambda y(y):\tau\to\tau}\ \text{(Abs)}}{\vdash\ \ \lambda x(\lambda y(y)):\sigma\to(\tau\to\tau)}\ \text{(Abs)}$$

where no silent replacement of terms with α-equivalent terms is necessary. We will in fact shortly see a theorem one of whose consequences is that silent replacement of terms with α-equivalent terms can be avoided in typing derivations for λ-terms that obey the Barendregt variable convention.

In all our examples so far, we have proved that τ is a possible type of E by constructing a proof of the sequent $\vdash E:\tau$. That is to say, we have proved that in the *empty* context E can be assigned type τ. Is it always true that when E can be assigned type τ, we can prove $\vdash E:\tau$ in our system?

The answer is clearly no. Consider an *open* λ-term such as the variable x. This term can be given any type τ we please. However, for no τ can we prove $\vdash x:\tau$. For it is clear that the final step in such a proof could not be (App) or (Abs), as these rules produce conclusions $\Gamma\vdash E:\tau$ in which E is *complex* – in any application of (App), E will have the form $E_1(E_2)$, and in any application of (Abs), E will have the form $\lambda v(E)$ – while the variable x is simple. The last step of the proof must therefore be a leaf of the form (Var). However all leaves in $\mathbf{TR_0}$ have non-empty contexts and thus a leaf on its own cannot constitute a proof of $\vdash x:\tau$.

On the other hand, the proof consisting of nothing other than the leaf $x:\tau\vdash x:\tau$ shows that in some context (namely, the context $x:\tau$), x has the possible type τ. The following theorem tells us more generally that typing derivations for untyped *open* λ-terms always involve non-empty contexts in which the type of each free variable is declared

Theorem 1.40

In any proof of $\Gamma\vdash E:X$ in $\mathbf{TR_0}$, for every free variable v of E the context Γ contains a declaration of the form $v:\tau$ for some τ.

The proof is a straightforward induction on the construction of typing derivation in $\mathbf{TR_0}$ and is left to the exercises. As a kind of converse of this result, it can also easily be shown that if there is any typing derivation whose conclusion is of the form $\Gamma\vdash E:X$, then there is a typing derivation whose conclusion has the form $\Gamma'\vdash E:X$, where the context Γ' consists of nothing but variable declarations for the free variables of E. In fact, we will show something stronger. First, consider

the following theorem, which tells us that typing derivations for terms that obey the Barendregt variable convention may be assumed to have a very well behaved structure.

Theorem 1.41

Suppose E is a λ-term obeying the Barendregt variable convention, and that $\Gamma \vdash E : X$ in $\mathbf{TR_0}$. Then there is a proof of $\Gamma' \vdash E : X$ in $\mathbf{TR_0}$ for some $\Gamma' \subseteq \Gamma$ such that (i) there is no silent replacement of λ-terms by α-equivalent terms in this proof, (ii) the variables whose types are declared in Γ' are precisely the free variables of E, (iii) all the elements of Γ' are distinct, and (iv) every variable declaration in the proof is either an element of Γ', or has the form $v : \tau$ where v is a bound variable of E.

The proof of this theorem is again is a straightforward induction on the typing derivations of $\mathbf{TR_0}$ and is left to the exercises. We then have the following corollary.

Corollary 1.42

Suppose there is a typing derivation of $\Gamma \vdash E : X$ in $\mathbf{TR_0}$ (where E does not necessarily obey the Barendregt variable convention.) Then there is a typing derivation $\Gamma' \vdash E : X$ in $\mathbf{TR_0}$ such that Γ' has the form $v_1 : \tau_1, v_2 : \tau_2, ..., v_n : \tau_n$, where $v_1, ..., v_n$ are the free variables of E. (Note that there is no repetition.) Moreover, in this typing derivation there is at most one silent replacement of a λ-term by an α-equivalent term, and it occurs in the last step.

Proof

Suppose we have a typing derivation T_0 whose conclusion is a sequent $\Gamma \vdash E : X$ in $\mathbf{TR_0}$, and let E' be a term α-equivalent to E that obeys the Barendregt variable convention. By silently replacing E with the α-equivalent term E' in the last step of T_0, we can construct a typing derivation T_1 of $\Gamma \vdash E' : X$. Applying Theorem 1.41, we then have a typing derivation T_2 of $\Gamma' \vdash E' : X$ such that Γ' is a list of typing declarations for all the free variables of E' (and no others) without repetition, and such that there is no silent replacement of λ-terms by α-equivalent λ-terms in T_2. Note that the free variables of E and E' are the same. If in the last step of T_2 we introduce a silent replacement of E' with E, we then have a typing derivation T_3 whose conclusion is $\Gamma' \vdash E : X$ in $\mathbf{TR_0}$. In T_3, Γ' consists only of typing declarations for all the free variables of E (and no others) without repetition. There is also at most one silent replacement of λ-terms by α-equivalent terms in this proof, and it occurs in the last step.

This tells us that even typing derivations for terms that do not obey the Barendregt variable convention may be assumed to have a well behaved structure.

We now turn to the question of in what sense $\mathbf{TR_0}$ allows us to prove all the basic facts about typing that we might want to prove. Indeed, there is a sense in which $\mathbf{TR_0}$ is sound and complete. This is captured by the following theorem:

Theorem 1.43: Soundness and Completeness of $\mathbf{TR_0}$.

Suppose E is an untyped λ_0-term whose free variables are $v_1, ..., v_n$, and let $\tau_1, ..., \tau_n$ be simple types. Then the following are equivalent:

(i) It is possible to assign simple types (i.e., types of λ_0^{type}) to all the variables of E in such a way that the free variables $v_1, ..., v_n$ of E are assigned types $\tau_1, ..., \tau_n$, and E itself has type X.

(ii) There is a proof of $\Gamma \vdash E : X$ in $\mathbf{TR_0}$, where the context Γ includes the variable declarations $v_1 : \tau_1,\ v_2 : \tau_2, ...,\ v_n : \tau_n$.

A consequence of this theorem is that $\mathbf{TR_0}$ proves all true claims of the form

$$E \text{ has } X \text{ as a possible type in context } \Gamma$$

for any untyped λ-term E, type X, and context Γ in which types are assigned to all the free variables of E. In this sense, the system $\mathbf{TR_0}$ proves every basic fact we might want to know about typing.

Unlike many soundness and completeness proofs in logic, the proof of Theorem 1.43 is fairly straightforward. That **(i)** entails **(ii)** (the 'completeness' of $\mathbf{TR_0}$) can be proven by an easy induction on the construction of untyped λ-terms E, and uses Theorem 1.41. That **(ii)** entails **(i)** (the 'soundness' of $\mathbf{TR_0}$) can be proven by an easy induction on the construction of proofs of $\mathbf{TR_0}$. The proof is left to the appendix, but you should try proving both directions on your own before consulting it.

A final important result is the following:

Theorem 1.44: Subject Reduction Theorem for $\mathbf{TR_0}$.

In $\mathbf{TR_0}$, if $\Gamma \vdash M : \sigma$ and $M \twoheadrightarrow_{\beta\eta} N$, then $\Gamma \vdash N : \sigma$.

This theorem tells us that neither β-reduction nor η-reduction decreases the set of possible types an untyped term may be assigned in the Curry-style typed λ-calculus. The proof of this theorem is also left to the exercises. Note, however, that reduction can *increase* the set of possible types an untyped term can be assigned. You will also demonstrate this in an exercise.

The Curry-style typed λ-calculus is not really a fundamentally different sort of thing from the Church-style typed λ-calculus. In both cases, we are ultimately concerned with the way in which untyped λ-terms (that is, terms of λ_0) may be typed (that is, transformed into terms of λ_0^{type}.) Church and Curry simply used different formalisms to analyse this basic phenomenon. Both formalisms will be useful to us in what follows.

Exercises for Section 1.12

1. Construct typing derivations in $\mathbf{TR_0}$ that prove the following.

(i) $\vdash \lambda x(x(\lambda y(y))) : ((\sigma \to \sigma) \to \tau) \to \tau$
(ii) $\vdash \lambda x \lambda y(y(x(y))) : ((\tau \to \sigma) \to \tau) \to ((\tau \to \sigma) \to \sigma)$
(iii) $\vdash \lambda x \lambda y \lambda z(x(z)(y(z))) : (\sigma \to (\tau \to \rho)) \to ((\sigma \to \tau) \to (\sigma \to \rho))$.

2. Construct a typing derivation in $\mathbf{TR_0}$ proving a conclusion of the form

$$\vdash \lambda x(x)(\lambda y(y)(\lambda z(z))) : \tau$$

for some τ. Then construct a typing derivation of $\vdash s : \tau$ for the same τ, where s is the β-normal form of $\lambda x(x)(\lambda y(y)(\lambda z(z)))$.

3. Construct an untyped closed λ-term α such that α cannot be assigned any type in the Curry-style λ-calculus (and thus there is no proof of $\vdash \alpha : \tau$ for any τ) , but for some untyped λ-term α' with $\alpha \twoheadrightarrow_\beta \alpha'$ the term α' *can* be assigned a type in the Curry-style λ-calculus (that is, there is some proof of $\vdash \alpha' : \sigma$ for some σ.)

4. Construct an untyped closed λ-term α such that α can be assigned a type in the Curry-style λ-calculus, but for some untyped λ-term α' with $\alpha \twoheadrightarrow_\beta \alpha'$ the term α' can be assigned a type in the Curry-style λ-calculus that α cannot be assigned.

5. Prove Theorem 1.38.

6. Prove Theorem 1.39.

7. Prove Theorem 1.40.

8. Prove Theorem 1.41.

9. Are the following true or false? If true, provide a proof. If false, provide a counterexample and explain why it is a counterexample.

(i) If $\Gamma \vdash E : X$, v is a variable not declared in Γ, and Y is a type, then $\Gamma, v : Y \vdash E : X$. (Sometimes this is called *weakening*.)
(ii) If $\Gamma, v : Y, v : Y \vdash E : X$, then $\Gamma, v : Y \vdash E : X$. (Sometimes this is called *contraction*.)
(iii) If $\Gamma \vdash E : X$, then every variable declaration in the proof is either an element of Γ, or has the form $v : \tau$ where v is a bound variable of E.
(iv) If $\Gamma \vdash E : X$ and there is no silent replacement of λ-terms by α-equivalent terms in this proof, then E obeys the Barendregt variable convention.
(v) If $\Gamma \vdash E : X$ and there is no silent replacement of λ-terms by α-equivalent terms in this proof, then every variable declaration in the proof is either an element of Γ, or has the form $v : \tau$ where v is a bound variable of E.

10. Prove Theorem 1.44. (Hint: use the Soundness and Completeness Theorem to reduce the

problem to that of showing that if $M \rightarrow_\beta N$, and if consistent with the free variables of M being assigned certain types, M itself may be assigned the type σ, then consistent with the free variables of N being assigned those same types, N itself may also be assigned the type σ.)

1.13 The Bound-typed λ-calculus.

In this section, we describe a second version of the Curry-style λ-calculus that will be used very frequently in the remainder of the book.

One slight oddity of the terminology

$$\Gamma \vdash E : X$$

is that it only expresses the idea that under the typing assumptions in Γ, the λ-term E *may* be given the type X. This does not mean that E *must* be given the type X. For example, for any type A we can easily derive in $\mathbf{TR_0}$ that

$$\vdash \lambda x(x) : A \rightarrow A.$$

For each type A, this means that the type $A \rightarrow A$ is a type that $\lambda x(x)$ *may* receive. For no particular A is $A \rightarrow A$ the type that $\lambda x(x)$ *must* receive.

However, typically in logic the relation $\Gamma \vdash S$ denotes a type of *necessitation*, and says that if Γ is true, then S *must* be true. We can bring our notation $\Gamma \vdash E : X$ into conformity with this idea of necessitation by moving to a so-called *partially typed* λ-calculus, which we now explain.

Consider the following definition

Definition 1.45: Bound-typed λ-terms.

A λ-term is *bound-typed* if all its bound variables are typed, and all its free variables are untyped.

So for example, the following terms are all bound-typed:

$\lambda x^\alpha(x^\alpha)$,
$\lambda x^\alpha(yx^\alpha)$,
$\lambda x^\alpha(x^\alpha(\lambda z^\beta(yz^\beta)))$.

Using our former conventions, when there is no ambiguity we can eliminate the type symbols on bound variables except when they are 'declared' immediately next to a λ symbol. For example, we can write the above three expressions more compactly as follows:

$\lambda x^\alpha(x)$,
$\lambda x^\alpha(yx)$,

$\lambda x^{\alpha}(x(\lambda z^{\beta}(yz)))$.

Some bound-typed λ-terms can have further superscripts added to them and be transformed to a full λ-term of λ_0^{type}. In this way, such terms then acquire an overall type. For example, consider the term $\lambda x^{\alpha}(yx)$ (that is, $\lambda x^{\alpha}(yx^{\alpha})$.) Given the occurrence of the subexpression yx^{α}, for this to be transformed into a grammatical λ-term of λ_0^{type} the variable y must have a type of the form $\alpha \to \beta$. Assuming then that y has type $\alpha \to \beta$, yx^{α} has type β, and so the bound-typed λ-term $\lambda x^{\alpha}(yx^{\alpha})$ is transformed into the λ_0^{type} term $\lambda x^{\alpha}(y^{\alpha\to\beta}x^{\alpha})$ with overall type $\alpha \to \beta$. The set of possible types of $\lambda x^{\alpha}(yx^{\alpha})$ is therefore the set of types of the form $\alpha \to \beta$.

As before, we will say that bound-typed λ-terms that are identical up to a renaming of bound variables are α-equivalent. (All the delicate provisos discussed earlier still apply.) When renaming bound variables, any types associated with such variables remain fixed. So for example, the terms

$$\lambda x^{\alpha}(x) \qquad \lambda y^{\alpha}(y)$$

are α-equivalent, but

$$\lambda x^{\alpha}(x) \qquad \lambda y^{\beta}(y)$$

are not.

In the version of the Curry-style typed λ-calculus we develop in this section, we will be assigning types to bound-typed terms. We do so by deriving sequents of the form

$$v_1 : \tau_1,\ v_2 : \tau_2,\ ...,\ v_n : \tau_n \vdash E : \sigma \tag{19}$$

where E is a bound-typed term and the list of variables $v_1, v_2, ..., v_n$ includes (but is not necessarily limited to) all the free variables of E. As before, this sequent will have the interpretation

> Consistent with the constraint that the variables $v_1, v_2, ..., v_n$ of E are assigned types $\tau_1, \tau_2, ..., \tau_n$ respectively, it is possible to type E so that it becomes a λ_0^{type} term of type σ.

Note however that the type of a term is completely determined by specifying the types of all its variables (both free and bound). Because in a bound-typed term the types of bound variables are already given, it follows that the type of a bound-typed term is uniquely determined by the types of its free variables. This means, however, that

$$v_1 : \tau_1,\ v_2 : \tau_2,\ ...,\ v_n : \tau_n \vdash E : \sigma$$

will be true for at most one σ. Thus we may interpret (19) as saying

> If the variables $v_1, v_2, ..., v_n$ are assigned types $\tau_1, \tau_2, ..., \tau_n$ respectively, then E *must* be assigned type σ.

In this way, our type-theoretic sequents now make a claim about *necessitation*, that is, what *must* be the case if something else is the case.

In the version of the λ-calculus we consider here, we will be able to derive sequents such as

$$y : \alpha \to \beta \vdash \lambda x^{\alpha}(yx) : \alpha \to \beta.$$

This sequent may be understood as saying that if in $\lambda x^{\alpha}(yx)$ the free variable y gets assigned the type $\alpha \to \beta$, then the entire term $\lambda x^{\alpha}(yx)$ *must* get assigned the type $\alpha \to \beta$.

We will also allow sequents such as

$$y : \alpha \to \beta,\ u : \gamma \vdash \lambda x^{\alpha}(yx) : \alpha \to \beta.$$

This says that if in $\lambda x^{\alpha}(yx)$ any free occurrence of y gets assigned the type $\alpha \to \beta$ and any free occurrence of u gets assigned the type γ, then the entire term $\lambda x^{\alpha}(yx)$ *must* get assigned the type $\alpha \to \beta$. This is trivially true because, in addition to the previous considerations, u does not actually occur free in the λ-term in question.

We furthermore even allow sequents such as

$$y : \alpha \to \beta,\ x : \gamma \vdash \lambda x^{\alpha}(yx) : \alpha \to \beta.$$

Such a sequent looks confusing at first, as x appears to be typed differently on the left and right. However as before we will allow 'silent' replacing of λ-terms with α-equivalent λ-terms. This means that the sequent just written can be re-expressed in a less confusing way as follows

$$y : \alpha \to \beta,\ x : \gamma \vdash \lambda u^{\alpha}(yu) : \alpha \to \beta,$$

where we have replaced the λ-term $\lambda x^{\alpha}(yx)$ with the α-equivalent term $\lambda u^{\alpha}(yu)$. This later sequent is of course entirely unproblematic.

Let us think about the rules that our new system of sequents should obey. First of all, as before we can keep the (Var) rule

$$\Gamma, v : \tau \vdash v : \tau \quad \text{(Var)}$$

where v is a variable. The variable v of course a bound-typed λ-term, and thus this axiom just makes the trivial claim that if among other variable declarations v has been given type τ, then the bound-typed term v must be given type τ.

Next, the (App) rule can also be used in its familiar form

$$\frac{\Gamma \ \vdash \ E_1 : \tau \to \sigma \qquad \Gamma \ \vdash \ E_2 : \tau}{\Gamma \ \vdash \ E_1(E_2) : \sigma} \ \text{(App)}$$

where E_1 and E_2 are bound-typed terms. For if in context Γ the bound-typed terms E_1 and E_2 must be assigned types $\tau \to \sigma$ and τ respectively, then in this same context Γ the term $E_1(E_2)$ (which will also be bound-typed) must be assigned type σ.

The (Abs) rule will clearly require a small modification, as follows

$$\frac{\Gamma, v : \tau \ \vdash \ E : \sigma}{\Gamma \ \vdash \ \lambda v^{\tau}(E) : \tau \to \sigma} \ \text{(Abs}^{\mathbf{bt}}\text{)}$$

Here E is a bound-typed term. The only difference between this and the previous version of the (Abs) rule is that in the conclusion, the bound variable v in the λ-term is explicitly assigned type τ. (We indicate this difference with the superscript **bt** in Abs$^{\mathbf{bt}}$, indicating that it is the 'bound-typed' variant of the rule.) Suppose that as stated in the premise of this rule, in the context Γ and under the assumption that the variable v has type τ, the bound-typed term E must receive the type σ. This then means that in the context Γ, the bound-typed term $\lambda v^{\tau}(E)$ must receive the type $\tau \to \gamma$.

In sum, we have the following typing rules for our system:

Definition 1.46: Rules of the bound-typed λ-calculus $\mathbf{TR}_0^{\mathbf{bt}}$.

(i) $\Gamma, v : \tau \vdash v : \tau$ (Var)

(ii) $$\frac{\Gamma \vdash E_1 : \tau \to \sigma \qquad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_1(E_2) : \sigma}$$ (App)

(iii) $$\frac{\Gamma, v : \tau \vdash E : \sigma}{\Gamma \vdash \lambda v^{\tau}(E) : \tau \to \sigma}$$ (Abs$^{\mathbf{bt}}$)

where v is a variable, E_1, E_2 and E are bound-typed terms of the λ-calculus, σ, τ are types, and all contexts are assumed to be grammatical.

We call this set of rules $\mathbf{TR}_0^{\mathbf{bt}}$. It is of course simply a trivial variant of $\mathbf{TR_0}$, with the obvious types attached to bound variables in λ-terms where needed.

Even though their interpretations are different, any proof in $\mathbf{TR_0}$ can be modified easily to give a proof in $\mathbf{TR}_0^{\mathbf{bt}}$. For example, the following proof in $\mathbf{TR_0}$:

$$\dfrac{\dfrac{\dfrac{x : \sigma,\ y : \sigma \to \tau \vdash x : \sigma \qquad x : \sigma,\ y : \sigma \to \tau \vdash y : \sigma \to \tau}{x : \sigma,\ y : \sigma \to \tau \vdash yx : \tau}\text{(App)}}{x : \sigma \vdash \lambda y(yx) : (\sigma \to \tau) \to \tau}\text{(Abs)}}{\vdash \lambda x(\lambda y(yx)) : \sigma \to ((\sigma \to \tau) \to \tau)}\text{(Abs)}$$

can be transformed into the following proof in $\mathbf{TR}_0^{\mathbf{bt}}$:

$$\dfrac{\dfrac{\dfrac{x : \sigma,\ y : \sigma \to \tau \vdash x : \sigma \qquad x : \sigma,\ y : \sigma \to \tau \vdash y : \sigma \to \tau}{x : \sigma,\ y : \sigma \to \tau \vdash yx : \tau}\text{(App)}}{x : \sigma \vdash \lambda y^{\sigma \to \tau}(yx) : (\sigma \to \tau) \to \tau}\text{(Abs}^{\mathbf{bt}}\text{)}}{\vdash \lambda x^{\sigma}(\lambda y^{\sigma \to \tau}(yx)) : \sigma \to ((\sigma \to \tau) \to \tau)}\text{(Abs}^{\mathbf{bt}}\text{)}$$

by typing the bound variables in the λ-terms in the obvious way. One can also go from the $\mathbf{TR}_0^{\mathbf{bt}}$ proof to the original $\mathbf{TR_0}$ proof by simply removing all types from bound variables. So the structure of proofs in $\mathbf{TR_0}$ and $\mathbf{TR}_0^{\mathbf{bt}}$ are for all intents and purposes the same.

As a result, for many theorems of $\mathbf{TR_0}$ there is a corresponding theorem about $\mathbf{TR}_0^{\mathbf{bt}}$. For example, we have the following.

Theorem 1.47

In any proof of $\Gamma \vdash E : X$ in $\mathbf{TR_0^{bt}}$, for every free variable v of E the context Γ contains a declaration of the form $v : \tau$ for some τ.

Theorem 1.48

Suppose E is a bound-typed λ-term obeying the Barendregt variable convention, and that $\Gamma \vdash E : X$ in $\mathbf{TR_0^{bt}}$. Then there is a proof of $\Gamma' \vdash E : X$ in $\mathbf{TR_0^{bt}}$ for some $\Gamma' \subseteq \Gamma$ such that (i) there is no silent replacement of λ-terms by α-equivalent terms in this proof, (ii) the variables whose types are declared in Γ' are precisely the free variables of E, (iii) all the elements of Γ' are distinct, and (iv) every variable declaration in the proof is either an element of Γ', or has the form $v : \tau$ where v is a bound variable of E.

Theorem 1.49: Soundness and Completeness of $\mathbf{TR_0^{bt}}$.

Suppose E is a bound-typed λ-term whose free variables are $v_1, ..., v_n$, and let $\tau_1, ..., \tau_n$ be types. Then the following are equivalent:

(i) If the variables $v_1, v_2, ..., v_n$ are assigned types $\tau_1, \tau_2, ..., \tau_n$ respectively, then E must be assigned type σ.

(ii) There is a proof of $\Gamma \vdash E : X$ in $\mathbf{TR_0^{bt}}$, where the context Γ includes the variable declarations $v_1 : \tau_1,\ v_2 : \tau_2, ...,\ v_n : \tau_n$.

Theorem 1.50: Subject Reduction Theorem for $\mathbf{TR_0^{bt}}$.

In $\mathbf{TR_0^{bt}}$, if $\Gamma \vdash M : \sigma$ and $M \twoheadrightarrow_{\beta\eta} N$, then $\Gamma \vdash N : \sigma$.

These proofs may all be obtained by minor modifications of the proofs of the corresponding theorems for $\mathbf{TR_0}$. Verifying this is left to the exercises and the appendix to this chapter.

In the chapters that follow, we will be interested in expanding the system $\mathbf{TR_0}$ to richer systems $\mathbf{TR_1}$, $\mathbf{TR_2}$, ... in which we reason in more complicated ways about the typing of untyped λ-terms. As in the case of $\mathbf{TR_0}$, trivial modifications of these systems will yield richer systems $\mathbf{TR_1^{bt}}$, $\mathbf{TR_2^{bt}}$, ... in which we reason in more complicated ways about the typing of bound-typed λ-terms.

Exercises for Section 1.13

1. Construct typing derivations in $\mathbf{TR_0^{bt}}$ that prove the following.

(i) $\vdash \lambda x^{\sigma}(\lambda y^{\tau}(x)) : \sigma \to (\tau \to \sigma)$
(ii) $\vdash \lambda x^{\sigma\to(\tau\to\rho)}\lambda y^{\sigma\to\tau}\lambda z^{\sigma}(x(z)(y(z))) : (\sigma \to (\tau \to \rho)) \to ((\sigma \to \tau) \to (\sigma \to \rho))$.

2. Prove Theorem 1.47.

3. Prove Theorem 1.48.

4. Prove Theorem 1.50.

1.14 Other Resources

For a basic and very readable introduction to the λ-calculus, see chapters 1 through 4 of Selinger's *Lecture Notes on the Lambda Calculus* [9]. For a more extended treatment, see Hindley and Seldin's *Lambda-Calculus and Combinators* [4]. For an extremely detailed treatment, see Barendregt's *The Lambda Calculus, Its Syntax and Semantics* [1]. Chapters 1 and 2 of Hindley's *Basic Simple Type Theory* [5] also contain a more condensed but very good introduction to the untyped λ-calculus, and chapters 1 and 3 of Sorenson and Urzyczyn's *Lectures on the Curry-Howard Isomorphism* [10] contains a very good introduction to both the untyped and typed λ-calculus.

For a more general discussion of the themes of the chapter, Barendregt's *The Lambda Calculus, Its Syntax and Semantics* [2] is also worth reading.

1.15 Appendix

In this section, we provide proofs for several results in the main text.

Theorem 1.13: The Church-Rosser Theorem for β-reductions in λ_0.

In the untyped λ-calculus λ_0, if $X \twoheadrightarrow_\beta Y_1$ and $X \twoheadrightarrow_\beta Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$.

Proof of Theorem 1.13.

In this proof, all λ-terms are understood to be terms of the untyped λ-calculus λ_0. Without loss of generality, we assume all terms obey the Barendregt variable convention.

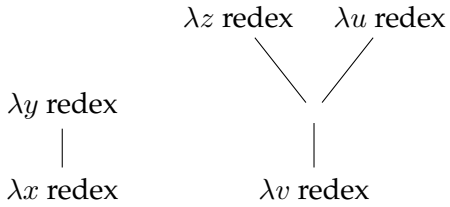
It is useful to begin by defining a more general notion of reduction. Given a λ-term e, there may be many distinct 1-step β-reductions that could potentially be performed on it,

each corresponding to a different redex of the form $\lambda v(t_1)(t_2)$. Given any two distinct redexes $r = \lambda v(t_1)(t_2)$ and $r' = \lambda v'(t_1')(t_2')$, they will either be disjoint, r will be a subexpression of r', or r' will be a subexpression of r. The set of redexes contained in e thus form a finite set of trees under the relation of inclusion which we call the *redex trees* of e; we assume these trees are oriented so that redexes containing no other redexes are at the top.

For example, if e is

$$\underbrace{\lambda x(x)(\underbrace{\lambda y(y)(y)}_{\lambda y \text{ redex}})}_{\lambda x \text{ redex}} \; \underbrace{w(\lambda v(v(\underbrace{\lambda z(z)(z)}_{\lambda z \text{ redex}}))(\lambda w(\underbrace{\lambda u(u)(u)}_{\lambda u \text{ redex}})))}_{\lambda v \text{ redex}} \tag{20}$$

then there are 5 redexes in e as depicted. The redex trees of e are then as follows



Note that if one redex r' lies above another redex $r = \lambda v(t_1)(t_2)$ in the redex trees, then not only is r' a subexpression of r, but it is in fact a (not necessarily proper) subexpression of t_1 or t_2. This is because r' is a proper subexpression of $r = \lambda v(t_1)(t_2)$, and the only proper subexpressions of $\lambda v(t_1)(t_2)$ are $\lambda v(t_1)$ and the (not necessarily proper) subexpressions of t_1 and t_2. The term $\lambda v(t_1)$ does not have the form of a redex, and thus r' must be a (not necessarily proper) subexpression of t_1 or t_2.

As a consequence of this, if r' lies above r and a 1-step β-reduction is performed on r', the redex $r = \lambda v(t_1)(t_2)$ is transformed into a term $\lambda v(t_1^*)(t_2)$ or $\lambda v(t_1)(t_2^*)$. Either way, this still has the form of a redex. Because of this, it is possible to perform β-reduction on as many (or as few) of the redexes contained in the redex trees of e that we want, so long as we work from the innermost redexes outwards – that is, so long as we reduce redexes starting with those uppermost in the redex trees, then continue reducing other redexes in the redex trees of e, never moving upwards. If doing so produces a term e', we will say that $e \Rightarrow_\beta e'$. If we reduce *all* the redexes contained in e (working from the uppermost leaves of the redex trees and moving downwards), we call the resulting term e^* (or when there is risk of confusion, $[e]^*$), which we call the *complete reduction* of e. (Note that the order in which we perform β-reductions associated with disjoint redexes does not matter.) By definition, we then have $e \Rightarrow_\beta e^*$. For example, for the term e^* given in (20), the complete reduction of e is given by $e^* = yw(\lambda w(u)(z))$. (Note that this term is not necessarily in β-normal form.)

We then have the following sequence of lemmas.

Lemma 1: For any λ-terms e and e', $e \rightarrow_\beta e'$ entails $e \Rightarrow_\beta e'$.
Proof: The reduction $e \rightarrow_\beta e'$ corresponds to the case in which we perform a β-reduction on only one redex of e, and so $e \Rightarrow_\beta e'$ is immediate from the definition of $\Rightarrow_\beta$. $\square$
Lemma 2: For any λ-terms e and e', $e \Rightarrow_\beta e'$ entails $e \twoheadrightarrow_\beta e'$.
Proof: If $e \Rightarrow_\beta e'$, the reduction of e to e' may be thought of as a sequence of β-reductions (never moving upwards in the redex trees of e), and thus $e \twoheadrightarrow_\beta e'$.
Lemma 3: For any λ-terms s, s', t, t', if $s \Rightarrow_\beta s'$ and $t \Rightarrow_\beta t'$, then $s[t/v] \Rightarrow_\beta s'[t'/v]$.
Proof: Assume $s \Rightarrow_\beta s'$ and $t \Rightarrow_\beta t'$. The term $s[t/v]$ may be β-reduced to the term $s'[t'/v]$ as follows. First reduce every occurrence of t that appears as a result for substituting t for v in $s[t/v]$ to t' (using $t \Rightarrow_\beta t'$), thereby obtaining $s[t'/v]$. Then perform the same reductions in $s \Rightarrow_\beta s'$ to obtain a β-reduction from $s[t'/v]$ to $s'[t'/v]$. (The fact that we have substituted t' for the free variable v has no effect on this sequence of reductions.) The result is a sequence of β-reductions from $s[t/v]$ to $s'[t'/v]$ which is easily seen to never move upwards in the redex trees of $s[t/v]$. Thus $s[t/v] \Rightarrow_\beta s'[t'/v]$. $\square$
Lemma 4: For any λ-terms s and t, if $s \Rightarrow_\beta t$ then $t \Rightarrow_\beta s^*$.

For a λ-term s, let height(s) be the length of the longest branch in the redex trees of s. (For example, if e is the term given in (20), then height$(e) = 2$.) The proof of the lemma is then by induction on height(s).

If height$(s) = 0$, this means that s contains no redexes, and so $s^* = s$. From $s \Rightarrow_\beta t$, we must also have $t = s$. Thus $t \Rightarrow_\beta s^*$ holds iff $s \Rightarrow_\beta s$ holds, which is immediate.

Suppose now that the result holds for terms s' with height$(s') \leq l$, and consider a term s with height$(s) = l + 1$. We begin by considering the case in which s itself is a redex, and so has the form

$$s = \lambda x(M)(N),$$

with height(M), height$(N) \leq l$. Note that we have

$$s^* = M^*[N^*/x],$$

because the complete reduction of $\lambda x(M)(N)$ is obtained by completely β-reducing M and N, and then performing the outmost and final β-reduction on the λx redex.

Suppose next that $s \Rightarrow_\beta t$.There are two cases to consider

(Case 1) In the reduction $s \Rightarrow_\beta t$, the redex s itself is among the redexes that are reduced.
(Case 2) In the reduction $s \Rightarrow_\beta t$, the redex s itself is not among the redexes that are reduced.

We consider these cases in turn. In Case 1, because any redex properly contained in $\lambda x(M)(N)$ must either be contained in M or contained in N, we know that t has the form

$$t = M'[N'/x] \quad \text{where } M \Rightarrow_\beta M' \text{ and } N \Rightarrow_\beta N'.$$

By inductive hypothesis, we have $M' \Rightarrow_\beta M^*$ and $N' \Rightarrow_\beta N^*$. From $t = M'[N'/x], M' \Rightarrow_\beta$

$M^*, N' \Rightarrow_\beta N^*$ and Lemma 3 we then have

$$t = M'[N'/x] \Rightarrow_\beta M^*[N^*/x] = s^*$$

so $t \Rightarrow_\beta s^*$ as desired. Consider now Case 2. In this case, t has the form

$$t = \lambda x(M')(N') \quad \text{where } M \Rightarrow_\beta M' \text{ and } N \Rightarrow_\beta N'.$$

Again, by inductive hypothesis, we have $M' \Rightarrow_\beta M^*$ and $N' \Rightarrow_\beta N^*$. From this we have

$$t = \lambda x(M')(N') \Rightarrow_\beta M^*[N^*/x] = s^*$$

and so $t \Rightarrow_\beta s^*$ as desired. So either way, $t \Rightarrow_\beta s^*$. This completes the inductive step for the case in which s is a redex.

If s is not itself a redex and height$(s) = l + 1$, then s can be written

$$s = ... \; r_1 \; ... \; r_2 \; ... \; r_i \; ...$$

where the r_i are the redexes at the bottom of the redex trees of s, and for all i, height$(r_i) \leq l$. Then we have

$$s^* = ... \; r_1^* \; ... \; r_2^* \; ... \; r_i^* \; ...$$

where the expressions occurring in ... remain unchanged. If $s \Rightarrow_\beta t$, then t may be written

$$t = ... \; r_1' \; ... \; r_2' \; ... \; r_i' \; ...$$

where $r_i \Rightarrow_\beta r_i'$ for each i. By inductive hypothesis we therefore have $r_i' \Rightarrow_\beta r_i^*$, and so

$$t = ... \; r_1' \; ... \; r_2' \; ... \; r_i' \; ... \Rightarrow_\beta ... \; r_1^* \; ... \; r_2^* \; ... \; r_i^* ... = s^*.$$

Thus $t \Rightarrow_\beta s^*$ as desired. This completes the induction and the proof of the lemma. □

In what follows, let $s^{[n]}$ be $s^{**\cdots*}$ with n asterisks, and let $s^{[0]}$ be s. We then have

Lemma 5: If $s_0 \Rightarrow_\beta s_1 \Rightarrow_\beta ... \Rightarrow_\beta s_{n-1} \Rightarrow_\beta s_n$ then $s_n^{[0]} \Rightarrow_\beta s_{n-1}^{[1]} \Rightarrow_\beta ... \Rightarrow_\beta s_1^{[n-1]} \Rightarrow_\beta s_0^{[n]}$

Proof: By induction on n. If $n = 1$, then from $s_0 \Rightarrow_\beta s_1$ and Lemma 4 we have $s_1 \Rightarrow_\beta s_0^*$ as desired.

Suppose then that the lemma is true for n, and that

$$s_0 \Rightarrow_\beta s_1 \Rightarrow_\beta ... \Rightarrow_\beta s_n \Rightarrow_\beta s_{n+1}. \tag{21}$$

By inductive hypothesis, from $s_0 \Rightarrow_\beta s_1 \Rightarrow_\beta ... \Rightarrow_\beta s_n$ we have

$$s_n^{[0]} \Rightarrow_\beta s_{n-1}^{[1]} \Rightarrow_\beta ... \Rightarrow_\beta s_1^{[n-1]} \Rightarrow_\beta s_0^{[n]}. \tag{22}$$

We have $s_n^{[0]} \Rightarrow_\beta s_{n-1}^{[1]}$ from (22) and $s_n^{[0]} \Rightarrow_\beta s_{n+1}^{[0]}$ (i.e., $s_n \Rightarrow_\beta s_{n+1}$) from (21). Applying Lemma 4 to both of these, we have $s_{n-1}^{[1]} \Rightarrow_\beta s_n^{[1]}$ and $s_{n+1}^{[0]} \Rightarrow_\beta s_n^{[1]}$. From $s_{n-1}^{[1]} \Rightarrow_\beta s_{n-2}^{[2]}$ and $s_{n-1}^{[1]} \Rightarrow_\beta s_n^{[1]}$ and Lemma 4 we have $s_{n-2}^{[2]} \Rightarrow_\beta s_{n-1}^{[2]}$ and $s_n^{[1]} \Rightarrow_\beta s_{n-1}^{[2]}$. Continuing this reasoning, we end up with $s_0^{[n]} \Rightarrow_\beta s_1^{[n]}$ and $s_2^{[n-1]} \Rightarrow_\beta s_1^{[n]}$. From $s_0^{[n]} \Rightarrow_\beta s_1^{[n]}$ and Lemma 4 we then have $s_1^{[n]} \Rightarrow_\beta s_0^{[n+1]}$. This reasoning can be depicted more intuitively with the following diagram

$$\begin{array}{ccccccccc}
s_n^{[0]} & \Longrightarrow & s_{n-1}^{[1]} & \Longrightarrow & s_{n-2}^{[2]} & \cdots\cdots & s_1^{[n-1]} & \Longrightarrow & s_0^{[n]} \\
\Downarrow & & \Downarrow & & \Downarrow & & \Downarrow & & \Downarrow \\
s_{n+1}^{[0]} & \Longrightarrow & s_n^{[1]} & \Longrightarrow & s_{n-1}^{[2]} & \cdots\cdots & s_2^{[n-1]} & \Longrightarrow & s_1^{[n]} \Longrightarrow s_0^{[n+1]}
\end{array}$$

Thus we have $s_{n+1}^{[0]} \Rightarrow_\beta s_n^{[1]} \Rightarrow_\beta ... \Rightarrow_\beta s_1^{[n]} \Rightarrow_\beta s_0^{[n+1]}$ as desired. This completes the proof of Lemma 5. □

To prove the Church-Rosser Theorem itself, suppose $X \twoheadrightarrow_\beta Y_1$ and $X \twoheadrightarrow_\beta Y_2$. There are then sequences $U_1, ..., U_n$ and $V_1, ..., V_m$ of terms such that

$$X \rightarrow_\beta U_1 \rightarrow_\beta ... \rightarrow_\beta U_n = Y_1 \quad \text{and} \quad X \rightarrow_\beta V_1 \rightarrow_\beta ... \rightarrow_\beta V_m = Y_2$$

and so by Lemma 1 we have

$$X \Rightarrow_\beta U_1 \Rightarrow_\beta ... \Rightarrow_\beta U_n = Y_1 \quad \text{and} \quad X \Rightarrow_\beta V_1 \Rightarrow_\beta ... \Rightarrow_\beta V_m = Y_2.$$

Letting l be $\max(n, m)$ and adding reductions of the form $e \Rightarrow_\beta e$ to the shorter of these two sequences, we then have

$$X \Rightarrow_\beta U_1 \Rightarrow_\beta ... \Rightarrow_\beta U_l = Y_1 \quad \text{and} \quad X \Rightarrow_\beta V_1 \Rightarrow_\beta ... \Rightarrow_\beta V_l = Y_2.$$

From Lemma 5 it then follows that

$$Y_1 \Rightarrow_\beta ... \Rightarrow_\beta X^{[l]} \quad \text{and} \quad Y_2 \Rightarrow_\beta ... \Rightarrow_\beta X^{[l]}.$$

From Lemma 2 we then have $Y_1 \twoheadrightarrow_\beta X^{[l]}$ and $Y_2 \twoheadrightarrow_\beta X^{[l]}$. Thus $X^{[l]}$ is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$. □

Theorem 1.20: The Church-Rosser Theorem for $\beta\eta$-reductions in λ_0.

In the untyped λ-calculus λ_0, if $X \twoheadrightarrow_{\beta\eta} Y_1$ and $X \twoheadrightarrow_{\beta\eta} Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_{\beta\eta} Z$ and $Y_2 \twoheadrightarrow_{\beta\eta} Z$.

Proof of Theorem 1.20.

In this proof, all λ-terms are understood to be terms of the untyped λ-calculus λ_0. As before, we assume all terms obey the Barendregt variable convention. The proof follows the same strategy as the proof of Theorem 1.13, with which familiarity is assumed. We mostly restrict ourselves to just describing the way in which this earlier proof must be modified to produce a proof of Theorem 1.20.

Again, we define a more general notion of $\beta\eta$-reduction that we denote $\Rightarrow_{\beta\eta}$. As before, for a given λ-term e the set of β- or η-redexes contained in e form a finite set of trees when ordered under the relation of inclusion, with the innermost redexes lying at the top. We would like it to be the case that whenever one redex r lies higher than another redex $\lambda x(M)(N)$ or $\lambda x(E(x))$ (with x free in E) that r is a subexpression of M or N or E, so that when a redex in a tree is reduced, the redexes below maintain their forms as redexes. But there are (precisely) two sorts of counterexamples to this:

(a)
$$\begin{array}{c} \lambda x(E(x)) \\ | \\ \lambda x(E(x))(N) \end{array}$$

(b)
$$\begin{array}{c} \lambda x(M)(z) \\ | \\ \lambda z(\lambda x(M)(z)) \end{array}$$

(a) the η-redex $\lambda x(E(x))$ is a subexpression of of the β-redex $\lambda x(E(x))(N)$ (where x is not free in E), so that when $\lambda x(E(x))$ is reduced to E, $\lambda x(E(x))(N)$ is reduced to $E(N)$ (which need not be a redex)

(b) the β-redex $\lambda x(M)(z)$ is a subexpression of of the η-redex $\lambda z(\lambda x(M)(z))$ (where z is not free in $\lambda x(M)$), so that when $\lambda x(M)(z)$ is reduced to $M[z/x]$, $\lambda z(\lambda x(M)(z))$ is reduced to $\lambda z(M[z/x])$ (which need not be a redex)

To deal with this problem, we will declare any redex in e of the form $\lambda x(E(x))(N)$ (where x is not free in E) or $\lambda z(\lambda x(M)(z))$ (where z is not free in $\lambda x(M)$) to be *prohibited*. If we perform a 1-step reduction on any non-prohibited redex, then any other non-prohibited redex below it maintains its form as a redex. We can therefore perform a 1-step β- or η-reduction on as many (or as few) of the non-prohibited redexes contained in the redex trees of e that we want, working from the innermost redexes outwards – that is, working from the uppermost redexes downwards. As before, if doing so produces a term e', we will say that $e \Rightarrow_\beta e'$. If in this way we reduce *all* the non-prohibited redexes contained in e, we call the resulting term e^* (or $[e]^*$),

which we call the *complete reduction* of e. (Again, the order in which we perform β-reductions associated with disjoint redexes does not matter.) By definition, we then have $e \Rightarrow_\beta e^*$.

We now have a similar sequence of lemmas to that contained in the proof of Theorem 1.13. Recall that $s \rightarrow_{\beta\eta} t$ means $s \rightarrow_\beta t$ or $s \rightarrow_\eta t$.

Lemma 1: For any λ-terms s and t, $s \rightarrow_{\beta\eta} t$ entails $s \Rightarrow_{\beta\eta} t$.
Proof: The reduction $s \rightarrow_{\beta\eta} t$ corresponds to the case in which a single redex r in the redex trees of s is reduced, with the result being t. If the redex r is not prohibited, then it is immediate that $s \Rightarrow_{\beta\eta} t$.

Suppose then that r is prohibited. Then r has one of the two forms **(a)** or **(b)** defined earlier. In case **(a)** it has the form of the β-redex $\lambda x(E(x))(N)$ (with x not free in E) , which β-reduces to $E(N)$. Note however that if in this term the η-redex $\lambda x(E(x))$ (which is a proper subterm of r) is reduced, the result is also $E(N)$.

In case **(b)**, a similar phenomenon occurs. In this case r has the form of the η-redex $\lambda z(\lambda x(M)(z))$, where z is not free in $\lambda x(M)$. This η-reduces to $\lambda x(M)$. However, if in this term the β-redex $\lambda x(M)(z)$ (which is a proper subterm of r) is reduced, the result is $\lambda z(M[z/x])$. Note that if z is not free in $\lambda x(M)$, then the terms $\lambda x(M)$ and $\lambda z(M[z/x])$ are α-equivalent, and thus the same.

As a consequence, we have

> If $s \rightarrow_{\beta\eta} t$ via the reduction of a prohibited redex r, then $s \rightarrow_{\beta\eta} t$ via the reduction of a redex r' properly contained in r. (23)

Of course, the redex r' given by (23) may also be prohibited. But then using the principle (23) again, we can find a redex r'' properly contained in r' such $s \rightarrow_{\beta\eta} t$ via the reduction of the redex r''. Because we cannot have an infinite sequence $r, r', r'', ...$ of redexes, each a subredex of the previous one on the list, there must be a non-prohibited redex r^* in s such that $s \rightarrow_{\beta\eta} t$ via the reduction of r^*. Thus, $s \Rightarrow_{\beta\eta} t$. □
Lemma 2: For any λ-terms s and t, $s \Rightarrow_{\beta\eta} t$ entails $s \twoheadrightarrow_{\beta\eta} t$.
Proof: Identical to that given in the proof of Theorem 1.13 □
Lemma 3: For any λ-terms s, s', t, t', if $s \Rightarrow_{\beta\eta} s'$ and $t \Rightarrow_{\beta\eta} t'$, then $s[t/v] \Rightarrow_{\beta\eta} s'[t'/v]$.
Proof: Exactly as in Theorem 1.13, but using $\beta\eta$-reduction instead of β-reduction. □
Lemma 4: For any λ-terms s and t, if $s \Rightarrow_{\beta\eta} t$ then $t \Rightarrow_{\beta\eta} s^*$.
Proof: In this proof, in order to deal with both types of reduction at once we will sometimes rewrite the β-redex $\lambda x(M)(N)$ more abstractly as $W(M, N)$ and the η-redex $\lambda x(E(x))$ more abstractly as $W(E)$. In this way, all redexes have the form $W(X, Y, ...)$ for some $X, Y, ...$. Given a redex $W(X, Y, ...)$, we will write $W^{\mathbf{r}}(X, Y, ...)$ for the reduction of this redex. So for example, if $W(M, N)$ is the β-redex $\lambda x(M)(N)$, then $W^{\mathbf{r}}(M, N)$ is the term $M[N/x]$, and if $W(E)$ is the η-redex $\lambda x(E(x))$ (with x not free in E), then $W^{\mathbf{r}}(E)$ is the term E.

Using this notation, we have the following

(P1) If a redex $r = W(X, Y, ...)$ is not prohibited and another redex r' is properly contained in r, then r' is properly contained in one of $X, Y, ...$

(P2) For any redex $W(X, Y, ...)$, if $X \Rightarrow_{\beta\eta} X', Y \Rightarrow_{\beta\eta} Y', ...$ then

$$W^{\mathbf{r}}(X, Y, ...) \Rightarrow_{\beta\eta} W^{\mathbf{r}}(X', Y', ...).$$

The property **(P1)** follows from our characterization of prohibited redexes. For a β-redex $\lambda x(M)(N)$, property **(P2)** amounts to the claim that if $M \Rightarrow_{\beta\eta} M'$ and $N \Rightarrow_{\beta\eta} N'$, then $M[N/x] \Rightarrow_{\beta\eta} M'[N'/x]$, which follows from Lemma 3. For an η-redex $\lambda x(E(x))$, property **(P2)** amounts to the trivial claim that if $E \Rightarrow_{\beta\eta} E'$ then $E \Rightarrow_{\beta\eta} E'$. So both **(P1)** and **(P2)** are true in general. They will be useful in what follows.

As in Theorem 1.13, Lemma 4 is proved by induction on height(s), where height(s) is the length of the longest branch in the redex trees of s (this includes prohibited redexes.) As before, the base case is trivial.

Suppose now that Lemma 4 holds for terms s' with height$(s') \leq l$, and consider a term s with height$(s) = l + 1$. Suppose initially that s is a redex, i.e.,

$$s = W(X, Y, ...).$$

Then note that we have

$$s^* = W^{\mathbf{r}}(X^*, Y^*, ...).$$

Suppose next that $s \Rightarrow_\beta t$. There are two cases to consider

(Case 1) In the reduction $s \Rightarrow_\beta t$, the redex s itself is among the redexes that are reduced.

(Case 2) In the reduction $s \Rightarrow_\beta t$, the redex s itself is not among the redexes that are reduced.

(Note of course that if in the reduction $s \Rightarrow_\beta t$ the redex s is reduced, it will be the last redex reduced, as it lies at the very bottom of the redex trees for s.)

Making use of **(P1)**, in Case 1 the term t has the form $W^{\mathbf{r}}(X', Y', ...)$ for some $X \Rightarrow_{\beta\eta} X', Y \Rightarrow_{\beta\eta} Y', ...$, while in Case 2 the term t has the form $W(X', Y', ...)$ for some $X \Rightarrow_{\beta\eta} X', Y \Rightarrow_{\beta\eta} Y', ...$ In either case, height(X), height(Y), ... are all strictly less than height(s), and so the inductive hypothesis holds of $X, Y, ...$ Thus we have $X' \Rightarrow_{\beta\eta} X^*, Y' \Rightarrow_{\beta\eta} Y^*, ...$

In Case 1 we then have $t = W^{\mathbf{r}}(X', Y', ...)$ and $X' \Rightarrow_{\beta\eta} X^*$, $Y' \Rightarrow_{\beta\eta} Y^*, ...$, and so using **(P2)** we have $t \Rightarrow_{\beta\eta} W^{\mathbf{r}}(X^*, Y^*, ...) = s^*$.

In Case 2 we have $t = W(X', Y', ...), X' \Rightarrow_{\beta\eta} X^*, Y' \Rightarrow_{\beta\eta} Y^*, ...$ and so $t \Rightarrow_{\beta\eta} W^{\mathbf{r}}(X^*, Y^*) = s^*$. (The reduction $t \Rightarrow_{\beta\eta} W^{\mathbf{r}}(X^*, Y^*, ...) = s^*$ is obtained by starting with $t = W(X', Y', ...)$, reducing $X' \Rightarrow_{\beta\eta} X^*, Y' \Rightarrow_{\beta\eta} Y^*, ...$ to obtain $W(X^*, Y^*, ...)$, then finally reducing the redex W to obtain $W^{\mathbf{r}}(X^*, Y^*, ...) = s$.)

Therefore, in either case we have $t \Rightarrow_{\beta\eta} s^*$ as desired. To complete the induction, we must finally consider the case in which s is not a redex. This is dealt with as in Theorem 1.13. $\square$

Given Lemma 4, the proof of Lemma 5 and remainder of the proof is the same as before, with only trivial notational changes (e.g., replacing β wherever it appears with $\beta\eta$.) $\square$

Theorem 1.31: Strong Normalizability of λ_0^{type} under β-reduction.

There is no infinite sequence $\tau_1, \tau_2, ...$ of terms in the typed λ-calculus λ_0^{type} for which

$$\tau_1 \rightarrow_\beta \tau_2 \rightarrow_\beta \tau_3 \rightarrow_\beta ...$$

Proof of Theorem 1.31.

In this proof, all λ-terms are understood to be terms of the typed λ-caclulus λ_0^{type}.

For each fundamental type τ, we add a new atomic, closed term d^τ of type τ to the typed λ-calculus. For any compound type $\tau_1 \rightarrow \tau_2$, we let $d^{\tau_1 \rightarrow \tau_2}$ be an abbreviation for $\lambda x^{\tau_1}(d^{\tau_2})$. In this way, d^τ is inductively defined for all simple types τ, and is always a closed term of type τ, already in β-normal form. (Without expanding the λ-calculus in this way, it is not true that for every type τ there is a closed term of type τ.) Showing that strong normalization holds for this expanded typed λ-calculus obviously entails that it holds for our original typed λ-calculus.

Definition: A λ-term e is *strongly normalizing* if there is no infinite sequence $e_0, e_1, e_2, ...$ of λ-terms with $e = e_0$ such that each e_{i+1} may be obtained from e_i by a single step β-reduction.

For all types τ, we define a set $[\![\tau]\!]$ of closed λ-terms of type τ as follows

Definition: For every type τ, $[\![\tau]\!]$ is given as follows

(i) If τ is fundamental, $[\![\tau]\!]$ is the set of strongly normalizing, closed terms of type τ.

(ii) If τ has the form $\tau_1 \rightarrow \tau_2$, $[\![\tau]\!]$ is the set of closed terms e of type τ such that for all $e' \in [\![\tau_1]\!]$, we have that $e(e') \in [\![\tau_2]\!]$.

It is easily seen (by induction on the construction of types) that every type can be written uniquely in the form

$$\tau_1 \rightarrow (\tau_2 \rightarrow (\tau_3 \rightarrow ... \rightarrow (\tau_n \rightarrow \sigma)...))$$

where σ is a fundamental type. We use this fact freely in what follows. The following alternate characterization of $[\![\tau]\!]$ is useful

Lemma 1: If a closed term e has type $\tau = \tau_1 \rightarrow (\tau_2 \rightarrow (\tau_3 \rightarrow ... \rightarrow (\tau_n \rightarrow \sigma)...))$ where σ is fundamental, then e is in $[\![\tau]\!]$ iff for all closed terms $e_1 \in [\![\tau_1]\!], e_2 \in [\![\tau_2]\!]..., e_n \in [\![\tau_n]\!]$, we have that $e(e_1)...(e_n)$ is strongly normalizing.

Proof: By induction on τ. If τ is fundamental, the result is immediate. Suppose then that τ has the form $\sigma_1 \rightarrow \sigma_2$ and the lemma holds of σ_2. Suppose further that σ_2 has the form

$\tau_1 \to (... \to (\tau_n \to \rho)...)$ where ρ is fundamental, so that $\sigma_1 \to \sigma_2$ can be written in the form $\sigma_1 \to (\tau_1 \to (... \to (\tau_n \to \rho)...))$. Then for any closed term e of type $\sigma_1 \to \sigma_2$, we have

$$\begin{array}{rll} e \in [\![\sigma_1 \to \sigma_2]\!] & \text{iff} & \text{for all closed terms } e' \in [\![\sigma_1]\!],\ e(e') \in [\![\sigma_2]\!] \text{ (by definition)} \\ & \text{iff} & \text{for all closed terms } e' \in [\![\sigma_1]\!], e_1 \in [\![\tau_1]\!], ...e_n \in [\![\tau_n]\!],\ e(e')(e_1)...(e_n) \\ & & \text{is strongly normalizing (applying inductive hypothesis to } e(e').) \end{array}$$

as desired. This completes the induction. $\square$.

The following technical lemma forms the heart of the proof.

Lemma 2: For any type τ,

(i) If $t \in [\![\tau]\!]$ and t is strongly normalizing, and e is a closed term with $e \in [\![\sigma]\!]$, then $\lambda x^\tau(e)(t) \in [\![\sigma]\!]$.

(ii) If $t \in [\![\tau]\!]$, and e is term of type σ with one free variable x^τ and $e[t/x^\tau] \in [\![\sigma]\!]$, then $\lambda x^\tau(e)(t) \in [\![\sigma]\!]$.

Proof: We first prove (i). Suppose the type σ can be written as

$$\sigma_1 \to (\sigma_2 \to (\sigma_3 \to ... \to (\sigma \to \rho)...)),$$

where ρ is a fundamental type. Suppose then that $t \in [\![\tau]\!]$, t is strongly normalizing, $e \in [\![\sigma]\!]$, and $\lambda x^\tau(e)(t) \notin [\![\sigma]\!]$. Then by Lemma 1, for some terms $e_1 \in [\![\tau_1]\!], e_2 \in [\![\tau_2]\!]..., e_n \in [\![\tau_n]\!]$ we have that

$$\lambda x^\tau(e)(t)(e_1)...(e_n) \tag{24}$$

is not strongly normalizing, so that we may perform an infinite sequence of β-reductions starting with (24). Fix some infinite sequence of β-reductions that begins with the term (24).

It is easily seen that the only redexes contained in a term of the form (24) are the redex $\lambda x^\tau(e)(t)$, and any redexs that occur as (not necessarily proper) subexpressions of $e, t, e_1, ..., e_n$. Thus, the first β-reduction must be either performed on (a) the initial redex $\lambda x^\tau(e)(t)$, or (b) a (not necessarily proper) subterm of one of $e, t, e_1, ..., e_n$. If the first reduction is not of form (a), then the newly reduced term can be written in the form $\lambda x^\tau(e^1)(t^1)(e_1^1)...(e_n^1)$ (where all but one of $e^1, t^1, e_1^1, ..., e_n^1$ is identical with the corresponding $e, t, e_1, ..., e_n$), and so the next β-reduction must be either performed on (a) the initial subterm $\lambda x^\tau(e^1)(t^1)$, or (b) a (not necessarily proper) subterm of one of $e^1, t^1, e_1^1, ..., e_n^1$. Iterating this reasoning, we see that in any infinite reduction sequence, either (c) a β-reduction involving the initial subterm $\lambda x^\tau(e^*)(t^*)$ with $e \twoheadrightarrow_\beta e^*$ and $t \twoheadrightarrow_\beta t^*$ is eventually performed at some point, or (d) the ith term of the reduction sequence has the form $\lambda x^\tau(e^i)(t^i)(e_1^i)...(e_n^i)$, where letting $e^0 = e, t^0 = t, e_1^0 = e_1, ..., e_n^0 = e_n$, we have that for each $i > 0$, exactly one of the terms $e^i, t^i, e_1^i, ..., e_n^i$ is obtained by a one-step β-reduction from $e^{i-1}, t^{i-1}, e_1^{i-1}, ..., e_n^{i-1}$, and the remainder of the $e^i, t^i, e_1^i, ..., e_n^i$ are identical to the corresponding $e^{i-1}, t^{i-1}, e_1^{i-1}, ..., e_n^{i-1}$. We consider these two cases (c) and (d) in turn.

In case (c), because the term e is closed, so too the term e^* is closed, and the redex $\lambda x^\tau(e^*)(t^*)$ thus β-reduces to e^*. Thus a term of the form

$$e^*(e_1^*)...(e_n^*)$$

with $e \twoheadrightarrow_\beta e^*$, $e_1 \twoheadrightarrow_\beta e_1^*$, ... $e_n \twoheadrightarrow_\beta e_n^*$ occurs in the infinite β-reduction sequence that starts with (24). Thus $e(e_1)...(e_n)$ is also not strongly normalizing, as $e(e_1)...(e_n) \twoheadrightarrow_\beta e^*(e_1^*)...(e_n^*)$. But if $e(e_1)...(e_n)$ is not strongly normalizing, then because $e_1 \in [\![\tau_1]\!], e_2 \in [\![\tau_2]\!]..., e_n \in [\![\tau_n]\!]$, it follows from Lemma 1 that $e \notin [\![\sigma]\!]$, which is a contradiction.

Consider then case (d). In this case, our infinite reduction sequence involves nothing more than reductions of the subexpressions $e, t, e_1, ..., e_n$ individually. Thus, at least one of the terms $e, t, e_1, ..., e_n$, must fail to be strongly normalizing. By assumption, t is strongly normalizing, and thus one of $e, e_1, ..., e_n$ must fail to be strongly normalizing. Thus $e(e_1)...(e_n)$ is not strongly normalizing, which as in case (c) contradicts $e \in [\![\sigma]\!]$. This completes the proof of (i).

The proof of (ii) is similar. This time, suppose that $t \in [\![\tau]\!]$, $e[t/x^\tau] \in [\![\sigma]\!]$, and $\lambda x^\tau(e)(t) \notin [\![\sigma]\!]$. Then for some terms $e_1 \in [\![\tau_1]\!], e_2 \in [\![\tau_2]\!]..., e_n \in [\![\tau_n]\!]$, we have that

$$\lambda x^\tau(e)(t)(e_1)...(e_n) \tag{25}$$

is not strongly normalizing. Fix some infinite sequence of β-reductions that begin with the term (25). As before, either (c) a β-reduction involving the initial subterm $\lambda x^\tau(e^*)(t^*)$ with $e \twoheadrightarrow_\beta e^*$ and $t \twoheadrightarrow_\beta t^*$ is eventually performed at some point, or (d) the infinite reduction sequence involves nothing more than reductions of the subexpressions $e, t, e_1, ..., e_n$ individually.

In case (c), the redex $\lambda x^\tau(e^*)(t^*)$ β-reduces to $e^*[t^*/x^\tau]$, and thus a term of the form

$$e^*[t^*/x^\tau](e_1^*)...(e_n^*)$$

with $e \twoheadrightarrow_\beta e^*$, $t \twoheadrightarrow_\beta t^*$, $e_1 \twoheadrightarrow_\beta e_1^*$, ... $e_n \twoheadrightarrow_\beta e_n^*$ occurs in the infinite β-reduction sequence that starts with (25). Thus $e[t/x^\tau](e_1)...(e_n)$ is not strongly normalizing (as $e[t/x^\tau](e_1)...(e_n) \twoheadrightarrow_\beta e^*[t^*/x^\tau](e_1^*)...(e_n^*)$), contradicting $e[t/x^\tau] \in [\![\sigma]\!]$.

In case (d), our infinite reduction sequence involves nothing more than reductions of the subexpressions $e, t, e_1, ..., e_n$ individually. Thus, at least one of the terms $e, t, e_1, ..., e_n$ must fail to be strongly normalizing. In any such case, the term $e[t/x^\tau](e_1)...(e_n)$ also fails to be strongly normalizing so long as we suppose, as we have, that the variable x^τ actually occurs in e, so that t actually occurs as a subexpression of $e[t/x^\tau]$. (We also invoke the easily proved fact that if there is an infinite reduction sequence starting from e, then there is an infinite reduction sequence starting from $e[t/x^\tau]$.) But $e[t/x^\tau](e_1)...(e_n)$ not being strongly normalizing contradicts $e[t/x^\tau] \in [\![\sigma]\!]$. □

Lemma 3: For any type τ,

(i) $d^\tau \in [\![\tau]\!]$
(ii) for all $t \in [\![\tau]\!]$, t is strongly normalizing.

Proof: By induction on τ. If τ is fundamental, (i) and (ii) are immediate from the fact that d^τ is in β-normal form and the definition of $[\![\tau]\!]$ for fundamental τ.

Suppose by inductive hypothesis that (i) and (ii) are true for types τ_1 and τ_2, and consider the type $\tau = \tau_1 \to \tau_2$. Suppose that τ_2 may be written $\sigma_1 \to (\sigma_2 \to ... \to (\sigma_n \to \rho))$, where ρ is fundamental. We may also suppose by inductive hypothesis that (i) and (ii) hold of $\sigma_1, ..., \sigma_n$.

To prove (i) for τ, note that by definition $d^\tau \in [\![\tau]\!]$ iff for all $t \in [\![\tau_1]\!]$, we have $d^\tau(t) \in [\![\tau_2]\!]$. Because d^τ is $\lambda x^{\tau_1}(d^{\tau_2})$, we have that $d^\tau \in [\![\tau]\!]$ iff

$$\text{for all } t \in [\![\tau_1]\!],\ \lambda x^{\tau_1}(d^{\tau_2})(t) \in [\![\tau_2]\!]. \tag{26}$$

Because by inductive hypothesis we know that all elements of $[\![\tau_1]\!]$ are strongly normalizing and that $d^{\tau_2} \in [\![\tau_2]\!]$, that (26) is true follows from Lemma 2(i).

To prove (ii) for τ, note that by inductive hypothesis we have $d^{\tau_1} \in [\![\tau_1]\!], d^{\sigma_1} \in [\![\sigma_1]\!], ..., d^{\sigma_n} \in [\![\sigma_n]\!]$. Thus, from $t \in [\![\tau]\!]$ and Lemma 1, we have that $t(d^{\tau_1})(d^{\sigma_1})...(d^{\sigma_n})$ is strongly normalizing. Because any subterm of a strongly normalizing term is strongly normalizing, it follows that t is strongly normalizing. □

Lemma 4: If e is a (not necessarily closed) λ-term of type σ with free variables $x_1^{\sigma_1}, x_2^{\sigma_2}, ..., x_n^{\sigma_n}$ and $t_1, ..., t_n$ are closed λ-terms of type $\sigma_1, ..., \sigma_n$ respectively with $t_1 \in [\![\sigma_1]\!]$, ..., $t_n \in [\![\sigma_n]\!]$, then

$$e[t_1/x_1^{\sigma_1}]...[t_n/x_n^{\sigma_n}] \in [\![\sigma]\!].$$

(Note that because the terms t_i are closed, the order of the substitutions $t_1/x_1^{\sigma_1}, ..., t_n/x_n^{\sigma_n}$ does not matter.)

Proof: By induction on the construction of e.

For the first base case, suppose e is just the variable x^σ. We must prove that if t is a closed λ-term of type σ with $t \in [\![\sigma]\!]$, then $x[t/x] \in [\![\sigma]\!]$, i.e., $t \in [\![\sigma]\!]$, which is immediate.

For the second base case, suppose e is d^σ for some fundamental type σ. Then e has no free variables, and it suffices to show $d^\sigma \in [\![\sigma]\!]$, which we have from Lemma 3.

Next suppose that e has the form $e_1(e_2)$ and has the type σ, so that e_1 has the type $\sigma \to \sigma'$ and e_2 has the type σ' for some type σ'. Suppose also that e has free variables $x_1^{\sigma_1}, x_2^{\sigma_2}, ..., x_n^{\sigma_n}$ and that $t_1, ..., t_n$ are closed λ-terms of type $\sigma_1, ..., \sigma_n$ with $t_1 \in [\![\sigma_1]\!]$, ..., $t_n \in [\![\sigma_n]\!]$. Then by inductive hypothesis we have $e_1[t_1/x_1]...[t_n/x_n] \in [\![\sigma' \to \sigma]\!]$ and $e_2[t_1/x_1]...[t_n/x_n] \in [\![\sigma']\!]$. From this and the definition of $[\![\sigma' \to \sigma]\!]$, it follows that

$$e_1[t_1/x_1]...[t_n/x_n](e_2[t_1/x_1]...[t_n/x_n]) \in [\![\sigma]\!].$$

From the basic properties of substitution we then have

$$e_1(e_2)[t_1/x_1]...[t_n/x_n] \in [\![\sigma]\!]$$

as desired.

Finally, suppose that e has the form $\lambda z^\sigma(u)$ where u has the type σ', so that e has the type $\sigma \to \sigma'$. Suppose that $e = \lambda z^\sigma(u)$ has free variables $x_1^{\sigma_1}, x_2^{\sigma_2}, ..., x_n^{\sigma_n}$ (all of course distinct from z^σ) and that $t_1 \in [\![\sigma_1]\!]$, ..., $t_n \in [\![\sigma_n]\!]$. To start, assume that the variable z^σ explicitly occurs free in u. The free variables of u are then $x_1^{\sigma_1}, x_2^{\sigma_2}, ..., x_n^{\sigma_n}$ and z^σ. By inductive hypothesis we have

$$\text{for all } t \in [\![\sigma]\!], \qquad u[t_1/x_1^{\sigma_1}]...[t_n/x_n^{\sigma_n}][t/z^\sigma] \in [\![\sigma']\!].$$

Applying Lemma 2(ii) (with $u[t_1/x_1^{\sigma_1}]...[t_n/x_n^{\sigma_n}]$ playing the role of e), we have

$$\text{for all } t \in [\![\sigma]\!], \qquad \lambda z^\sigma(u[t_1/x_1]...[t_n/x_n])(t) \in [\![\sigma']\!]$$

from which it follows from the definition of $[\![\sigma \to \sigma']\!]$ that

$$\lambda z^\sigma(u[t_1/x_1]...[t_n/x_n]) \in [\![\sigma \to \sigma']\!]$$

and because the terms t_i are closed,

$$\lambda z^\sigma(u)[t_1/x_1]...[t_n/x_n] \in [\![\sigma \to \sigma']\!]$$

as desired.

It suffices to consider the case in which e has the form $\lambda z^\sigma(u)$ and the variable z^σ does *not* occur free in u. In this case (using the same notation), by inductive hypothesis we have

$$u[t_1/x_1]...[t_n/x_n] \in [\![\sigma']\!].$$

Using Lemma 2(i) (and invoking the fact proved in Lemma 3 that all the elements of $[\![\sigma]\!]$ are strongly normalizing), we have as before that

$$\text{for all } t \in [\![\sigma]\!], \qquad \lambda z^\sigma(u[t_1/x_1]...[t_n/x_n])(t) \in [\![\sigma']\!].$$

from which it again follows that

$$\lambda z^\sigma(u)[t_1/x_1]...[t_n/x_n] \in [\![\sigma \to \sigma']\!]$$

as desired. □.

It follows from Lemma 4 that if e is a *closed* λ-term of type σ, then $e \in [\![\sigma]\!]$. From Lemma 3 it then follows that e is strongly normalizing. So all closed λ-terms are strongly normalizing. Let e be an open λ-term with free variables $x_1^{\sigma_1}, x_2^{\sigma_2}, ..., x_n^{\sigma_n}$. Then the λ-term $\lambda x_1^{\sigma_1} x_2^{\sigma_2} ... x_n^{\sigma_n}(e)$ is closed and therefore strongly normalizing. But because every subterm of a strongly normalizing term is strongly normalizing, it follows that e is strongly normalizing. So all λ-terms are strongly normalizing. □

Theorem 1.32: The Church-Rosser Theorem for β-reductions in λ_0^{type}.

In the typed λ-calculus λ_0^{type}, if $X \twoheadrightarrow_\beta Y_1$ and $X \twoheadrightarrow_\beta Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$.

Theorem 1.34: The Church-Rosser Theorem for $\beta\eta$-reductions in λ_0^{type}.

In the typed λ-calculus λ_0, if $X \twoheadrightarrow_{\beta\eta} Y_1$ and $X \twoheadrightarrow_{\beta\eta} Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_{\beta\eta} Z$ and $Y_2 \twoheadrightarrow_{\beta\eta} Z$.

Proof of Theorems 1.32 and 1.34.

To prove Theorems 1.32 and 1.34, it suffices to use the proofs of Theorems 1.13 and 1.20 and note that nothing more than notational alterations are required in the typed case. In particular, every term used in these proofs can be typed appropriately if required. □

Theorem 1.43: Soundness and Completeness of $\mathbf{TR_0}$.

Suppose E is an untyped λ_0-term whose free variables are $v_1, ..., v_n$, and let $\tau_1, ..., \tau_n$ be types. Then the following are equivalent:

(i) It is possible to assign simple types (i.e., types of λ_0^{type}) to all the variables of E in such a way that the free variables $v_1, ..., v_n$ of E are assigned types $\tau_1, ..., \tau_n$, and E itself has type X.

(ii) There is a proof of $\Gamma \vdash E : X$ in $\mathbf{TR_0}$, where the context Γ includes the variable declarations $v_1 : \tau_1,\ v_2 : \tau_2, ...,\ v_n : \tau_n$.

Proof of Theorem 1.43.

First we prove **(i)** $\rightarrow$ **(ii)**. The proof is by induction on the construction of the λ-term E. Let $\overline{\Gamma}$ be the context $v_1 : \tau_1, v_2 : \tau_2, ..., v_n : \tau_n$. We assume without loss of generality that E obeys the Barendregt variable convention. For suppose E is a term that does not obey the Barendregt variable convention, and let E' be a term α-equivalent to E that does obey the Barendregt variable convention. Because E and E' are α-equivalent, if E satisfies **(i)**, then so does E' . Thus, if **(i)** $\rightarrow$ **(ii)** holds for E', we have a proof of $\Gamma \vdash E' : X$ for some Γ with $\overline{\Gamma} \subseteq \Gamma$. Silent replacement of α-equivalent terms in the final step of this proof then gives $\Gamma \vdash E : X$, and so

(ii) holds, as desired.

For the base case of the induction, suppose that E is a variable v. Then E can be assigned any type X whatsoever. Because there is a proof of $v : X \vdash v : X$ for any X, **(i)** $\rightarrow$ **(ii)** holds in this case.

Suppose next that E has the form $E_1(E_2)$. Let us suppose that consistent with the variable declarations in $\overline{\Gamma}$, it is possible to type the remaining variables of E so that E is assigned the type X. In this situation, we must have that for some type Y, E_1 is assigned type $Y \rightarrow X$ and E_2 is assigned type Y. Then by inductive hypothesis we have

$$\Gamma_1 \vdash E_1 : Y \rightarrow X \quad \text{and} \quad \Gamma_2 \vdash E_2 : Y \tag{27}$$

for contexts Γ_1 and Γ_2 such that for every free variable v_i of E_1, the context Γ_1 contains $v_i : \tau_i$, and for every free variable v_i of E_2, the context Γ_2 contains $v_i : \tau_i$. By Theorem 1.41, we may in fact assume that Γ_1 and Γ_2 consist *only* of typing declarations for the the free variables of E_1 and E_2 respectively (and thus $\Gamma_1, \Gamma_2 \subseteq \overline{\Gamma}$), and that the only other variable declarations that appear anywhere else in the typing derivations (27) are typing declarations for bound variables of E_1 and E_2. Because E_1 and E_2 obey the Barendregt variable convention, the bound variables of E_1 and E_2 are distinct from the free variables of $E_1(E_2)$, and so these additional variable declarations all involve variables distinct from $v_1, ..., v_n$.

By adding additional (redundant) variable declarations to the contexts in the typing derivations (27), we may then produce typing derivations

$$\overline{\Gamma} \vdash E_1 : Y \rightarrow X \quad \text{and} \quad \overline{\Gamma} \vdash E_2 : Y. \tag{28}$$

(Note that we must invoke the fact that any additional variable declarations in (27) that occur above the final conclusion involve variables other than $v_1, ..., v_n$. If this were not true there would be a chance that adding redundant variable declarations to the typing constructions in (27) would produce a context with a variable declared in two different ways, which would be ungrammatical. By working only with terms that obey the Barendregt variable convention we avoid this possibility.)

From (28), we may infer $\overline{\Gamma} \vdash E : X$ using the (App) rule. So **(i)** $\rightarrow$ **(ii)** holds in this case.

Suppose finally that E has the form $\lambda v(E')$ and that the free variables of $\lambda v(E')$ are $v_1, ..., v_n$, so that the free variables of E' are either $v, v_1, ..., v_n$ or $v_1, ..., v_n$, depending on whether v explicitly appears free in E'. Suppose also that consistent with the variable declarations of $\overline{\Gamma}$, the remaining variables of $\lambda v(E')$ can be assigned types in such a way that $\lambda v(E')$ is assigned type X. In this situation, for some types Y and Z the type X will have the form $Y \rightarrow Z$, the term E' will have the type Z, and the variable v will have the type Y. Then by inductive hypothesis and Theorem 1.41 there is a typing derivation with conclusion

$$\overline{\Gamma}, v : Y \vdash E' : Z \quad \text{or} \quad \overline{\Gamma} \vdash E' : Z \tag{29}$$

depending on whether v explicitly appears free in E' or not. In the second case of (29), the variable v does not occur bound in E' and so by Theorem 1.41 we may assume that v is not declared anywhere in the typing derivation $\overline{\Gamma} \vdash E' : Z$. Thus we may freely add $v : Y$ as a (redundant) variable declaration throughout the proof of $\overline{\Gamma} \vdash E' : Z$, obtaining a typing derivation with conclusion $\overline{\Gamma}, v : Y \vdash E' : Z$; i.e., the first case of (29). So either way we have a typing derivation with conclusion $\overline{\Gamma}, v : Y \vdash E' : Z$. Applying the (Abs) rule then gives

$$\overline{\Gamma} \vdash \lambda v(E') : Y \to Z$$

as desired, and **(i)** $\to$ **(ii)** holds in this case. This completes the proof that **(i)** $\to$ **(ii)** holds in general.

Now we prove **(ii)** $\to$ **(i)**. The proof is by induction on the construction of typing derivations. (We do not need to assume anymore that any particular terms obey the Barendregt variable convention.) If $\Gamma \vdash E : X$ has the form of an initial sequent $\Delta, v : X \vdash v : X$, then it suffices to show that it is consistent with v being assigned the type X that v is assigned the type X, which is trivial. Thus **(ii)** $\to$ **(i)** holds in this case.

Suppose next that $\Gamma \vdash E_1(E_2) : X$ is the result of an (App) inference from

$$\Gamma \vdash E_1 : Y \to X \quad \text{and} \quad \Gamma \vdash E_2 : Y$$

Let $\overline{\Gamma}$ be the context $v_1 : \tau_1, ..., v_n : \tau_n$, let $\overline{\Gamma}_1$ be the subset of variable declarations in $\overline{\Gamma}$ that declare types for free variables of E_1, and $\overline{\Gamma}_2$ be the subset of variable declarations in $\overline{\Gamma}$ that declare types for free variables of E_2. (Every typing declaration in $\overline{\Gamma}$ will then be in one or both of $\overline{\Gamma}_1$ and $\overline{\Gamma}_2$, as every free variable of $E_1(E_2)$ occurs either free in E_1 or free in E_2.)

By inductive hypothesis, it then follows that consistent with the free variables of E_1 being typed in accordance with the variable declarations in $\overline{\Gamma}_1$, the term E_1 can be assigned the type $Y \to X$, and consistent with the the free variables of E_2 being typed in accordance with the variable declarations in $\overline{\Gamma}_2$, the term E_2 can be assigned the type $Y \to X$. Because both $\overline{\Gamma}_1$ and $\overline{\Gamma}_2$ are contained in $\overline{\Gamma}$, there is no conflict between the variable declarations of $\overline{\Gamma}_1$ and $\overline{\Gamma}_2$, and it follows that consistent with the free variables of $E_1(E_2)$ being typed in accordance with the variable declarations in $\overline{\Gamma}$, the term $E_1(E_2)$ can be assigned the type X, as desired.

Suppose finally that $\Gamma \vdash \lambda v(E) : X \to Y$ is the result of an (Abs) inference from

$$\Gamma, v : X \vdash E : Y \tag{30}$$

where if $v_1, ..., v_n$ are the free variables of $\lambda v(E)$, then Γ includes the typing declarations $v_1 : \tau_1, ..., v_n : \tau_n$. Again, let $\overline{\Gamma}$ be the context $v_1 : \tau_1, ..., v_n : \tau_n$.

If v occurs free in E, then the free variables of E are $v_1, ..., v_n, v$, and so from (30) and inductive hypothesis, we have that consistent with the variable declarations in $\overline{\Gamma}, v : X$ it is possible to assign E the type Y. It follows that consistent with the variable declarations in $\overline{\Gamma}$ it is possible to assign $\lambda v(E)$ the type $X \to Y$, as desired. If v does not occur free in E,

then by inductive hypothesis and (30) we have that consistent with the variable declarations in $\overline{\Gamma}$ it is possible to assign E the type Y. Again, this means that consistent with the variable declarations in $\overline{\Gamma}$ it is possible to assign $\lambda v(E)$ the type $X \to Y$, as desired. This completes the proof that **(ii)** $\to$ **(i)** holds in general.

Theorem 1.49: Soundness and Completeness of $\mathbf{TR_0^{bt}}$.

Suppose E is a bound-typed λ-term whose free variables are $v_1, ..., v_n$, and let $\tau_1, ..., \tau_n$ be types. Then the following are equivalent:

(i) If the variables $v_1, v_2, ..., v_n$ are assigned types $\tau_1, \tau_2, ..., \tau_n$ respectively, then E must be assigned type σ.

(ii) There is a proof of $\Gamma \vdash E : X$ in $\mathbf{TR_0^{bt}}$, where the context Γ includes the variable declarations $v_1 : \tau_1,\ v_2 : \tau_2, ...,\ v_n : \tau_n$.

Proof of Theorem 1.49.

The proof is largely the same as that of Theorem 1.43. The proof that **(i)** $\to$ **(ii)** is by induction on the construction of the λ-term E. Let $\overline{\Gamma}$ be the context $v_1 : \tau_1, v_2 : \tau_2, ..., v_n : \tau_n$. As in Theorem 1.43, we may assume that E obeys the Barendregt variable convention.

The base case in which E is a variable is trivial. Suppose then that E has the form $E_1(E_2)$. Let us suppose that if the free variables in E are typed in accordance with $\overline{\Gamma}$, the term E must receive type X. Because the type of E and all its subterms are uniquely determined by the typing declarations in $\overline{\Gamma}$, these typing declarations also uniquely determine the types of E_1 and E_2. Suppose then that if the free variables in E_1 and E_2 are typed in accordance with $\overline{\Gamma}$, that E_1 must be assigned the type $Y \to X$ and E_2 must be assigned type Y. Then by inductive hypothesis we have

$$\Gamma_1 \vdash E_1 : Y \to X \quad \text{and} \quad \Gamma_2 \vdash E_2 : Y$$

for contexts Γ_1 and Γ_2 such that for every free variable v_i of E_1, Γ_1 contains $v_i : \tau_i$, and for every free variable v_i of E_2, Γ_2 contains $v_i : \tau_i$. Precisely as in the proof of Theorem 1.43 we can then argue that

$$\overline{\Gamma} \vdash E_1(E_2) : X$$

So **(i)** $\to$ **(ii)** holds in this case.

Suppose finally that E has the form $\lambda v^Y(E')$ for some type Y, so that E' has free variables $v_1, ..., v_n$ and possibly v. Assume also that given the typing declarations in $\overline{\Gamma}$, $\lambda v^Y(E')$ must be assigned the type $Y \to Z$. If v occurs free in E', this means that given the typing declarations in $\overline{\Gamma}, v : Y$, the term E' must be assigned the type Z, and if v does not occur free in E', this

means that given the typing declarations in $\overline{\Gamma}$, the term E' must be assigned the type Z. So by inductive hypothesis we have

$$\overline{\Gamma}, v : Y \vdash E' : Z \quad \text{or} \quad \overline{\Gamma} \vdash E' : Z$$

The same argument as in Theorem 1.43 then shows that in either case we have

$$\overline{\Gamma} \vdash \lambda v^Y(E') : Y \to Z$$

as desired, and **(i)** $\to$ **(ii)** holds in this case. This completes the proof that **(i)** $\to$ **(ii)** holds in general.

The proof that **(ii)** $\to$ **(i)** is by induction on the construction of proofs. If $\Gamma \vdash E : X$ has the form of an initial sequent $\Delta, v : X \vdash v : X$, then the result is trivial.

Suppose then that $\Gamma \vdash E_1(E_2) : X$ is the result of an (App) inference from

$$\Gamma \vdash E_1 : Y \to X \quad \text{and} \quad \Gamma \vdash E_2 : Y$$

where $v_1, ..., v_n$ are the free variables of $E_1(E_2)$. Thus, Γ will include the typing declarations $v_1 : \tau_1, ..., v_n : \tau_n$ for some $\tau_1, ..., \tau_n$. As before, let $\overline{\Gamma}$ be the context $v_1 : \tau_1, ..., v_n : \tau_n$, let $\overline{\Gamma}_1$ be the subset of variable declarations in $\overline{\Gamma}$ for free variables of E_1, and let $\overline{\Gamma}_2$ be the subset of variable declarations in $\overline{\Gamma}$ for free variables of E_2.

By inductive hypothesis, the variable declarations in $\overline{\Gamma}_1$ entail that E_1 is assigned the type $Y \to X$, and the variable declarations in $\overline{\Gamma}_2$ entail that E_2 is assigned the type $Y \to X$. Because $\overline{\Gamma}_1$ and $\overline{\Gamma}_2$ are contained in $\overline{\Gamma}$, it follows that the variable declarations in $\overline{\Gamma}$ entail that $E_1(E_2)$ is assigned the type X, as desired.

Suppose finally that $\Gamma \vdash \lambda v^X(E') : X \to Y$ is the result of an (Abs$^{\mathbf{bt}}$) inference from

$$\Gamma, v : X \vdash E' : Y$$

where if $v_1, ..., v_n$ are the free variables of $\lambda v(E')$, then Γ includes the typing declarations $v_1 : \tau_1, ..., v_n : \tau_n$. Let $\overline{\Gamma}$ be the context $v_1 : \tau_1, ..., v_n : \tau_n$.

If v occurs free in E', then by inductive hypothesis, the variable declarations in $\overline{\Gamma}, v : X$ require E' to be assigned the type Y. This means that the variable declarations in $\overline{\Gamma}$ require $\lambda v^X(E')$ to be assigned the type $X \to Y$, as desired.

If v does not occur free in E', then by inductive hypothesis, the variable declarations in $\overline{\Gamma}$ require E' to be assigned the type Y. This also means that the variable declarations in $\overline{\Gamma}$ require $\lambda v^X(E')$ to be assigned the type $X \to Y$, as desired. This completes the proof that **(ii)** $\to$ **(i)** holds in general.

Chapter 2

Intuitionistic Propositional Logic and the Curry-Howard Correspondence.

2.1 Preamble

In this chapter, our main goal is to understand the Curry-Howard correspondence in its simplest forms. Broadly speaking, the Curry-Howard correspondence connects the notions of computation and proof - two notions which might otherwise seem quite distinct. There are many different forms of the Curry-Howard correspondence; in this chapter, we examine a simple version that connects the notion of certain sorts of computation with the notion of provability in *intuitionistic propositional logic*. In later chapters we examine more sophisticated versions of the Curry-Howard correspondence that connect richer notions of computation with richer notions of provability. We begin with a brief discussion of the main ideas behind intuitionistic logic.

2.2 Intuitionistic Logic: Motivation

One of the guiding ideas behind intuitionistic logic is that our reasoning ought to be *constructive*. So for example, according to constructive reasoning the only situation in which we are entitled to assert a disjunction $A \vee B$ would be a situation in which either we are entitled to assert A or we are entitled to assert B. To put this in terms of knowledge, we could say that we only know a disjunctive claim $A \vee B$ if we know A or know B. Intuitionistic logic makes similar claims about existentially quantified sentences $(\exists x)\phi(x)$. The only situation in which we are entitled to assert such a sentence is a situation in which we can name (or construct) an object o for which $\phi(o)$. To put it in terms of knowledge, we only know a claim of the form $(\exists x)\phi(x)$ if we know that $\phi(o)$ for some particular o.

Intuitionistic logic places different demands on us than classical logic. This point is illustrated nicely by the following classic example:

Theorem 2.1

There are irrational numbers x, y such that x^y is rational.

Proof

Consider the number $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}}$. Using the fact that $(a^b)^c = a^{(b \cdot c)}$, this equals 2. So if $\sqrt{2}^{\sqrt{2}}$ is irrational, then $x = \sqrt{2}^{\sqrt{2}}$, $y = \sqrt{2}$ gives an example of irrational numbers x and y such that x^y is rational. On the other hand, if $\sqrt{2}^{\sqrt{2}}$ is rational, then $x = \sqrt{2}$, $y = \sqrt{2}$ gives an example of irrational numbers x and y such that x^y is rational. So either way, there are irrational numbers x, y such that x^y is rational.

From the point of view of classical logic, this proof is completely persuasive. From the point of view of intuitionistic logic, it is not a valid proof at all.The theorem being proved is of course an existential claim; it states that *there are* irrational numbers x, y such that x^y is rational. But at the end of the proof we are still not in a position to name specific irrational numbers x and y for which x^y is rational. The intuitionistic logician is bothered by this - what could it mean to have proved an existential claim if a confirming instance has not been unambiguously specified? If one thinks that to prove a disjunction one must prove one of the disjuncts, or if one thinks that to prove an existential claim one must construct a specific confirming instance, then the above proof is completely unpersuasive.

Intuitionistic logic is an attempt to restrict our techniques of reasoning in such a way that whenever we can prove a disjunction $A \vee B$, then we know which of A or B obtains, and whenever we can prove an existential claim $(\exists x)\phi(x)$, then we can name an object o for which we have $\phi(o)$. It is perhaps surprising that one can quite formally and rigorously define a system of reasoning that conforms to these principles, and can in fact do so merely by making some small modifications to the usual rules of classical logic. Exploring how this is so will be one of the main themes of this chapter.

Although intuitionstic logic has a propositional form (in which there are no quantifiers) and a predicate form (which includes the usual quantifiers), in this chapter we will restrict our attention to propositional intuitionistic logic, turning to predicate intruitionistic logic in a later chapter.

First, we have the following definition

Definition 2.2:

In propositional intuitionistic logic, all formulae are constructed from atomic formulae and the the 0-ary connective $\bot$ (sometimes called 'the false' or 'falsum') using the binary connectives $\rightarrow, \vee$ and $\&$.

In what follows, $\neg\phi$ will be an abbreviation for $\phi \rightarrow \bot$, $\phi \leftrightarrow \psi$ will be an abbreviation for $(\phi \rightarrow \psi)\&(\psi \rightarrow \phi)$, and $\top$ (sometimes called 'the true' or 'verum') will be an abbreviation for $\bot \rightarrow \bot$.

So for example

$(\neg A) \vee B$	is an abbreviation of	$(A \rightarrow \bot) \vee B$
$(\neg A) \leftrightarrow B$	is an abbreviation of	$((A \rightarrow \bot) \rightarrow B)\ \&\ (B \rightarrow (A \rightarrow \bot))$
$(\neg A) \vee \neg\top$	is an abbreviation of	$(A \rightarrow \bot) \vee ((\bot \rightarrow \bot) \rightarrow \bot)$

Next we turn to a discussion of the connectives themselves. The logical constant $\bot$ and the connectives $\rightarrow, \vee$ and $\&$ are the basic connectives of propositional intuitionistic logic. How are we to analyse their meaning? In the usual presentations of classical propositional logic, much emphasis is placed on *truth tables*. In particular, the connectives are often regarded as defined by the way they transform truth values. For example, consider the truth table for $\&$:

A	B	A&B
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

This table presents $\&$ as if it were a function taking a pair of truth values as inputs (given in the first two columns), and outputting a truth value (given in the third column). From the point of classical logic, one can even think of this as something like the *definition* of the connective $\&$ – it is the binary connective that transforms truth values in the way shown in this table.

Intuitionistic logic takes a different point of view in analysing the meaning of the connectives. Rather than analysing the connectives in terms of truth values, intuitionistic logic analyzes them in terms of *proofs*, or even more broadly in terms of *constructions*. The basic ideas intuitionistic logic begins with are then as follows:

Definition 2.3: The BHK Conditions

BHK1 : To prove $A\ \&\ B$ is to prove A and to prove B.
BHK2 : To prove $A \vee B$ is to prove A or to prove B.

BHK3 : To prove $A \to B$ is to construct a way of transforming a proof of A into a proof of B.
BHK4 : There is no proof of $\bot$.

These conditions are described here somewhat informally, but we will later consider much more precise versions of them. They are sometimes described as the **BHK** conditions on the connectives in honor of Brouwer, Heyting, and Kolmogorov who first developed many of the main ideas of intuitionistic logic.

Consider **BHK1**. This does not just say that one way of proving $A\&B$ is to prove A and prove B; it rather makes the stronger claim that *what it is* to prove $A\&B$ is to prove A and to prove B. Moreover, this is taken as our basic analysis of $\&$. That is to say, the *meaning* of $\&$ is constituted by the fact that it is the connective having the property that proving $A\&B$ just amounts to proving A and proving B.

Similarly, according to **BHK2**, the meaning of $\vee$ is constituted by the fact that it is the connective having the property that proving $A \vee B$ amounts to either a proof of A or a proof of B.

The idea behind **BHK3** is that to prove $A \to B$ is to have some method for transforming a proof of A into a proof of B; this is sometimes expressed by the idea that a proof of $A \to B$ is a *function* that transforms a proof of A into a proof of B. This is usually more specifically taken to mean that there is some kind of *mechanical procedure* for transforming a proof of A into a proof of B. (At this point, we can be somewhat open as to what counts as a 'mechanical procedure'.) The connective $\to$ can even be regarded as being defined by the fact that it is the connective having the property that proving $A \to B$ amounts to specifying a function that takes a proof of A into a proof of B.

The claim **BHK4** is perhaps more modest in character than **BHK1-3**. Presumably, we do *not* want to say that $\bot$ is characterized by the fact that there is no proof of $\bot$, because there are sentences other than $\bot$ which are not provable. While it might be argued that **BHK4** captures *part* of the meaning of $\bot$, it certainly does not capture all of it. Nevertheless, the claim made in **BHK4** is a true one, and it will suffice in what follows to regard **BHK4** as a basic truth about $\bot$.

The way of thinking of the connectives captured in **BHK1-4** is known as the **BHK** (Brouwer-Heyting-Kolmogorov) interpretation of intuitionistic logic. Note that according to this interpretation, because we regard $\neg A$ as an abbreviation of $A \to \bot$, proving $\neg A$ amounts to giving a procedure that takes a proof of A into a proof of $\bot$ (i.e., a proof of a contradiction.)

In **BHK1-4**, the main non-classical idea lies in **BHK2** – that is, in the claim that proving $A \vee B$ amounts to proving A or proving B. The classical logician could in fact happily accept **BHK1**, **BHK3**, and **BHK4**, but would emphatically reject **BHK2**. This is because in classical logic, one can have a proof of $A \vee B$ without having either a proof of A or a proof of B. We have seen this already in our discussion of Theorem 2.1. For an even simpler example, consider the claim

Even: There are an even number of electrons in the universe.

Using classical logic, we can easily prove

Even $\vee$ $\neg$**Even**,

as this is a tautology. This however does not require that we have a proof of **Even** or a proof of ¬**Even**. Only scientific investigation can determine whether **Even** is true or false; logic on its own gives us neither a proof of **Even** nor a proof of ¬**Even**. The claim that proving $A \vee B$ amounts to proving A or proving B is therefore a departure from classical logic. In effect, it imposes much stricter requirements on proving a disjunction $A \vee B$ than are imposed by classical logic. According to the **BHK** conditions, we do *not* have a proof of **Even** $\vee$ ¬**Even**, and perhaps never will.

We have not yet spelt out any sort of formal system that captures the ideas of **BHK1-4**. However, even without doing so, we can see somewhat informally that various sentences should be theorems of intuitionistic logic. For example, consider the sentence $(p\&q) \rightarrow p$. This is a classical tautology. Should we expect this to be intuitionistically provable? According to **BHK3**, a proof of $(p\&q) \rightarrow p$ would be a procedure that transforms a proof of $p\&q$ into a proof of p. Given that according to **BHK1** a proof of a $p\&q$ is just a proof of p and a proof of q, an intuitionistic proof of $(p\&q) \rightarrow p$ would just be a function that takes a proof of p and a proof of q and returns a proof of p. Such a function, of course, exists - given a proof of p and a proof of q, just forget about the proof of q. One then has a proof of p. Thus, $(p\&q) \rightarrow p$ should be intuitionistically provable.

Consider the sentence $\bot \rightarrow p$. This too is a classical tautology – anything follows from $\bot$. Should we expect this to be intuitionistically provable? For $\bot \rightarrow p$ to be provable would be to have a way of transforming a proof of $\bot$ into a proof of p. Because there are no proofs of $\bot$, we think of ourselves as 'vacuously' being able to transform a proof of $\bot$ into a proof of p – that is to say, any transformational procedure may be said to take a proof of $\bot$ into a proof of p, because there are no proofs of $\bot$. So this should be intuitionistically provable.

Consider now the classically tautology $p \rightarrow (q \rightarrow p)$. Should we expect this to be intuitionistically provable? The question we must ask is whether we have a way of transforming a proof of p into way of transforming a proof of q into a proof of p. Indeed, we do. Given a proof Π of p, we have a way of transforming a proof Σ of q into a proof of p as follows – erase Σ, and write down Π. This transforms any proof of q into a proof of p. Thus, we should expect $p \rightarrow (q \rightarrow p)$ to be intuitionistically provable.

Likewise, you should be able to convince yourself that $p \rightarrow \neg\neg p$ ought to be intutionistically provable. (This is included in the exercises.)

There are of course classical tautologies that we would *not* expect to be intutionistically provable. We have already considered the sentence $p \vee \neg p$. For this to be provable is for either p or $\neg p$ to be provable. Of course, it is not the case that for every sentence p, we have that either p or $\neg p$ is provable using pure logic. So we should expect that $p \vee \neg p$ will *not* be intutionistically provable in general. In this sense, the law of the excluded middle fails for intuitionistic logic, as a result of **BHK2**. You should also try to convince yourself that there are no good reasons to expect $\neg\neg p \rightarrow p$ be intuitionistically provable. Perhaps a little more surprisingly, you should be able to convince yourself that there are no good reasons to expect *Peirce's Law* $((p \rightarrow q) \rightarrow p) \rightarrow p$ to be intuitionstically provable, even though this is a purely implicational classical tautology and does not involve negation or disjunction in any way.

Of course, none of the arguments here suggesting that something should or should not be intuitionistically provable are completely rigorous. They are simply argument sketches. Armed

with further material, we will later be able to replace such argument sketches with completely rigorous arguments. Arguments sketches of the sort we have considered here are nevertheless suggestive and have played an important historical role in the development of intuitionistic logic.

Exercises for Section 2.2

1. Using the **BHK** interpretation, argue informally that the following should be intuitionistically provable. It suffices to give informal arguments of the sort given in this section of the text.

(a) $p \to (p \vee q)$
(b) $p \to \neg\neg p$
(c) $(p \to q) \to (\neg q \to \neg p)$
(d) $\neg\neg\neg p \to \neg p$

2. Using the **BHK** interpretation, argue informally that the following should not be intuitionistically provable. It suffices to describe how naive ways of trying to prove them using the **BHK** rules fail.

(a) $\neg\neg p \to p$
(b) $(\neg q \to \neg p) \to (p \to q)$
(c) $((p \to q) \to p) \to p)$

2.3 Natural Deduction for Propositional Intuitionistic Logic

We now set up a formal proof system that captures the informal notion of intuitionistic provability developed in the last section. It will not be clear until somewhat later in this chapter why or how the proof system presented in this section captures the main ideas behind the **BHK** interpretation. In particular, only once we understand the Curry-Howard correspondence and proof normalization will we be able to clearly see the sense in which this proof system captures the **BHK** rules. Until then, the reader should just focus on the details of the proof system itself.

In our proof system, each step in the proof will consist of a sequent of the form

$$s_1, s_2, ..., s_n \vdash t$$

where $s_1, s_2, ..., s_n, t$ are fomulae. For brevity, we will typically write this as

$$\Delta \vdash t$$

where Δ is the unordered list of sentences $s_1, s_2, ..., s_n$. In Δ, we will allow elements to be repeated. Intuitively, the sequent $\Delta \vdash t$ says that t is derivable from the sentences in Δ. We will write

$$\vdash s$$

as an abbreviation of

$$\varnothing \vdash s,$$

where $\varnothing$ is the empty list. To say $\vdash s$ is then to say that s is provable outright.

A proof will consist of a 'tree' of such sequents. The leaves of the trees will be located at the top, and will correspond to our logical axioms. The statement being proven will lie at the base of the tree. Often we are interested in proving a statement of the form $\vdash s$, but we will allow more general assertions of the form $\Delta \vdash s$ to be the conclusion of a proof and lie at the base of the tree. We will get from one level of a tree to the level below by performing a rule of inference. Examples will be provided shortly.

We specify the logical axioms and rules of inference of our system, which we call **NJ**.

Definition 2.4: The Intuitionistic Propositional Calculus NJ.

For any formulae ϕ, ψ, θ and any finite unordered list of formulae Δ, we have

Logical Axioms:

$$\Gamma, \phi \vdash \phi$$

Rules of Inference:

$$\frac{\Delta, \phi \vdash \psi}{\Delta \vdash \phi \to \psi} (\to \text{I}) \qquad \frac{\Delta \vdash \phi \to \psi \qquad \Delta \vdash \phi}{\Delta \vdash \psi} (\to \text{E})$$

(major premise: $\Delta \vdash \phi \to \psi$)

$$\frac{\Delta \vdash \phi \qquad \Delta \vdash \psi}{\Delta \vdash \phi \& \psi} (\& \text{ I}) \qquad \frac{\Delta \vdash \phi \& \psi}{\Delta \vdash \phi} (\& \text{ E}) \qquad \frac{\Delta \vdash \phi \& \psi}{\Delta \vdash \psi} (\& \text{ E})$$

$$\frac{\Delta \vdash \phi}{\Delta \vdash \phi \vee \psi} (\vee \text{ I}) \qquad \frac{\Delta \vdash \psi}{\Delta \vdash \phi \vee \psi} (\vee \text{ I})$$

$$\frac{\Delta \vdash \phi \vee \psi \qquad \Delta, \phi \vdash \theta \qquad \Delta, \psi \vdash \theta}{\Delta \vdash \theta} (\vee \text{ E})$$

(major premise: $\Delta \vdash \phi \vee \psi$)

$$\frac{\Delta \vdash \bot}{\Delta \vdash \phi} (\bot \text{ E})$$

In this system when something like Δ, ϕ appears on the left hand side of a $\vdash$ symbol, we understand Δ, ϕ as the unordered list of sentences obtained by adding ϕ to the unordered list of sentences Δ.

In the case of the ($\to$E) rule, we say that the premise $\Delta \vdash \phi \to \psi$ is the *major* premise, and the remaining premise is *minor*, and in the case of the ($\vee$E) rule, we say that the premise $\Delta \vdash \phi \vee \psi$ is the major premise, and the remaining premises minor. For the remaining rules, we will not need to classify premises as major or minor.

Note that the rules and logical axioms of **NJ** are all classically acceptable. Unlike classical logic, however, **NJ** does *not* have the following rule:

$$\frac{\Delta, \neg\phi \vdash \ \bot}{\Delta \vdash \ \phi} \quad (\times)$$

This rule is of course classically valid, but we do not include it in our system, and it is not derivable in any way from the rules of **NJ**.

It is not obvious that this moderate restriction of classical logic captures the ideas behind the **BHK** interpretation, but we will see quite rigorously by the end of the chapter that it does. Traditional treatments of intuitionistic logic develop a semantics for intuitionistic logic (typically using the machinery of 'Kripke frames'), and then prove a soundness and completeness theorem relative to this semantics, in analogy with the familiar soundness and completeness theorems of classical logic. (See [7] and chapter 2 of [10] for examples of this.) We will not take this route. Rather than thinking of intuitionisitic proofs as arguments that preserve truth in Kripke frames, we will end up thinking of intuitionistic proofs as computer programs or algorithms (given by typed λ-terms) that obey the **BHK** rules.

As simple examples of proofs in intuitionistic logic, here are intuitionistic proofs of $p \to (q \to p)$ and $p \to \neg\neg p$, recalling that this later sentence is an abbreviation for $p \to ((p \to \bot) \to \bot)$:

$$\dfrac{\dfrac{p, q \vdash \ p}{p \vdash \ q \to p}\,(\to \text{I})}{\vdash \ p \to (q \to p)}\,(\to \text{I})$$

$$\dfrac{\dfrac{\dfrac{p, p \to \bot \vdash \ p \qquad p, p \to \bot \vdash \ p \to \bot}{p, p \to \bot \vdash \ \bot}\,(\to \text{E})}{p \vdash \ (p \to \bot) \to \bot}\,(\to \text{I})}{\vdash \ p \to ((p \to \bot) \to \bot)}\,(\to \text{I})$$

The classical principle that everything follows from a contradiction also holds in intuitionistic logic:

$$\dfrac{\dfrac{\dfrac{\dfrac{p\&\neg p \vdash \ p\&\neg p}{p\&\neg p \vdash \ p}\,(\&\text{ E}) \qquad \dfrac{p\&\neg p \vdash \ p\&\neg p}{p\&\neg p \vdash \ \neg p}\,(\&\text{ E})}{p\&\neg p \vdash \ \bot}\,(\to \text{E})}{p\&\neg p \vdash \ q}\,(\bot \text{ E})}{\vdash \ (p\&\neg p) \to q}\,(\to \text{I})$$

For a more complicated argument, note that using the proofs

$$\dfrac{(\neg p) \vee (\neg q),\ p\&q,\ \neg p \vdash \ \neg p \qquad \dfrac{(\neg p) \vee (\neg q),\ p\&q,\ \neg p \vdash \ p\&q}{(\neg p) \vee (\neg q),\ p\&q,\ \neg p \vdash \ p}\,(\&\text{ I})}{(\neg p) \vee (\neg q),\ p\&q,\ \neg p \vdash \ \bot}\,(\to \text{E})$$

and

$$\dfrac{\dfrac{(\neg p)\vee(\neg q),\ p\&q,\ \neg q\ \vdash\ \neg q \qquad \dfrac{(\neg p)\vee(\neg q),\ p\&q,\ \neg q\ \vdash\ p\&q}{(\neg p)\vee(\neg q),\ p\&q,\ \neg q\ \vdash\ q}\,(\&\text{ I})}{(\neg p)\vee(\neg q),\ p\&q,\ \neg q\ \vdash\ \bot}\,(\to\text{ E})$$

we can prove $((\neg p)\vee(\neg q))\to\neg(p\&q)$:

$$\dfrac{\dfrac{\dfrac{(\neg p)\vee(\neg q),\ p\&q\ \vdash\ (\neg p)\vee(\neg q) \qquad \begin{matrix}(\neg p)\vee(\neg q),\ p\&q,\ \neg p\vdash\bot\\ \vdots\end{matrix} \qquad \begin{matrix}(\neg p)\vee(\neg q),\ p\&q,\ \neg q\vdash\bot\\ \vdots\end{matrix}}{(\neg p)\vee(\neg q),\ p\&q\vdash\bot}\,(\vee\text{ E})}{(\neg p)\vee(\neg q)\vdash\neg(p\&q)}\,(\to\text{ I})}{\vdash((\neg p)\vee(\neg q))\to\neg(p\&q)}\,(\to\text{ I})$$

We have called the proof system just presented **NJ**. The system **NJ**($\to$) denotes the 'implicational' fragment of this system - that is, the system with the same logical axioms, but with only the rules ($\to$ I) and ($\to$ E).

Definition 2.5: The System NJ($\to$).

Logical Axioms:

$$\Gamma,\phi\vdash\phi$$

Rules of Inference:

$$\dfrac{\Delta,\phi\ \vdash\ \psi}{\Delta\ \vdash\ \phi\to\psi}\,(\to\text{ I}) \qquad \dfrac{\Delta\ \vdash\ \phi\to\psi \qquad \Delta\ \vdash\ \phi}{\Delta\ \vdash\ \psi}\,(\to\text{ E})$$

(major premise: $\Delta\vdash\phi\to\psi$)

Of the many familiar theorems from classical logic, **NJ** proves some but not others. Most notably, it does not prove the law of the excluded middle $p\vee\neg p$ in general. (Later in this chapter, we will rigorously *prove* that **NJ** does not prove the law of the excluded middle!) And although we have seen that **NJ** can prove $p\to\neg\neg p$, it cannot prove $\neg\neg p\to p$ in general. (Interestingly, it can prove $\neg\neg\neg p\to\neg p$.) Not all the familiar deMorgan's laws hold either. So although the following

$(\neg p\&\neg q)\to\neg(p\vee q)$
$\neg(p\vee q)\to(\neg p\&\neg q)$
$(\neg p\vee\neg q)\to\neg(p\&q)$.

are all provable in **NJ**,

$\neg(p\&q)\to(\neg p\vee\neg q)$

is not provable in **NJ**. However, all the familiar distributivity rules relating conjunction and disjunction

$(p\vee(q\&r))\to((p\vee q)\&(p\vee r))$

$((p \vee q)\&(p \vee r)) \rightarrow (p \vee (q\&r))$
$((p\&q) \vee (p\&r)) \rightarrow (p\&(q \vee r))$
$(p\&(q \vee r)) \rightarrow ((p\&q) \vee (p\&r))$

are provable in **NJ**. In the exercises you will verify many of these facts about what is provable in **NJ**. Later in this chapter we will be in a position to show rigorously that certain sentences are *not* provable in **NJ**.

Exercises for Section 2.3

1. Prove the following sentences in **NJ**:
(a) $p \rightarrow (p \vee q)$
(b) $\neg(p \vee q) \rightarrow \neg p$
(c) $\neg(p \vee q) \rightarrow (\neg p)\&(\neg q)$
(d) $(\neg p)\&(\neg q) \rightarrow \neg(p \vee q)$
(e) $(\neg p) \vee (\neg q) \rightarrow \neg(p\&q)$
(f) $p \rightarrow \neg\neg p$
(g) $(p \vee (q\&r)) \rightarrow ((p \vee q)\&(p \vee r))$
(h) $((p \vee q)\&(p \vee r)) \rightarrow (p \vee (q\&r))$
(i) $((p\&q) \vee (p\&r)) \rightarrow (p\&(q \vee r))$
(j) $(p\&(q \vee r)) \rightarrow ((p\&q) \vee (p\&r))$
(k) $\neg\neg(p \vee \neg p)$
(l) $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$
(m) $\neg\neg\neg p \rightarrow \neg p$

2. Show that a formula X is provable in classical propositional logic iff $\neg\neg X$ is provable in **NJ** as follows. (This result is sometimes called *Glivenko's Theorem*.) For our purposes, we take classical propositional logic to be **NJ** with additional logical axioms of the form

$$\Delta \vdash X \vee \neg X$$

permitted for any sentence X and unordered list of sentences Δ.
(a) Prove that if $\vdash \neg\neg X$ is provable in **NJ**, then $\vdash X$ is provable in classical propositional logic. (Hint: prove and use the fact that $\neg\neg X \vdash X$ is derivable in classical propositional logic, using ($\vee$ E).)
(b) Show that if $\vdash X$ is provable in classical propositional logic, then for some formulae $p_1, ..., p_n$,

$$p_1 \vee \neg p_1, p_2 \vee \neg p_2, ..., p_n \vee \neg p_n \vdash X$$

is provable in **NJ**. Infer from this that

$$p_1 \vee \neg p_1, p_2 \vee \neg p_2, ..., p_n \vee \neg p_n, \neg X \vdash \bot$$

is provable in **NJ**.
(c) From this, show that if $\vdash\ X$ is provable in classical propositional logic, then $\vdash\ \neg\neg X$ is provable in **NJ**. (Hint: use the fact that $\vdash \neg\neg(p \vee \neg p)$ is provable in **NJ**.)

2.4 Some Abbreviations

It will help to introduce some conventions to simplify the presentation of proofs. The following rule, sometimes called weakening, allows us to introduce an additional premise on the left of a sequent

$$\frac{\Delta\ \vdash\ B}{\Delta, A\ \vdash\ B}\ (\mathrm{W})$$

In fact, anything we can prove with this rule we can prove without it. For example, consider the proof

$$\frac{\dfrac{B, B \to A\ \vdash\ B \qquad B, B \to A\ \vdash\ B \to A}{B, B \to A\ \vdash\ A}\ (\to\mathrm{E})}{B, B \to A, C\ \vdash\ A}\ (\mathrm{W}) \tag{31}$$

In the final step of this proof, we introduce C on the left by a weakening. However, we could in fact have just included C in the left hand side of each sequent all along, thereby eliminating the need for a weakening, as follows:

$$\frac{B, B \to A, C\ \vdash\ B \qquad B, B \to A, C\ \vdash\ B \to A}{B, B \to A, C\ \vdash\ A}\ (\to\mathrm{E}) \tag{32}$$

Although weakening adds an extra step to the proof, it allows us to introduce a formula on the left hand side of a sequent only when it is needed, rather than being forced to carry it around from the beginning of a proof. Weakening can thus make our proofs more compact, and so we allow ourselves to use this rule. However, we do not regard weakening as one of the basic rules of **NJ**. Instead, we think of (31) as an 'abbreviation' of (32). So (31) is not technically a proof of **NJ**, but rather an abbreviation of proof (32), which is a proof of **NJ**.

It will be useful to introduce several other 'abbreviations' of this sort. Consider the rule ($\to$ E)

$$\frac{\Delta\ \vdash\ A \to B \qquad \Delta\ \vdash\ A}{\Delta\ \vdash\ B}\ (\to \mathrm{E})$$

In this rule the context Δ on the left hand side of each sequent is required to be the same. An alternative would be to not require this, but instead require that the context in the conclusion be the concatenation of the contexts in the premises. We call this rule ($\to$ E′)

$$\frac{\Delta\ \vdash\ A \to B \qquad \Delta'\ \vdash\ A}{\Delta, \Delta'\ \vdash\ B}\ (\to \mathrm{E}')$$

where by Δ, Δ' we simply mean the unordered list consisting of Δ conjoined with Δ'. If Δ and Δ' have an element X in common, then X may of course appear multiple times in Δ, Δ' . We allow

the option of eliminating as many such 'duplicate' formulae as we want on the left hand side of the sequent when applying this rule. So for example, each of the following are valid instances of ($\rightarrow$ E')

$$\frac{A, A \rightarrow B, C_1 \vdash A \rightarrow B \qquad A, A \rightarrow B, C_2 \vdash A}{A, A, A \rightarrow B, A \rightarrow B, C_1, C_2 \vdash B} (\rightarrow \text{E}')$$

$$\frac{A, A \rightarrow B, C_1 \vdash A \rightarrow B \qquad A, A \rightarrow B, C_2 \vdash A}{A, A \rightarrow B, A \rightarrow B, C_1, C_2 \vdash B} (\rightarrow \text{E}') \tag{33}$$

$$\frac{A, A \rightarrow B, C_1 \vdash A \rightarrow B \qquad A, A \rightarrow B, C_2 \vdash A}{A, A \rightarrow B, C_1, C_2 \vdash B} (\rightarrow \text{E}')$$

Like the rule of weakening, we do not need to think of ($\rightarrow$ E') as a new basic rule of **NJ**. In fact, anything we can prove with ($\rightarrow$ E') can be proven just with ($\rightarrow$ E) by adding needed formulae to the left hand side of sequents above the ($\rightarrow$ E), in much the same way that we did for weakening. For example, the middle proof (33) can be regarded as an abbreviation for

$$\frac{A, A \rightarrow B, A \rightarrow B, C_1, C_2 \vdash A \rightarrow B \qquad A, A \rightarrow B, A \rightarrow B, C_1, C_2 \vdash A}{A, A \rightarrow B, A \rightarrow B, C_1, C_2 \vdash B} (\rightarrow \text{E}) \tag{34}$$

In this way, we regard proofs using ($\rightarrow$ E') as simply abbreviations of proofs using the conventional ($\rightarrow$ E) rule. Comparing (33) and (34) we see that the ($\rightarrow$ E') rule can simplify our proofs, and for this reason we use it.

Using these new abbreviations, we can compress (31) even further as follows

$$\frac{\dfrac{B \vdash B \qquad B \rightarrow A \vdash B \rightarrow A}{B, B \rightarrow A \vdash A} (\rightarrow\text{E}')}{B, B \rightarrow A, C \vdash A} (\text{W}) \tag{35}$$

We allow similar 'abbreviations' for all our rules which take more than one sequent as premise – so not just ($\rightarrow$E), but also ($\&$I') and ($\vee$E').

The material of this section is captured by the following theorem

Theorem 2.6: The Admissibility of (W), ($\rightarrow$E'), ($\&$I') and ($\vee$E').

In each of the rules

$$\frac{\Delta \vdash B}{\Delta, A \vdash B} (\text{W}) \qquad \frac{\Delta \vdash A \rightarrow B \qquad \Delta' \vdash A}{\Delta, \Delta' \vdash B} (\rightarrow \text{E}')$$

$$\frac{\Delta \vdash A \qquad \Delta' \vdash B}{\Delta, \Delta' \vdash A \& B} (\& \text{ I}) \qquad \frac{\Delta \vdash A \vee B \qquad \Delta', A \vdash C \qquad \Delta'', B \vdash C}{\Delta, \Delta', \Delta'' \vdash C} (\vee \text{ E})$$

if the premise(s) are provable in **NJ**, then so is the conclusion. In the conclusions of the rules ($\rightarrow$E'), ($\&$I') and ($\vee$E'), we allow the optional elimination of 'duplicate' copies of formulae

created when the contexts Δ, Δ', ... partially overlap.

This theorem tells us that even though (W), ($\to$E'), ($\&$I') and ($\vee$E') are not rules of **NJ**, we may treat them as if they were. The proof is a straightforward induction on the construction of proofs, and a representative case is left to the exercises.

Exercises for Section 2.4

1. To what proofs using only the rules of **NJ** do the following proofs correspond?

(i)
$$\dfrac{C \vdash C \qquad \dfrac{B \vdash B}{A, B \vdash B}\,(\text{W})}{A, B, C \vdash B \& C}\,(\&\text{I}')$$

(ii)
$$\dfrac{A \vee B \vdash A \vee B \qquad B \vdash B \qquad \dfrac{\dfrac{A \vdash A \qquad \neg A \vdash \neg A}{A, \neg A \vdash \bot}\,(\to\text{E}')}{A, \neg A \vdash B}\,(\bot\text{E})}{A \vee B, \neg A \vdash B}\,(\vee\text{E}')$$

2. By using the rules (W), ($\to$E'), ($\&$I') and ($\vee$E') wherever possible, simplify the following proofs

(i)
$$\dfrac{\dfrac{\dfrac{A, A \to \bot \vdash A \qquad A, A \to \bot \vdash A \to \bot}{A, A \to \bot \vdash \bot}\,(\to\text{E})}{A \vdash (A \to \bot) \to \bot}\,(\to\text{I})}{\vdash A \to ((A \to \bot) \to \bot)}\,(\to\text{I})$$

(ii) the proof of $((\neg p) \vee (\neg q)) \to \neg(p \& q)$ given in section 2.3.

3. Prove that if $\Delta \vdash B$ is provable in **NJ**, then so is $\Delta, A \vdash B$. (Hint: use induction on the construction of proofs.)
4. Prove that if $\Delta \vdash A$ and $\Delta' \vdash B$ are provable in **NJ**, then so is $\Delta, \Delta' \vdash A \& B$. (Hint: use the result of the previous exercise.)

2.5 The Curry-Howard Correspondence: Version I

We now turn our attention back to typing derivations and the arguments of the formal system $\mathbf{TR}_0$. This is a system in which it can be rigorously shown that a given untyped λ-term can be assigned a given type. We have seen that it is a central tool in the Curry-style typed λ-calculus.

The arguments of $\mathbf{TR}_0$ are strikingly similar to ordinary proofs in logic. For example, take the typing derivation in which it is shown that the term $\lambda x(\lambda y(x(xy)))$ can be assigned the type $(A \to A) \to (A \to A)$:

$$\dfrac{\dfrac{\dfrac{x : A \to A,\ y : A \vdash x : A \to A}{x : A \to A,\ y : A \vdash x : A \to A} \qquad \dfrac{x : A \to A,\ y : A \vdash x : A \to A \qquad x : A \to A,\ y : A \vdash y : A}{x : A \to A,\ y : A \vdash xy : A}}{x : A \to A,\ y : A \vdash x(xy) : A}}{\dfrac{x : A \to A \vdash \lambda y(x(xy)) : A \to A}{\vdash \lambda x(\lambda y(x(xy))) : (A \to A) \to (A \to A)}}$$

If in each judgment of the form $z : B$ we simply discard the left hand side z (the λ-term) and keep the right hand side B (the type), the above proof becomes the following:

$$\dfrac{\dfrac{\dfrac{A \to A,\ A \vdash A \to A}{A \to A,\ A \vdash A \to A} \qquad \dfrac{A \to A,\ A \vdash A \to A \qquad A \to A,\ A \vdash A}{A \to A,\ A \vdash A}}{A \to A,\ A \vdash A}}{\dfrac{A \to A \vdash A \to A}{\vdash (A \to A) \to (A \to A)}}$$

This is now an ordinary proof in propositional logic using only the rules ($\to$ I) and ($\to$ E), and is thus a proof in **NJ**($\to$). This should not be surprising. Our typing derivations use only the rules:

$$\Gamma, x : A \vdash x : A \quad \text{(Var)}$$

$$\dfrac{\Gamma \vdash x : A \to B \qquad \Gamma \vdash y : A}{\Gamma \vdash xy : B} \ \text{(App)}$$

$$\dfrac{\Gamma, x : A \vdash y : B}{\Gamma \vdash \lambda xy : A \to B} \ \text{(Abs)}$$

The process of discarding the λ-terms and keeping the types in these rules yields:

$$\overline{\Gamma}, A \vdash A$$

$$\dfrac{\overline{\Gamma} \vdash A \to B \qquad \overline{\Gamma} \vdash A}{\overline{\Gamma} \vdash B}$$

$$\dfrac{\overline{\Gamma}, A \vdash B}{\overline{\Gamma} \vdash A \to B}$$

where $\overline{\Gamma}$ is the list of types in Γ. This of course is precisely the set of logical axioms and rules of **NJ**($\to$).

A striking consequence of this is as follows. Suppose a term t of the untyped λ-calculus is closed (i.e., has no free variables), and can be assigned the type τ. According to the Soundness and Completeness Theorem of $\mathbf{TR}_0$ (Theorem 1.43), there is then a proof of $\vdash t : \tau$ in $\mathbf{TR}_0$. By deleting terms and keeping types in this proof, we can transform this typing derivation in $\mathbf{TR}_0$ into a proof of $\vdash \tau$ in **NJ**($\to$). Thus, if a closed term of the untyped λ-calculus can be assigned the type τ, then τ must be a theorem of **NJ**($\to$). So for example, it immediately follows that the type $\neg\neg A \to A$ will *not* in general be the type of *any* closed λ-term, as $\neg\neg A \to A$ is not a theorem of **NJ**($\to$).

This reveals a surprising connection between two things one might have thought of as quite distinct – the typing of programs, and the proving of theorems. In this and in what follows we will be examining this connection and its consequences more closely.

More generally, we have the following theorem

Theorem 2.7: The Curry-Howard Correspondence for $\mathbf{TR}_0$ and NJ($\rightarrow$), Part 1.

If $\mathbf{TR_0}$ proves

$$x_1 : A_1, ..., x_n : A_n \vdash s : B$$

then **NJ**($\rightarrow$) proves

$$A_1, ..., A_n \vdash B$$

Proof

In any proof in $\mathbf{TR}_0$, if one takes each assertion of the form $z : B$ and discards the term z leaving only the type B, the result is a proof of **NJ**($\rightarrow$).

It is easy to transform proofs in $\mathbf{TR}_0$ into proofs in **NJ**($\rightarrow$). But what about the other way around? Can we transform proofs in **NJ**($\rightarrow$) into proofs in $\mathbf{TR}_0$?

This is indeed possible. Consider the **NJ**($\rightarrow$) proof given earlier:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A \to A,\ A \vdash A \to A} \quad \dfrac{\dfrac{}{A \to A,\ A \vdash A \to A} \quad \dfrac{}{A \to A,\ A \vdash A}}{A \to A,\ A \vdash A}}{A \to A,\ A \vdash A}}{A \to A \vdash A \to A}}{\vdash\ (A \to A) \to (A \to A)}$$

This has 3 leaves; two of the form $A \to A, A \vdash A \to A$, and one of the form $A \to A, A \vdash A$. Only two types A and $A \to A$ appear in these leaves. Let us identify these types with two variables x and y, by declaring $x : A$ and $y : A \to A$. The leaf $A \to A, A \vdash A \to A$ then becomes:

$$y : A \to A, x : A \vdash y : A \to A$$

and the leaf $A \to A, A \vdash A$ becomes:

$$y : A \to A, x : A \vdash x : A$$

Moving from the leaves of the tree downwards and applying the rules of $\mathbf{TR}_0$, we may then attach a λ-term built from x and y to each type in the tree. This gives the proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : A \to A,\ y : A \vdash x : A \to A} \quad \dfrac{\dfrac{}{x : A \to A,\ y : A \vdash x : A \to A} \quad \dfrac{}{x : A \to A,\ y : A \vdash y : A}}{x : A \to A,\ y : A \vdash xy : A}}{x : A \to A,\ y : A \vdash x(xy) : A}}{x : A \to A \vdash \lambda y(x(xy)) : A \to A}}{\vdash\ \lambda x(\lambda y(x(xy))) : (A \to A) \to (A \to A)}$$

This sort of strategy can be applied quite generally. We have

Theorem 2.8: The Curry-Howard Correspondence for TR_0 and NJ($\rightarrow$), Part 2.

If **NJ**($\rightarrow$) proves

$$A_1, ..., A_n \vdash B$$

Then for any distinct variables $x_1, ..., x_n$, there is some λ-term s such that $\mathbf{TR_0}$ proves

$$x_1 : A_1, ..., x_n : A_n \vdash s : B$$

Proof

The proof is a straightforward induction on the construction of proofs in **NJ**($\rightarrow$). For the base case, consider a leaf

$$A_1, \ ... \ , A_n, B \vdash B$$

then for any distinct variables $x_1, \ ... \ , x_n, y$ we have that

$$x_1 : A_1, \ ... \ , x_n : A_n, y : B \vdash y : B$$

is a proof in $\mathbf{TR_0}$. This completes the base case.

Suppose next that a proof of $A_1, \ ... \ , A_n \vdash B$ has as its final step an application of ($\rightarrow$E):

$$\frac{\begin{array}{c}\vdots\\ A_1, \ ... \ , A_n \vdash C \rightarrow B\end{array} \qquad \begin{array}{c}\vdots\\ A_1, \ ... \ , A_n \vdash C\end{array}}{A_1, \ ... \ , A_n \vdash B} \ (\rightarrow E)$$

and let $x_1, .., x_n$ be distinct variables. By inductive hypothesis, for some λ-terms s_1 and s_2, there are proofs of

$$x_1 : A_1, \ ... \ , x_n : A_n \vdash \ s_1 : C \rightarrow B \quad \text{ and } \quad x_1 : A_1, \ ... \ , x_n : A_n \vdash \ s_2 : C$$

in $\mathbf{TR_0}$. Combining these, we then have the following proof in $\mathbf{TR_0}$

$$\frac{\begin{array}{c}\vdots\\ x_1 : A_1, \ ... \ , x_n : A_n \vdash s_1 : C \rightarrow B\end{array} \qquad \begin{array}{c}\vdots\\ x_1 : A_1, \ ... \ , x_n : A_n \vdash s_2 : C\end{array}}{x_1 : A_1, \ ... \ , x_n : A_n \vdash s_1(s_2) : B}$$

as desired.

Finally, suppose that a proof of $A_1, \ ... \ , A_n \vdash B$ has as its final step an application of ($\rightarrow$E). This means that B has the form $C \rightarrow D$:

$$\frac{\begin{array}{c}\vdots\\ A_1,\ \ldots, A_n, C \vdash D\end{array}}{A_1,\ \ldots, A_n \vdash C \to D}\ (\to \text{E})$$

Let $x_1,\ \ldots, x_n, x_{n+1}$ be distinct variables. By inductive hypothesis, for some λ-term s there is a proof of

$$x_1 : A_1,\ \ldots, x_n : A_n, x_{n+1} : C \vdash\ s : D$$

in $\mathbf{TR_0}$. We then have the following proof in $\mathbf{TR_0}$

$$\frac{\begin{array}{c}\vdots\\ x_1 : A_1,\ \ldots, x_n : A_n, x_{n+1} : C \vdash s : D\end{array}}{x_1 : A_1,\ \ldots, x_n : A_n \vdash \lambda x_{n+1}(s) : C \to D}\ (\to \text{E})$$

as desired.

The two parts of the Curry-Howard corrpesondence tell us that there is an intimate connection between proofs in $\mathbf{NJ}(\to)$ and proofs in $\mathbf{TR}_0$. Every proof in $\mathbf{TR}_0$ contains as its 'skeleton' a proof in $\mathbf{NJ}(\to)$, and every proof in $\mathbf{NJ}(\to)$ is the 'skeleton' of some proof in $\mathbf{TR}_0$.

Let us return to questions about the typing of terms, and introduce some terminology. Given a type A, one might wonder where there is some λ-term s for which s can be assigned type A. This is equivalent to asking whether there is some s such that $\vdash s : A$ is provable in $\mathbf{TR}_0$. This is sometimes called the *type inhabitation problem* for A. If $\vdash s : A$, we say that the term s 'inhabits' the type A, and that the type A is 'inhabited'. One can then ask which types are inhabited. Theorems 2.7 and 2.8 tell us that a type A is inhabited iff A is a theorem of $\mathbf{NJ}(\to)$. So a question about types is perhaps surprisingly answered by invoking a concept from intuitionistic logic.

Not only can thinking about intuitionistic logic help us answer questions about the typing of λ-terms, but thinking about the typing of λ-terms can also help us answer questions about intuitionistic logic. For example, let us ask: is the sentence

$$(A \to B) \to ((B \to C) \to (A \to C)) \tag{36}$$

a theorem of $\mathbf{NJ}(\to)$? By the Curry-Howard correspondence, this sentence is a theorem of $\mathbf{NJ}(\to)$ iff there is a closed λ-term that can be given the type (36). Careful experimentation shows that there is such a term, namely

$$\lambda x(\lambda y(\lambda z(y(xz)))) \tag{37}$$

which may be typed as follows:

$$\lambda x^{A\to B}(\lambda y^{B\to C}(\lambda z^{A}(y(xz)))).$$

That the λ-term (37) may indeed be given the type (36) is shown explicitly by the following typing derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x{:}A{\to}B, y{:}B{\to}C, z{:}A \vdash y{:}B{\to}C \qquad \dfrac{x{:}A{\to}B, y{:}B{\to}C, z{:}A \vdash x{:}A{\to}B \qquad x{:}A{\to}B, y{:}B{\to}C, z{:}A \vdash z{:}A}{x{:}A{\to}B, y{:}B{\to}C, z{:}A \vdash xz{:}B}}{x{:}A{\to}B, y{:}B{\to}C, z{:}A \vdash y(xz){:}C}}{x{:}A{\to}B, y{:}B{\to}C \vdash \lambda z(y(xz)){:}A{\to}C}}{x{:}A{\to}B \vdash \lambda y(\lambda z(y(xz))){:}(B{\to}C){\to}(A{\to}C)}}{\vdash \lambda x(\lambda y(\lambda z(y(xz)))){:}(A{\to}B){\to}((B{\to}C){\to}(A{\to}C))}$$

Eliminating the terms and keeping the types in this typing derivation then yields a proof of (37) in **NJ**($\to$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{A{\to}B, B{\to}C, A \vdash B{\to}C \qquad \dfrac{A{\to}B, B{\to}C, A \vdash A{\to}B \qquad A{\to}B, B{\to}C, A \vdash A}{A{\to}B, B{\to}C, A \vdash B}}{A{\to}B, B{\to}C, A \vdash C}}{A{\to}B, B{\to}C \vdash A{\to}C}}{A{\to}B \vdash (B{\to}C){\to}(A{\to}C)}}{\vdash (A{\to}B){\to}((B{\to}C){\to}(A{\to}C))}$$

In this way, the λ-term $\lambda x(\lambda y(\lambda z(y(xz))))$ 'encodes' a proof of $(A \to B) \to ((B \to C) \to (A \to C))$ in **NJ**($\to$). Thinking of λ-terms as programs, proofs are then encoded by programs. Searching for a proof of a given sentence in **NJ**($\to$) is then equivalent to searching for a closed λ-term (i.e., a program) with a given type.

Finally, it is worth pointing out that the Curry-Howard Correspondence also holds between $\mathbf{TR}_0^{\mathbf{bt}}$ and **NJ**($\to$). In particular, we have the following theorems

Theorem 2.9: The Curry-Howard Correspondence for $\mathbf{TR}_0^{\mathbf{bt}}$ and NJ($\to$), Part 1.

If $\mathbf{TR}_0^{\mathbf{bt}}$ proves

$$x_1 : A_1, ..., x_n : A_n \vdash s : B$$

then **NJ**($\to$) proves

$$A_1, ..., A_n \vdash B$$

Theorem 2.10: The Curry-Howard Correspondence for $\mathbf{TR}_0^{\mathbf{bt}}$ and NJ($\to$), Part 2.

If **NJ**($\to$) proves

$$A_1, ..., A_n \vdash B$$

Then for any distinct variables $x_1, ..., x_n$, there is some bound-typed λ-term s such that $\mathbf{TR}_0^{\mathbf{bt}}$ proves

$$x_1 : A_1, ..., x_n : A_n \vdash s : B$$

The proofs are trivial variants of Theorems 2.7 and 2.8, and are left to the reader.

Given the simplicity of $\mathbf{TR}_0$ and $\mathbf{NJ}(\to)$, the Curry-Howard Correspondence and its applications given so far might seem somewhat superficial. But we will see over the next several chapters that the Curry-Howard Correspondence can be generalized and applied in further surprising ways. We have started this chapter with the most transparent form of the Curry-Howard Correspondence simply in order to have a firm and clear foundation on which to build more sophisticated versions.

Exercises for Section 2.5

1. For each of the following sentences τ, (i) construct a proof in $\mathbf{NJ}(\to)$ of τ, and (ii) using this proof and the method described in this section, generate a closed λ-term s and a proof in $\mathbf{TR}_0$ of $\vdash s : \tau$. (In this way, you are showing that the type τ is inhabited.)
(a) $A \to (B \to B)$
(b) $(A \to B) \to ((B \to C) \to (A \to C))$
(c) $(A \to B) \to ((C \to A) \to (C \to B))$

2. Show that the sentence $X \to (X \to X)$ is provable in $\mathbf{NJ}(\to)$, and that there exist terms s_1 and s_2 of the untyped λ-calculus such that both

$$\vdash s_1 : X \to (X \to X) \quad \text{and} \quad \vdash s_2 : X \to (X \to X)$$

are provable in $\mathbf{TR}_0$, but that s_1 and s_2 are not $\beta\eta$-equivalent. Thus, there is no sense in which the s postulated in Theorem 2.8 is unique.

2.6 Avoiding Misunderstandings: I

The Curry-Howard Correspondence tells us that a type A is inhabited (that is, there is a closed λ-term M which may be assigned type A) iff A is a theorem of $\mathbf{NJ}(\to)$.

One might first find this result a little puzzling. Fix any types A and B. For example, suppose A is $\mathbb{N}$ and B is $\mathbb{B}$. Surely I can come up with a computer program that, when given an input of type A returns an object of type B. For example, surely I can write a computer program that when given any natural number as input just returns the truth value $\top$. Is this not a computer program of type $A \to B$, for the values of A and B in question? And will this not ultimately mean that *any* type of the form $A \to B$ is inhabited, regardless of whether $A \to B$ is a theorem of $\mathbf{NJ}(\to)$?

We must be careful here. We can certainly write down λ-terms that take as input an object of type A and that return an output of type B, for any A and B. The term $\lambda x^A(y^B)$ is such a term. This term has type $A \to B$. This is true even if $A \to B$ is not a theorem of $\mathbf{NJ}(\to)$. But this is not a counterexample to the Curry-Howard Correspondence, as the term $\lambda x^A(y^B)$ is not closed, and so

$$\vdash \lambda x(y) : A \to B$$

is *not* provable in $\mathbf{TR}_0$. By contrast, you should be able to verify that

$$y : B \vdash \lambda x(y) : A \to B$$

is provable in $\mathbf{TR}_0$. But this poses no threat to the Curry-Howard Correspondence, as deleting λ-terms and keeping types gives us:

$$B \vdash A \to B$$

which *is* provable in $\mathbf{NJ}(\to)$ for any A and B.

Alternatively, starting with the λ-term $\lambda x^A(y^B)$, we can bind the free variable y^B to obtain the closed λ-term $\lambda y^B \lambda x^A(y^B)$. But now this is a computer program with type $B \to (A \to B)$. This term *does* correspond to a theorem of intuitionistic logic, as you should be able to verify.

The lesson here is that although for arbitrary types A and B we can construct computer programs that take as inputs objects of type A and produce as output objects of type B, if the program involves free variables of any sort then it is *not* really a program of type $A \to B$, and so cannot count as refutation of the Curry-Howard Correspondence.

Still, one might have further worries. Let us go back to our computer program that takes any natural number as input, and returns the Boolean truth value $\top$. Or more generally, for any types A and B, fix some arbitrary object o^B of type B, and consider the λ-term $\lambda x^A(o^B)$. On any input of type A, this returns the fixed value o^B. The λ-term $\lambda x^A(o^B)$ is *closed*. This closed λ-term looks like a solution to the inhabitation problem for $A \to B$. And so does this not mean after all that every type of the form $A \to B$ is inhabited, regardless of whether $A \to B$ is provable in $\mathbf{NJ}(\to)$?

Again, once we think things through we see that there is no threat to the Curry-Howard Correspondence. The problem is that in the typed λ-calculus, we do *not* have constants o^B (or even $\top^{\mathbb{B}}$) at our disposal. The term $\lambda x^A(o^B)$ is *not* a term of the typed λ-calculus as we have defined it. Any constants we want to introduce must be explicitly defined. For some types, it is easy to define a constant of that type. But for other types we simply cannot.

For example, for any type of the form $B \to B$ we can easily define an object o of type $B \to B$, just by regarding $o^{B \to B}$ as an abbreviation for the closed λ-term $\lambda x^B(x^B)$. Understanding o^B as an abbreviation in this way, the term $\lambda x^A(o^B)$ is perfectly acceptable. But this term has type $A \to (B \to B)$, which does indeed correspond to a theorem of $\mathbf{NJ}(\to)$.

By contrast, consider some simple type B such as $\mathbb{N}$ or $\mathbb{B}$. Given only the machinery in the typed λ-calculus as we have defined it, we *cannot* define a closed term with type B. Any definition of any object in the typed λ-calculus by a closed term must have the form $\lambda x^\rho(U)$ for for some U, and will thus have a type of the form $\rho \to \rho'$ for some ρ and ρ'. We thus cannot define *any* specific object of *any* fundamental type with a closed term. So although we can use variables such as x^B in our programming language, we do not have prior access to any constants of this type, nor can we define them. Of course, as we saw earlier we can define objects that behave like Boolean truth values or natural numbers, but these are actually objects of more complex types that merely mimic the Booleans or natural numbers. They do not provide us with constants of the form o^B for any simple type B. Thus, we cannot view $\lambda x^A(o^B)$ as a λ-term that solves the inhabitation problem for $A \to B$ for any simple type B.

In fact, it is a consequence of the Curry-Howard Correspondence that we can define a closed term of type B iff B is a theorem of **NJ**$(\rightarrow)$. (This further shows how restricted a programming language the typed λ-calculus is.) Thus, we can use terms of the form $\lambda x^A o^B$ to solve the inhabitation problem for $A \rightarrow B$ iff B is already a theorem of **NJ**$(\rightarrow)$. But if B is already a theorem of **NJ**$(\rightarrow)$, then so is $A \rightarrow B$. So again, there is no threat to the Curry-Howard Correspondence here.

2.7 More Consequences: Proof Normalization.

One important area in which the Curry-Howard correspondence can be put to very good use is *proof theory*. This is a rich and complex area of mathematics on its own, and we merely introduce the reader to a few important results without any claim to exhaustiveness. Some of the ideas introduced in this section will also be central in our later vindication of the **BHK** conditions.

Consider the following proof of $Y \rightarrow (X \rightarrow Y)$ in **NJ**$(\rightarrow)$

$$\dfrac{\dfrac{\dfrac{\dfrac{X,\ X,\ Y\ \vdash\ Y}{X,\ Y\ \vdash\ X \rightarrow Y}\,(\rightarrow \text{I}) \qquad X,\ Y\ \vdash\ X}{X,\ Y\ \vdash\ Y}\,(\rightarrow \text{E})}{Y\ \vdash\ X \rightarrow Y}\,(\rightarrow \text{I})}{\vdash\ Y \rightarrow (X \rightarrow Y)}\,(\rightarrow \text{I}) \tag{38}$$

This proof is completely correct. However, it is perverse and inefficient. For example, the conclusion of the $(\rightarrow$ E) inference is the sequent $X, Y \vdash Y$, which is a logical axiom with which we could have just begun the proof. There are other curious features of this proof that we will also turn to shortly.

Our goal is to discuss methods of proof simplification that could be used to at least somewhat simplify proofs like (38). There are presumably many different approaches that could be taken here, but as our goal is to develop very general methods of proof simplification, we want to focus on general principles rather than ad-hoc methods that only apply in very specific cases. We will see that there are in fact very general methods of proof simplification that can be applied not just to (38), but to any proof.

Let us then focus on a more subtle oddity of (38). Note that the proof begins with an application of the ($\rightarrow$I) rule introducing a conditional $X \rightarrow Y$, which is then immediately followed by an application of the ($\rightarrow$E) rule eliminating this very conditional. The result is the logical axiom $X, Y \vdash Y$ with which we began the proof. There is something odd about introducing a conditional and then immediately eliminating it, and one might wonder whether this is the main cause of the inefficiency of (38).

Even more specifically, the proof (38) involves an application of the ($\rightarrow$E) rule whose major premise involves a conditional $X \rightarrow Y$ that has just been introduced with the use of the ($\rightarrow$I) rule. Let us consider this pattern of reasoning more generally. We begin with the definition

Definition 2.11: Normal Proof of NJ($\to$).

A proof in **NJ**($\to$) is *normal* just in case it contains no occurrence of a ($\to$I) rule introducing a conditional $X \to Y$ which is immediately followed by an application of the ($\to$E) rule with this conditional $X \to Y$ as its major premise.

The proof (38) is not normal. We would like to consider systematic ways of transforming it into a normal proof.

To this end, consider the following typing derivation that corresponds to (38) using the Curry-Howard correspondence

$$\dfrac{\dfrac{\dfrac{\dfrac{}{u:X,\ v:X,\ w:Y\ \vdash\ w:Y}}{u:X,\ w:Y\ \vdash\ \lambda v(w):X\to Y} \qquad \dfrac{}{u:X,\ w:Y\ \vdash\ u:X}}{u:X,\ w:Y\ \vdash\ \lambda v(w)(u):Y}}{w:Y\ \vdash\ \lambda u(\lambda v(w)(u)):X\to Y}}{\vdash\ \lambda w\lambda u(\lambda v(w)(u)):Y\to(X\to Y)} \tag{39}$$

Just as there is something inefficient and unsatisfactory about (38), one would expect there to be something inefficient and unsatisfactory about (39). And indeed, there is something unsatisfactory about (39) that is very easy to identify. The typing derivation (39) shows that the type $Y \to (X \to Y)$ is inhabited. The example it provides of a term that can be given this type is $\lambda w\lambda u(\lambda v(w)(u))$. But this term is not in β-normal form and can be simplified. When it is β-reduced, we get the β-normal form term $\lambda w\lambda u(w)$. Theorem 1.44 tells us that whenever an untyped λ-term M can be assigned a type and $M \twoheadrightarrow_\beta N$, then N can also be assigned that same type. Thus, we could have much more easily have shown that the type $Y \to (X \to Y)$ was inhabited by showing that the term $\lambda w\lambda u(w)$ can be given this type. in fact, such a typing derivation can easily be constructed

$$\dfrac{\dfrac{\dfrac{}{w:Y,\ u:X\ \vdash\ w:Y}}{w:Y\ \vdash\ \lambda u(w):X\to Y}}{\vdash\ \lambda w\lambda u(w):Y\to(X\to Y)} \tag{40}$$

Removing the terms from this typing derivation and keeping the types then gives the following **NJ**($\to$) proof of $Y \to (X \to Y)$

$$\dfrac{\dfrac{\dfrac{}{Y,X\ \vdash\ Y}\,(\to \text{I})}{Y\ \vdash\ X\to Y}\,(\to \text{I})}{\vdash\ Y\to(X\to Y)} \tag{41}$$

This is a very much simpler proof of $Y \to (X \to Y)$. Note moreover that (41) is a normal proof.

In fact, there is a very general method of transforming non-normal proofs to normal proofs lurking here. Before stating the main theorem that shows the general phenomenon, we need a lemma

Lemma 2.12: The Subterm Lemma for $\mathbf{TR}_0$

In any typing derivation of $\mathbf{TR}_0$ with conclusion of the form $\Gamma \vdash s : A$, if s' is a subterm of s then the typing derivation contains a sequent of the form $\Gamma' \vdash s' : B$, where $\Gamma' \subseteq \Gamma$.

The proof of this lemma is a straightforward induction on the construction of typing derivations $\Gamma \vdash s : A$ and is left to the exercises.

Armed with this lemma, the following theorem can be easily proven

Theorem 2.13

Given a typing derivation T in $\mathbf{TR}_0$ whose conclusion is a sequent of the form $\Delta \vdash g : C$, let P be the $\mathbf{NJ}(\rightarrow)$ proof obtained from T by deleting all terms and keeping types. Then P is a normal proof iff g is in β-normal form.

This theorem shows that there is an intimate connection between non-normal proofs and typing derivations of terms not in β-normal form. The proof is straightforward

Proof

First we show that if g is not in β-normal form, then P is not normal. Suppose then that T is a typing derivation with conclusion $\Delta \vdash g : C$, where g is not in β-normal form. It follows from Lemma 2.12 that T contains a sequent of the form

$$\Gamma \vdash \lambda u(e)(f) : D.$$

The step prior to this sequent must be the (App) rule

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash \lambda u(e) : E \rightarrow D & \Gamma \vdash f : E\end{array}}{\Gamma \vdash \lambda u(e)(f) : D}\ (\text{App})$$

and the step prior to the leftmost premise of this inference must be the (Abs) rule

$$\frac{\begin{array}{cc}\begin{array}{c}\vdots \\ \Gamma, u : E \vdash e : D\end{array} & \\ \hline \Gamma \vdash \lambda u(e) : E \rightarrow D\ (\text{Abs}) & \begin{array}{c}\vdots \\ \Gamma \vdash f : E\end{array}\end{array}}{\Gamma \vdash \lambda u(e)(f) : D}\ (\text{App})$$

Deleting all terms and keeping types then gives the $\mathbf{NJ}(\rightarrow)$ proof P

$$\dfrac{\dfrac{\vdots}{\dfrac{\overline{\Gamma}, E \vdash D}{\overline{\Gamma} \vdash E \to D}\,(\to\text{I})} \qquad \dfrac{\vdots}{\overline{\Gamma} \vdash E}}{\overline{\Gamma} \vdash D}\,(\to\text{E})$$

for some list $\overline{\Gamma}$ of formulae. This proof P is not normal.

Next we show that if P is not normal, then g is not in β-normal form. Suppose then that P has the form

$$\dfrac{\dfrac{\dfrac{\vdots}{\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\,(\to\text{I})} \qquad \dfrac{\vdots}{\Gamma \vdash A}}{\Gamma \vdash B}\,(\to\text{E})}{\vdots}$$
$$\Delta \vdash C$$

Any typing derivation T from which this is derived must have the form

$$\dfrac{\dfrac{\dfrac{\vdots}{\overline{\Gamma}, u : A \vdash e : B}}{\overline{\Gamma} \vdash \lambda u(e) : A \to B} \qquad \dfrac{\vdots}{\overline{\Gamma} \vdash f : A}}{\overline{\Gamma} \vdash \lambda u(e)(f) : B}$$
$$\vdots$$
$$\overline{\Delta} \vdash g : C$$

where u is a variable, e, f and g are λ-terms, and $\overline{\Gamma}$ and $\overline{\Delta}$ are contexts.

Because in any typing derivation, every λ-term that appears in the proof is α-equivalent to a subexpression of any λ-term lower in the proof, it follows that g has $\lambda u(e)(f)$ (or something α-equivalent to it) as a subexpression. Thus g is not in β-normal form.

Theorem 2.13 gives us a method for transforming any non-normal proof into a normal proof, by a combination of the Curry-Howard Correspondence and β-reduction. For given any non-normal proof P, we may transform it into a typing derivation T that contains P as its 'skeleton'. By Theorem 2.13, this will be a typing derivation for a term e that is not in β-normal form, and it will assign e some type τ. If e β-reduces to e' where e' is in β-normal form, then e' can also be assigned the type τ. We can thus produce a typing derivation that shows that e' can be assigned the type τ. By Theorem 2.13, the skeleton of this typing derivation will be a normal proof P' (with the same conclusion as P). We have thus transformed the non-normal proof P into a normal proof P'. The procedure is competely mechanical and requires no ingenuity. This procedure is often called 'proof normalization'. Precisely this procedure transforms the proof (38) into the simpler proof (39). In the exercises, you will apply it to some other non-normal proofs.

Note then that we have the following result:

Theorem 2.14: Proof Normalization Theorem for NJ($\to$)

If $A_1, ..., A_n \vdash B$ has a proof in **NJ**($\to$), then it has a normal proof in **NJ**($\to$).

Proof

Suppose P_1 is a non-normal proof of $A_1, ..., A_n \vdash B$. By the Curry-Howard Correspondence (in particular, Theorem 2.8), this may be transformed into a typing derivation with conclusion

$$x_1 : A_1, ..., x_n : A_n \vdash s : B$$

for some λ-term s. By Theorem 2.13, s is not in β-normal form. Let s' be the β-normal form of s. By Theorem 1.44 (the Subject Reduction Theorem) there is a typing derivation of

$$x_1 : A_1, ..., x_n : A_n \vdash s' : B$$

in $\mathbf{TR}_0$. Because s' is in β-normal form, it follows from Theorem 2.13 that the **NJ**($\to$) proof obtained by taking this typing derivation and removing all terms and keeping types is a normal proof of $A_1, ..., A_n \vdash B$.

Normal proofs have a simple structure that makes them very useful in proving facts about **NJ**($\to$) or $\mathbf{TR}_0$. To see how, we first need another definition. Proofs in **NJ**($\to$) are *trees*. Such trees consists of *branches*, where a branch is a linear sequence of sequents that extends from some leaf of the tree (i.e., a logical axiom), and is such that if it includes some sequent then it includes the sequent immediately below it. In this way, a branch extends from a leaf of the tree all the way to the very bottom of the tree (i.e., the conclusion of the proof.) Consider for example the proof:

$$\dfrac{\dfrac{A, A \to B \vdash A \qquad A, A \to B \vdash A \to B}{A, A \to B \vdash B}\,(\to \text{E})}{A \vdash (A \to B) \to B}\,(\to \text{I}) \tag{42}$$

This has exactly two branches:

Branch 1: $A, A \to B \vdash A \qquad A, A \to B \vdash B \quad A \vdash (A \to B) \to B$
Branch 2: $A, A \to B \vdash A \to B \quad A, A \to B \vdash B \quad A \vdash (A \to B) \to B$

The following definition is useful

Definition 2.15: Major Branch

A *major branch* of a proof is a branch of the proof with the property that if the conclusion of an inference lies on the branch and that inference has a major premise, then that major premise also lies on in the branch.

You should be able to verify that in the proof just given, **Branch 1** is not a major branch, and **Branch 2** is a major branch.

Each proof contains at least one major branch: simply start at the bottom of the tree and move upwards from the conclusion of rule to its premises, picking the major premise whenever one reaches a (→E) rule and one has two premises to pick from. Proceeding upwards in this way, one will eventually reach a leaf of the tree. In this way, one traces out a major branch. In the case of **NJ**(→) proofs, the major branch is unique, though this will not hold true in more complicated systems we consider later.

Note that in a normal proof, on a major branch the (→I) rule cannot be immediately followed by the (→E) rule, as this would introduce the pattern:

major branch

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, A \vdash B}}{\Gamma \vdash A \to B}(\to I) \qquad \vdots}{\dfrac{\Gamma \vdash B}{\vdots}} \; \Gamma \vdash A \; (\to E)$$

which would mean that the original proof was non-normal. Thus, on a major branch the (→I) rule can only be followed by the (→I) rule. This means that a major branch can only consist of a sequence of (→E) rules followed by a sequence of (→I) rules. For example, in the normal proof (42), **Branch 2** consists of a single (→E) rule followed by a single (→I) rule.

We have thus proven

Lemma 2.16: The Major Branch Lemma.

In any normal proof in **NJ**(→), a major brach will consist of a (possibly empty) sequence of (→E) rules followed by a (possibly empty) sequence of (→I) rules.

This modest looking lemma turns out to be very useful. For example, with it one is easily able to prove the following

Theorem 2.17: Consistency of NJ($\to$).

No atomic sentence X is provable in **NJ**($\to$).

The only atomic sentences in the language used in **NJ**($\to$) are the propositional variables $X, Y, Z, ...$ and the sentence $\bot$. This theorem tells us that these sentences are not theorems of **NJ**($\to$). This theorem may thus be viewed as a type of consistency proof for **NJ**($\to$).

Proof

Suppose **NJ**($\to$) proves S where S is either a propositional variable or the symbol $\bot$. We then know from Theorem 2.14 that there is a normal proof P whose conclusion is $\vdash S$. The last step of P cannot be the ($\to$I) rule, because S does not have the form of a conditional. So using Lemma 2.16, we have that any major branch of P must consist solely of a sequence of ($\to$E) rules. But the ($\to$E) rule leaves the left hand side of a sequent unchanged. Because the left hand side of every leaf is non-empty, it follows that if a major branch of P consists only of ($\to$E) rules, then the left hand side of its bottommost sequent must also be non-empty. But the left hand side of $\vdash S$ is empty, which is a contradiction.

Some further intriguing applications of the Major Branch Lemma will be given in the exercises.

The simplicity of **NJ**($\to$) might lead one to think that all this effort and terminology to prove these theorems is perhaps overkill. But will see that the basic ideas here can be generalized quite straightforwardly to help us analyze much stronger proof systems in a very similar way. As with the Curry-Howard Correspondence, we start with the simplest version of these ideas in order to have a firm and clear foundation on which to generalize them later.

Exercises for Section 2.7

1. Each of the following proofs in **NJ**($\to$) is *not* normal. Use the techniques of this section to transform them into normal proofs. In particular, begin by decorating the proofs in such a way to produce proofs in $\mathbf{TR}_0$ of $\vdash M : \tau$, where τ is the conclusion of the original **NJ**($\to$) proof. You should notice that M is not in β-normal form. Reduce M to a β-normal form M', and produce a derivation of $\vdash M' : \tau$ in $\mathbf{TR}_0$. This later typing derivation should encode a normal proof of τ, and you should verify that this is the case.

(a)

$$\dfrac{\dfrac{\dfrac{A, A \vdash A}{A \vdash A \to A}(\to \text{I}) \qquad A \vdash A}{A \vdash A}(\to \text{E})}{\vdash A \to A}(\to \text{I})$$

(b)

$$\dfrac{\dfrac{\dfrac{A, A, B \vdash B}{A, B \vdash A \to B}(\to \text{I}) \qquad A, B \vdash A}{A, B \vdash B}(\to \text{E})}{B \vdash A \to B}(\to \text{I})}{\vdash B \to (A \to B)}(\to \text{I})$$

2. Show that the following are not derivable in **NJ**$(\to)$, where X and Y are distinct atomic sentences. (Hint: use the Major Branch Lemma.) In each case, you make invoke the result from earlier cases.

(i) $X, (X \to Y) \to X \vdash Y$
(ii) $(X \to Y) \to X \vdash X \to Y$
(iii) $\vdash ((X \to Y) \to X) \to X$ (Pierce's Rule).

(side note: Pierce's Rule *is* provable in classical logic.)

3. Using the Major Branch Lemma, show that every closed λ-term in β-normal form has the form

$$\lambda x_1 \lambda x_2 ... \lambda x_n (y(E_1)(E_2)...(E_m))$$

where $n \geq 1$, y is a variable, $m \geq 0$, and each $E_1, ..., E_m$ is in β-normal form.

4. We have seen that a proof of $\Delta \vdash E : X$ in $\mathbf{TR}_0$ in which the λ-term E is in β-normal form corresponds to a proof in **NJ**$(\to)$ in which a certain pattern of reasoning does not appear. What additional pattern of reasoning does not appear if the λ-term E is assumed to be in $\beta\eta$-normal form?

5. Prove Lemma 2.12. (Hint: proceed by induction on the construction of typing derivations $\Gamma \vdash s : A$.)

6. Is there a normal proof of **NJ**$(\to)$ that includes a sequent $\Gamma, X \vdash X$ *not* as a leaf of the proof?

7. Is there a normal proof of **NJ**$(\to)$ in which an occurrence of the $(\to$I) rule is immediately followed by an occurrence of the $(\to$E) rule, where it is *not* necessarily required that the conclusion of the $(\to$I) rule be the major premise of the $(\to$E) rule?

2.8 Adding Types: Ordered Pairs

The Curry-Howard correspondence shows that there is a perhaps surprising connection between certain natural concepts in the λ-calculus and the proof theory of **NJ**$(\to)$. The deeper significance of the Curry-Howard correspondence however comes from the fact that these ideas can be generalized so that they hold in logical systems far stronger than **NJ**$(\to)$. We begin this process of generalization now.

We have already argued that the typed λ-calculus can be viewed as a kind of (typed) program-

ming language. It is natural to want to expand the typed λ-calculus λ_0^{type} and its corresponding system $\mathbf{TR}_0$ of typing rules to include more complex types that repeatedly arise in computer programming.

In a computer program, it is common to want to store two distinct items into a single data structure – an ordered pair – for subsequent manipulation. To capture this practice, given two untyped λ-terms x and y (i.e., two 'computer programs'), we allow the formation of a new λ-term (i.e., a new computer program) $\langle x, y\rangle$, understood as the ordered pair with x as its first element and y as its second element. In order to extract information from ordered pairs, it is natural to also introduce into our programming language the functions π_1 and π_2 that take an ordered pair to its first and second elements respectively. Thus, in addition to our old formation rules, we allow the following new formation rules for λ-terms:

1. If x and y are λ-terms, then so is $\langle x, y\rangle$.
2. If x is a λ-term, then so are $\pi_1(x)$ and $\pi_2(x)$.

So for example, in this expanded version of the λ-calculus we have terms like the following:

$\langle z, x\rangle$
$\pi_1(\langle y, z\rangle)$
$\langle z, x\rangle(\pi_1(\langle y, z\rangle))$
$\lambda x\ (\langle z, x\rangle(\pi_1(\langle y, z\rangle)))$
$\pi_2(\lambda x\ (\langle z, x\rangle\ (\pi_1(\langle y, z\rangle))))$

As before, we will omit parentheses where this causes no confusion, subject to our usual conventions. For example, the last term on our list can perhaps be written more simply as

$$\pi_2(\lambda x\ (\langle z, x\rangle\ \pi_1\langle y, z\rangle)). \tag{43}$$

The free variables of $\langle x, y\rangle$ are just all the free variables of x and all the free variables of y, and the free variables of $\pi_1(x)$ and $\pi_2(x)$ are just the free variables of x. So for example, in the λ-term (43) the free variables are z and y. If a variable occurs both free and bound, as in the term $\langle x, \lambda x(x)\rangle$, renaming of bound variables can be useful in determining which occurrences of which variables are free or bound. So we can rewrite this term in an α-equivalent form as $\langle x, \lambda y(y)\rangle$, making it easier to see that the variable x is free and the variable y bound.

We expand the notion of β-reduction in this new programming language to allow the following reductions:

1. $\pi_1(\langle M, N\rangle) \to_\beta M$
2. $\pi_2(\langle M, N\rangle) \to_\beta N$

So for example, consider the term

$$\pi_1(\lambda x(x)(\langle y, z\rangle)).$$

This β-reduces to $\pi_1(\langle y, z\rangle)$, which then β-reduces to y. On the other hand, the term

$$\lambda x(x)(\pi_2(\langle y, z\rangle))$$

can be β-reduced in several ways. We have the following two β-reduction sequences:

$$\lambda x(x)(\pi_2(\langle y, z\rangle)) \to_\beta \lambda x(x)(z) \to_\beta z$$

$$\lambda x(x)(\pi_2(\langle y, z\rangle)) \to_\beta \pi_2(\langle y, z\rangle) \to_\beta z$$

Note that both β-reduction sequences give the same β-normal form term z. We will see later that the Church-Rosser Theorem holds for this new system, and so this convergence of results is not surprising.

Thus far, we have simply added some machinery to the untyped λ-calculus. The main task of the typed λ-calculus is the typing of untyped terms, and so in order to update the typed λ-calculus to accommodate our new constructions, we must think about how we should assign a type to $\langle x, y\rangle$. Here, there is a very natural idea: assuming that the type A can be assigned to x, and the type B can be assigned to y, we create a brand new type - the *product type* $A \times B$ - and assert that it can be assigned to $\langle x, y\rangle$. Given two types A and B, the product type $A \times B$ is simply the type of ordered pairs consisting of an element of A followed by an element of type B. This provides us with a brand new class of data structure. Up to now, the only way to form complex types from simple types was to build function types $A \to B$. Product types now give us a new way of forming more complex types.

The product type is governed by the following typing rules, which must be added to the rules of $\mathbf{TR}_0$:

$$\frac{\Gamma \vdash x : A \qquad \Gamma \vdash y : B}{\Gamma \vdash \langle x, y\rangle : A \times B}$$

$$\frac{\Gamma \vdash x : A \times B}{\Gamma \vdash \pi_1(x) : A} \qquad \frac{\Gamma \vdash x : A \times B}{\Gamma \vdash \pi_2(x) : B}$$

These rules tell us that if x has type A and y has type B, then the ordered pair $\langle x, y\rangle$ has type $A \times B$. Conversely, if x has type $A \times B$, then $\pi_1(x)$ has type A and $\pi_2(x)$ has type B. Nothing should be surprising about these rules.

We have expanded the set of λ-terms, as well as the set of possible types a λ-term can be assigned. As a result, old terms can be assigned new types. For example, the λ-term $\lambda x(x)$ can now be assigned the type $(A \times B) \to (A \times B)$ for any types A and B, as the following typing derivation shows

$$\frac{x : A \times B \vdash x : A \times B}{\vdash \lambda x(x) : (A \times B) \to (A \times B)}$$

New terms can also be assigned new types. For exampe, the closed λ-term

$$\langle \lambda x(x), \lambda y(y)\rangle$$

can be given the type $(A \to A) \times (B \to B)$ for any A and B, as the following typing derivation shows:

$$\dfrac{\dfrac{x : A \vdash\ x : A}{\vdash\ \lambda x(x) : A \to A} \qquad \dfrac{y : B \vdash\ y : B}{\vdash\ \lambda y(y) : B \to B}}{\vdash\ \langle \lambda x(x), \lambda y(y) \rangle : (A \to A) \times (B \to B)}$$

For a slightly more complex example, consider the closed λ-term

$$\lambda x(\pi_1 \langle \lambda y(x), x \rangle).$$

This term can be given type $A \to (B \to A)$ for any A and B, as the following typing derivation shows:

$$\dfrac{\dfrac{\dfrac{x : A,\ y : B \vdash\ x : A}{x : A \vdash\ \lambda y(x) : B \to A} \qquad x : A \vdash\ x : A}{x : A \vdash\ \langle \lambda y(x), x \rangle : (B \to A) \times A}}{\dfrac{x : A \vdash\ \pi_1 \langle \lambda y(x), x \rangle : B \to A}{\vdash\ \lambda x(\pi_1 \langle \lambda y(x), x \rangle) : A \to (B \to A)}}$$

As before, λ-terms with free variables (i.e., open λ-terms) can still be typed, but the relevant typing derivation will have a non-empty context in its conclusion, indicating the types of the free variables. So for example, we might expect that $\pi_1 \langle \lambda x(x), z \rangle$ can be assigned the type $A \to A$ for any A. The following typing derivation confirms this:

$$\dfrac{\dfrac{\dfrac{z : B,\ x : A \vdash\ x : A}{z : B \vdash\ \lambda x(x) : A \to A} \qquad z : B \vdash\ z : B}{z : B \vdash\ \langle \lambda x(x), z \rangle : (A \to A) \times B}}{z : B \vdash\ \pi_1 \langle \lambda x(x), z \rangle : A \to A}$$

Note that the conclusion of this typing derivation contains the non-empty context $z : B$ declaring the type of the free variable z.

Exercises for Section 2.8

1. Write each of the following λ-terms in β-normal form.
(a) $\pi_1(\pi_2(\langle x, y \rangle))$
(b) $\lambda x(\pi_1(\pi_1(\langle x, y \rangle)))(\pi_2(z))$
(c) $\lambda x(\pi_2(\pi_1(\langle x, y \rangle)))(\langle y, z \rangle)$

2. Each of the following untyped λ-terms can be assigned types. Find types for them, and confirm your result by producing an appropriate typing derivation.
(a) $\lambda x(\pi_1(\langle x, x \rangle))$
(b) $\lambda x(\pi_1(\langle y, x \rangle))$ (note: this is an open λ-term)
(c) $\lambda y(\langle \lambda x(\pi_1(\langle x, y \rangle)), y \rangle)$

2.9 Do We Need Product Types?

As an aside that may be skipped if the reader wishes, one might wonder whether we really need to add new machinery to the λ-calculus in order to encode ordered pairs. For example, given the untyped λ-terms E and F, consider the untyped λ-term

$$\lambda x(x(E)(F)).$$

Call this term $\mathbf{PAIR}(E, F)$. We will see that this λ-term 'stores' E and F. In particular, letting **FIRST** be the λ-term $\lambda y(y(\lambda u \lambda v(u)))$ and letting **SECOND** be the λ-term $\lambda y(y(\lambda u \lambda v(v)))$, and assuming x, v are not free in E or F, we have

$$\begin{aligned}\mathbf{FIRST}(\mathbf{PAIR}(E,F)) &= \lambda y(y(\lambda u(\lambda v(u))))(\lambda x(x(E)(F)))\\ &\to_\beta \lambda x(x(E)(F))(\lambda u(\lambda v(u)))\\ &\to_\beta \lambda u(\lambda v(u))(E)(F)\\ &\to_\beta \lambda v(E)(F)\\ &\to_\beta E\end{aligned}$$

and

$$\begin{aligned}\mathbf{SECOND}(\mathbf{PAIR}(E,F)) &= \lambda y(y(\lambda u \lambda v(v)))(\lambda x(x(E)(F)))\\ &\to_\beta \lambda x(x(E)(F))(\lambda u(\lambda v(v)))\\ &\to_\beta \lambda u(\lambda v(v))(E)(F)\\ &\to_\beta \lambda v(v)(F)\\ &\to_\beta F\end{aligned}$$

Thus, **FIRST** and **SECOND** 'extract' E and F respectively from $\mathbf{PAIR}(E, F)$. So $\mathbf{PAIR}(E, F)$ looks like it can play the role of an ordered pair, and **FIRST** and **SECOND** can play the role of π_1 and π_2.

This is fine as far as the untyped λ-calculus goes – if the untyped λ-calculus were all we cared about, we could indeed dispense with product types in precisely this way. But unfortunately, this approach fails for the typed λ-calculus, as **FIRST**, **SECOND** and $\mathbf{PAIR}(E, F)$ cannot be typed in a consistent manner.

To see why, suppose E has type A and F has type B. What type will $\mathbf{PAIR}(E, F)$ - i.e., $\lambda x(x(E)(F))$ receive? For the subexpression $x(E)(F)$ to be grammatical, x must have type $A \to (B \to C)$ for some type C. Then $x(E)(F)$ will have type C, and $\lambda x(x(E)(F))$ will have type $(A \to (B \to C)) \to C$. So $\mathbf{PAIR}(E, F)$ will have type $(A \to (B \to C)) \to C$ for some C.

Because $\mathbf{FIRST}(\mathbf{PAIR}(E, F))$ simplifies to E which is of type A, and $\mathbf{PAIR}(E, F)$ has type $(A \to (B \to C)) \to C$, it follows that **FIRST** must have type

$$((A \to (B \to C)) \to C) \to A.$$

Writing out **FIRST** as the λ-term $\lambda y(y(\lambda u \lambda v(u)))$, it follows that this term must be typed in the following way:

$$\lambda y^{(A\to(B\to C))\to C}(y^{(A\to(B\to C))\to C}(\lambda u \lambda v(u)))^A \tag{44}$$

(Here we use the fact that if $\lambda s(t)$ has type $E \to F$, then s must have type E and t must have type F.) How can we assign types to u and v for the expression (44) to make sense? In order for the subexpression

$$y^{(A\to(B\to C))\to C}(\lambda u \lambda v(u)) \tag{45}$$

to be grammatical, $\lambda u \lambda v(u)$ must have type $A \to (B \to C)$, and so then expression (45) gets type C. But looking at (44), the subexpression (45) was supposed to get type A. This means that C, which up to now was unspecified, must be A, and so **PAIR**(E, F) has type $(A \to (B \to A)) \to A$.

Because **SECOND**(**PAIR**(E, F)) simplifies to F which is of type B, and **PAIR**(E, F) has type $(A \to (B \to A)) \to A$, it follows that **SECOND** must have type

$$((A \to (B \to A)) \to A) \to B.$$

Writing out **SECOND** as the λ-term $\lambda y(y(\lambda u \lambda v(v)))$, it follows that this term must be typed in the following way:

$$\lambda y^{(A\to(B\to A))\to A}(y^{(A\to(B\to A))\to A}(\lambda u \lambda v(v)))^B \tag{46}$$

This means that the subexpression $\lambda u \lambda v(v)$ of (46) must get the type $A \to (B \to A)$. This can only happen as follows:

$$\lambda u^A \lambda v^B(v^A)$$

but then v is assigned both A and B as types, which need not be equal.

So we cannot type the terms **FIRST**, **SECOND** and **PAIR** in a consistent way. This problem plagues all attempts to try to simulate ordered pairs and projection functions within the λ-calculus λ_0. (You will see the deeper reason for this in the exercises.) It is for this reason that we need to introduce pairing and projections into the λ-calculus directly if we hope to incorporate this sort of machinery into a version of the *typed* λ-calculus.

Exercises for Section 2.9

1. Show that there do not exist typable λ_0-terms **FIRST**, **SECOND** and **PAIR**(x, y) such that the following conditions all hold:

(i) **FIRST** and **SECOND** are closed λ_0-terms
(ii) **PAIR**(x, y) is an open λ_0-term with free variables x and y,
(iii) **FIRST(PAIR**$(x, y)) \twoheadrightarrow_\beta x$
(iv) **SECOND(PAIR**$(x, y)) \twoheadrightarrow_\beta y$

(Hint: suppose to the contrary that such typable λ_0-terms exist. Then assuming that the variables x and y have types A and B respectively, we will be able to assign **PAIR**(x, y) some type P (which may depend on A and B.) Because we can input **PAIR**(x, y) to **FIRST** and

the result is of type A, it follows that **FIRST** must in fact have the type $P \to A$. Likewise, **SECOND** must in fact have the type $P \to B$. Thus, the following three sequents must be derivable in $\mathbf{TR}_0$:

$$x : A, y : B \vdash \mathbf{PAIR}(x, y) : P \qquad \vdash \mathbf{FIRST} : P \to A \qquad \vdash \mathbf{SECOND} : P \to B$$

By the Curry-Howard Correspondence, the following three sequents must then also be derivable in $\mathbf{NJ}(\to)$:

$$A, B \vdash P \qquad \vdash P \to A \qquad \vdash P \to B$$

Assuming without loss of generality that A, B are distinct propositional variables, show that no sentence P exists with this property. This in effect amounts to showing that conjunction is not definable from implication in $\mathbf{NJ}(\to)$. The Major Branch Lemma appled to $\vdash P \to A$ and $\vdash P \to B$ will help here. In particular, using the Major Branch Lemma you should be able to show that P has both the forms

$$C_n \to ... \to (C_2 \to (C_1 \to B))$$

and

$$D_m \to ... \to (D_2 \to (D_1 \to A)),$$

which is a contradiction.)

2.10 Adding Types: Sums

From the point of view of a computer programmer, there are other types it is natural to want to manipulate beyond function types and product types. For example, given two types A and B, it is natural to want to consider something like their union. We will denote this $A + B$. It is convenient to think of this as not just a union, but as something a little more complex. In particular, unlike a plain union, it is convenient to think of each element of $A + B$ as an object from A or B which also has attached to it a label indicating whether it was drawn from A or B. Just by looking at this label we can tell whether the object is from A or B, without looking at the object itself. In such a case, we call $A + B$ a *sum type*.

One way (amongst many) to think of the elements of $A + B$ is as something like ordered pairs $\langle a, b\rangle$ in which $a = 1$ and b is an element of A, or $a = 2$ and b is an element of B. As a result, merely by looking at the label a we can know whether b came from A or B. On this way of doing things, given an object x in A, x itself is not an element of $A + B$, but $\langle 1, x\rangle$ is. Likewise, given an object x in B, x itself is not an element of $A + B$, but $\langle 2, x\rangle$ is. This is one way of implementing the idea of 'labels' being attached to terms in $A + B$. So for example, on this way of doing things the sum type $A + A$ consists of the objects of the form $\langle i, x\rangle$ where i is 1 or 2, and x an object of type A.

We will *not* proceed in this way, but will instead think of these 'labels' in slightly more abstract terms. We do this by assuming that in our programming language we have a type of **if-then-else**

command that, when input with an element x of type $A+B$, does one thing if x is something with the label 1 attached to it, and something else if it sees that x is something with the label 2 attached to it. More specifically, we assume we have access to commands in our computer language that say something like:

> If x is an object z with the label 1 attached, then run the program $P[z/v]$; i.e., run the program P with the free variable v replaced with z, and if x is an object z with the label 2 attached, run the program $Q[z/w]$; i.e., run the program Q with the free variable w replaced with z. (47)

For the sake of completeness, we can also specify some sort of behavior when x has neither the forms in question (e.g., we could have the program halt and return an error message, or just output some fixed value and continue), though in what follows the details of what happens in this case will not matter.

We denote the command described in (47) by

$$\textbf{case}(x, [v]P, [w]Q).$$

Of course, the programs P and Q will here be λ-terms, and by 'running' a program, we just mean making the appropriate substitutions and then β-reducing (or $\beta\eta$-reducing) the resulting expression.

The notation $[v]P$ is perhaps unfamiliar. This is really just the program (or λ-term) P, but with a particular variable v singled out as special. This special variable v is the one whose free occurrences we will replace with x should the operation described in (47) require us to. Likewise for $[w]Q$.

So for example, consider the command:

$$\textbf{case}(x, [z]\lambda u(uz)[z], [v]v)$$

If x is the term $\langle 1, \overline{x}\rangle$, then this produces the result $\lambda u(u\overline{x})$. If x is the term $\langle 2, \overline{x}\rangle$, this produces the result $\overline{x}$.

In the case in which u does not occur free in P (for example, P is a closed term) and v does not occur free in Q, then the command

$$\textbf{case}(x, [u]P, [v]Q)$$

just tells us to produce the output P if the label attached to x is 1, and the output Q if the label attached to x is 2. So for example, the command

$$\textbf{case}(x, [v]\lambda u(u), [v]\lambda w(ww))$$

outputs $\lambda u(u)$ if the label attached to x is 1, and $\lambda w(ww)$ if the label attached to x is 2. This sort of command mimics the **if-then-else** command structures found in many programming languages.

Note that once we have access to these sorts of **case** commands, we have no reason to care about the precise mechanics by which the 'label' 1 or 2 is attached to an element of a sum type

$A + B$. As long as we have the ability to execute one of a given pair of commands in the case that the label is 1 and the other in case the label is 2, we can tell what label occurs in x simply by looking at the outcome of $\mathbf{case}(x, [u]P, [v]Q)$ for appropriately chosen P and Q. There is no need to talk about the mechanics of label attachment beyond that.

To see how this works in full detail, it will also be convenient to introduce an operator $\mathbf{in}_1$ that takes a λ-term x to x with the label 1 attached, as well as an operator $\mathbf{in}_2$ that takes the λ-term x to x with the label 2 attached. Again, the precise mechanism by which labels are being attached will not matter. Although we can think of $\mathbf{in}_1(x)$ as $\langle 1, x\rangle$ and $\mathbf{in}_2(x)$ as $\langle 2, x\rangle$, it is more clean to simply regard $\mathbf{in}_1(x)$ and $\mathbf{in}_2(x)$ as basic expressions that attach labels to x, without the need to specify the precise mechanics by which they do so.

In addition to **1.** and **2.** above, we then have the following new formation rules for untyped λ-terms:

3. If x is a λ-term, then $\mathbf{in}_1(x)$ and $\mathbf{in}_2(x)$ are λ-terms.
4. If x, P and Q are λ-terms and u, v variables, then $\mathbf{case}(x, [u]P, [v]Q)$ is a λ-term.

So for example, in this expanded version of the λ-calculus we have terms like the following:

$\mathbf{in}_1(z)$
$\langle \mathbf{in}_1(z), x\rangle$
$\langle \mathbf{in}_1(z), x\rangle\ \pi_1(y)$
$\lambda x(\langle \mathbf{in}_1 z, x\rangle\ \pi_1(y))$
$\mathbf{case}(\lambda x(\langle \mathbf{in}_1 z, x\rangle\ \pi_1(y)), [u]\pi_2(u), [w]\langle w, v\rangle)$

As always, we omit parentheses where no confusion results.

The free variables of $\mathbf{in}_1(x)$ and $\mathbf{in}_2(x)$ are just the free variables of x. The free variables of $\mathbf{case}(x, [u]P, [v]Q)$ are all the free variables of x, all the free variables of P except for u, and all the free variables of Q except for v. So in the expression

$$\mathbf{case}(\lambda x(\langle \mathbf{in}_1 z, x\rangle\ \pi_1(y)), [u]\pi_2(u), [w]\langle w, v\rangle)$$

you should be able to verify that the free variables are z, y and v.

Note in particular that any free occurrences of v in P become bound in the expression $[v]P$. In this way, $[v]$ is a 'binding operator' that, much like λ-abstraction, takes a free variable and renders it bound.

The obvious β-reduction rules to add at this point are

3. $\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]Q) \to_\beta P[R/u]$
4. $\mathbf{case}(\mathbf{in}_2(R), [u]P, [v]Q) \to_\beta Q[R/v]$

So for example, the terms

$$\mathbf{case}(\lambda x(\mathbf{in}_1(x))(y), [z]z, [y]\lambda z(y))$$

and

$$\lambda x(\mathbf{case}(x, [u]\pi_2(u), [v]\pi_1(v)))\ \mathbf{in}_2(\langle y, z\rangle)$$

may be β-reduced as follows:

$$\begin{aligned}\mathbf{case}(\lambda x(\mathbf{in}_1(x))(y), [z]z, [y]\lambda z(y)) &\to_\beta \mathbf{case}(\mathbf{in}_1(y), [z]z, [y]\lambda z(y))\\ &\to_\beta z[\mathbf{in}_1(y)/z]\\ &= \mathbf{in}_1(y)\end{aligned}$$

$$\begin{aligned}\lambda x(\mathbf{case}(x, [u]\pi_2(u), [v]\pi_1(v)))\ \mathbf{in}_2(\langle y, z\rangle) &\to_\beta \mathbf{case}(\mathbf{in}_2(\langle y, z\rangle), [u]\pi_2(u), [v]\pi_1(v))\\ &\to_\beta \pi_1(v)[\langle y, z\rangle/v]\\ &= \pi_1(\langle y, z\rangle)\\ &\to_\beta y\end{aligned}$$

Next, let us think about what sorts of types we should assign λ-terms of the form $\mathbf{in}_1(x)$, $\mathbf{in}_2(x)$, and $\mathbf{case}(x, P[v_1], Q[v_2])$. We begin by thinking about the typing rules for $\mathbf{in}_1(x)$ and $\mathbf{in}_2(x)$. Given types σ and τ and an object x of type σ, we use $\mathbf{in}_1(x)$ to attach a label to x indicating that it has type τ, thereby creating an object of the sum type $\sigma + \tau$. With this in mind, one might suspect that the following are the typing rules for $\mathbf{in}_1(x)$ and $\mathbf{in}_2(x)$:

$$\frac{\Gamma \vdash x : \sigma}{\Gamma \vdash \mathbf{in}_1(x) : \sigma + \tau} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash \mathbf{in}_2(x) : \sigma + \tau} \tag{48}$$

However, these rules cannot be quite right. Focusing on the leftmost rule, given an x of type σ, for *which* type τ is $\mathbf{in}_1(x)$ of type $\sigma + \tau$? The type τ simply appears out of nowhere in the conclusion of this rule. So if x is of type $\mathbb{N}$, is $\mathbf{in}_1(x)$ of type $\mathbb{N} + \mathbb{N}$, or $\mathbb{N} + \mathbb{B}$, or $\mathbb{N} + (\mathbb{N} \to \mathbb{N})$? We want every introduced expression to have a definite type. There must therefore be a determinate answer to the question of what type $\mathbf{in}_1(x)$ has.

To resolve this ambiguity, in the context of typed computations we add superscripts to the notation $\mathbf{in}_1(x)$, and write $\mathbf{in}_1^{\sigma,\tau}$ to indicate that this is an operation that takes an object of type σ and produces an output of type $\sigma + \tau$. Likewise, $\mathbf{in}_2^{\sigma,\tau}$ is an operation that takes an object of type τ and produces an output of type $\sigma + \tau$. In the context of the untyped λ-calculus, we do not need these superscripts, and can take at face value the rule that if x is a λ-term, then $\mathbf{in}_1(x)$ and $\mathbf{in}_2(x)$ are λ-terms. In the context of the typed λ-calculus, however, we must regard not only all variables as typed, but also all expressions of the form $\mathbf{in}_1$ and $\mathbf{in}_2$ as typed. So we will *not* regard the term $\mathbf{in}_1(x^{\mathbb{N}})$ as fully typed. By contrast, the term $\mathbf{in}_1^{\mathbb{N},\mathbb{B}}(x^{\mathbb{N}})$ will be regarded as fully typed, and moreover has the type $\mathbb{N} + \mathbb{B}$.

The correct typing rules for $\mathbf{in}_1(x)$ and $\mathbf{in}_2(x)$ are thus:

$$\frac{\Gamma \vdash x : \sigma}{\Gamma \vdash \mathbf{in}_1^{\sigma,\tau}(x) : \sigma + \tau} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash \mathbf{in}_2^{\sigma,\tau}(x) : \sigma + \tau} \tag{49}$$

Having said all this, in the context of a typing derivation the superscripts on $\mathbf{in}_1$ and $\mathbf{in}_2$ can typically be determined from context. So for example, in the leftmost of the rules (48), the only

superscripts that could be attached to $\mathbf{in}_1$ to render the inference grammatical would be the superscripts σ, τ. When the superscripts on $\mathbf{in}_1$ or $\mathbf{in}_2$ can be uniquely determined by context, we typically omit them for the sake of compact notation. In such cases, the superscripts should still be understood as present, but merely 'invisible'. In cases in which there is any ambiguity or possibility of confusion, we include all superscripts. With these conventions is mind, our original rules (48) are perfectly acceptable; but without these conventions in mind, they do not make sense.

We now turn to the question of what type we should assign $\mathbf{case}(x, [u]P, [v]Q)$. The problem is that in general this term can get two different types. Suppose, for example, that whenever z has type σ, $P[z/u]$ has type ρ_1. And suppose that whenever z has type τ, $Q[z/v]$ has type ρ_2. Now suppose that x is an element of type $\sigma + \tau$. Then depending on whether x is an object z of type σ with the label 1 attached to it or an object z of type τ with the label 2 attached to it, $\mathbf{case}(x, [u]P, [v]Q)$ will either have type ρ_1 or type ρ_2. And so it looks like we cannot make a definitive statement in advance about which type $\mathbf{case}(x, [u]P, [v]Q)$ will receive.

Suppose however that ρ_1 and ρ_2 (i.e., the type of $P[z/u]$ if z has type σ, and the type of $Q[z/v]$ if z has type τ) are actually the same type ρ. In this case, we *can* definitively say that $\mathbf{case}(x, [u]P, [v]Q)$ gets the type ρ. This suggests the following typing rule

$$\frac{\Gamma \vdash x : \sigma + \tau \qquad \Gamma, u : \sigma \vdash P : \rho \qquad \Gamma, v : \tau \vdash Q : \rho}{\Gamma \vdash \mathbf{case}(x, [u]P, [v]Q) : \rho} \tag{50}$$

The rules (49) and (50) capture the defining properties of sum types. Note that they do so without presupposing anything about the mechanism by which labels are attached to elements in sum types. From a type theoretic point of view, directs sum types are then typically thought of as defined by these typing rules, rather than in terms of ordered pairs.

As before, armed with these rules we can type terms of the untyped λ-calculus. For example, consider the term

$$\lambda x(\mathbf{case}(x, [y]\mathbf{in}_1(y), [z]\mathbf{in}_2(z)))$$

This can be assigned the type $(A{+}B) \to (A{+}B)$ for any A and B, as the following typing derivation shows:

$$\dfrac{\dfrac{x : A{+}B \vdash x : A{+}B \qquad \dfrac{x : A{+}B,\ y : A \vdash y : A}{x : A{+}B,\ y : A \vdash \mathbf{in}_1(y) : A{+}B} \qquad \dfrac{x : A{+}B,\ z : B \vdash z : B}{x : A{+}B,\ z : A \vdash \mathbf{in}_2(z) : A{+}B}}{x : A{+}B \vdash \mathbf{case}(x, [y]\mathbf{in}_1(y), [z]\mathbf{in}_2(z))) : A{+}B}}{\vdash \lambda x(\mathbf{case}(x, [y]\mathbf{in}_1(y), [z]\mathbf{in}_2(z)))) : (A{+}B) \to (A{+}B)}$$

In this typing derivation, $\mathbf{in}_1$ should of course be understood as $\mathbf{in}_1^{A,B}$ throughout, and $\mathbf{in}_2$ should be understood as $\mathbf{in}_2^{A,B}$ throughout. Consider next the open λ-term

$$\mathbf{case}(x, [y]\langle y, u\rangle, [z]\langle u, z\rangle)$$

This λ-term has x and u as free variables. If x has type $A{+}A$ and u has type A, then this λ-term can be assigned type $A{\times}A$, as the following typing derivation sketch shows:

$$\dfrac{x : A{+}A,\ u : A \vdash x : A + A \qquad \begin{array}{c}\vdots\\ x : A{+}A,\ u : A,\ y : A \vdash \langle y, u\rangle : A{\times}A\end{array} \qquad \begin{array}{c}\vdots\\ x : A{+}A,\ u : A,\ z : A \vdash \langle u, z\rangle : A{\times}A\end{array}}{x : A{+}A,\ u : A \vdash \mathbf{case}(x, [y]\langle y, u\rangle, [z]\langle u, z\rangle) : A{\times}A}$$

You should easily be able to reconstruct the omitted upper parts $\vdots$ of this typing derivation. Note also that the conclusion of this typing derivation contains the context $x : A{+}A,\ u : A$ declaring the type of the free variables x and u of $\mathbf{case}(x, [y]\langle y, u\rangle, [z]\langle u, z\rangle)$.

Exercises for Section 2.10

1. Write each of the following λ-terms in β-normal form.
(a) $\mathbf{case}(\mathbf{in}_1(x), \lambda z(y)[y], z[z])$
(b) $\lambda v(\mathbf{case}(v, [y]y, [z]\lambda x(z)(x)))(\mathbf{in}_2(x))$
(c) $\lambda v(\mathbf{case}(v(z), [z]\pi_1(z), [z]\pi_2(z)))(\lambda x(\mathbf{in}_2(\langle x, y\rangle)))$

2. Each of the following untyped λ-terms can be assigned types. Find types for them, and confirm your result by producing an appropriate typing derivation. If the λ-term is not closed, the conclusion of your typing derivation will of course have a non-empty context.
(a) $\lambda x(\mathbf{case}(x, [y]\pi_1(y), [z]z))$
(b) $\mathbf{case}(x, [y]\langle y, u\rangle, [z]\langle v, z\rangle)$
(c) $\mathbf{case}(x, [u]u(y), [v]z(v))$

2.11 Adding Types: The Empty Type

In type theory, it is also useful to sometimes be able to refer to the *empty* type, which we denote 0. The idea is that no expression should have this type. We express this somewhat modestly with the idea that if a term has type 0, then from this term we can construct an object of any type we please. In particular, given an x of type 0 and a type A, we can construct a term of type A, which we call $\mathbf{abs}^A(x)$ (the 'absurd' object of type A.) We thus add the following rule for λ-term formation in the untyped λ-calculus:

5. If x is a λ-term, then so is $\mathbf{abs}(x)$.

and the typing rule:

$$\dfrac{\Gamma \vdash x : 0}{\Gamma \vdash \mathbf{abs}^A(x) : A}$$

Because λ-terms must have determinate types in the typed λ-calculus, it is necessary to add a superscript A to $\mathbf{abs}(x)$, just as it was necessary to add superscripts to the **in** operators. However, just as with the **in** operators we will omit superscripts when they are obvious from the context. We may thus rewrite our rule as

$$\frac{\Gamma \vdash x : 0}{\Gamma \vdash \mathbf{abs}(x) : A}$$

The free variables of $\mathbf{abs}(x)$ are just the free variables of x. There is no corresponding new rule of β-reduction to be introduced.

Using this expanded λ-calculus and new typing rule, we can for example construct a λ-term that can be given the type $(A\&(A \to 0)) \to B$ for any types A and B as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{x : A\&(A \to 0) \vdash x : A\&(A \to 0)}{x : A\&(A \to 0) \vdash \pi_2(x) : A \to 0} \quad \dfrac{x : A\&(A \to 0) \vdash x : A\&(A \to 0)}{x : A\&(A \to 0) \vdash \pi_1(x) : A}}{x : A\&(A \to 0) \vdash \pi_2(x)(\pi_1(x)) : 0}}{x : A\&(A \to 0) \vdash \mathbf{abs}(\pi_2(x)(\pi_1(x))) : B}}{\vdash \lambda x(\mathbf{abs}(\pi_2(x)(\pi_1(x)))) : (A\&(A \to 0)) \to B}$$

In this derivation, $\mathbf{abs}$ is of course to be understood as $\mathbf{abs}^B$.

Exercises for Section 2.11

1. Each of the following untyped λ-terms can be assigned types. Find types for them, and confirm your result by producing an appropriate typing derivation.
(a) $\lambda x(\mathbf{abs}(\pi_1(x)))$
(b) $\lambda x(\mathbf{abs}(\mathbf{case}(x, y[y], \pi_1(z)[z])$

2.12 η Rules for Product and Sum Types.

In addition to the new rules of β-reduction we have introduced for product and sum types, there are also new rules of η-reduction appropriate for product and sum types. We introduce these in this section.

The primary way of constructing functions in the λ-calculus is by λ-abstraction over a λ-term. So given a λ-term t, we can abstract over it to obtain a function $\lambda x(t)$. But consider an object f of some function type $A \to B$. Is there anything that guarantees that f is obtained by abstracting over some λ-term? Identifying f with $\lambda x(f(x))$ (where x is a variable not free in f) tells us that f is indeed an abstraction over the term $f(x)$ of type B with respect to a variable x of type A. So the rule of η-reduction

$$\lambda x(t(x)) \to_\eta t$$

(where x is not free in t) tells us that *all* functions in the λ-calculus are abstractions over terms. More precisely, it allows us to say that every object f of type $A \to B$ is η-equivalent to a λ-abstraction over a λ-term – that is, f is η-equivalent to $\lambda x(f(x))$ (where we select a variable x that is not free in f.)

Likewise, the primary way of constructing objects of a product type $A \times B$ is by taking an object t_1 of type A and an object t_2 of type B and forming the object $\langle t_1, t_2 \rangle$. But is there anything

that guarantees that any object t of type $A \times B$ has this form? It is natural to identify any t of type $A \times B$ with $\langle \pi_1(t), \pi_2(t) \rangle$. We therefore introduce the new rule of η-reduction

$$\langle \pi_1(t), \pi_2(t) \rangle \to_\eta t.$$

This rule guarantees that every object of type $A \times B$ can be written in the form $\langle t_1, t_2 \rangle$. More precisely, it allows us to say that every object t of type $A \times B$ is η-equivalent to an expicit ordered pair – namely, $\langle \pi_1(t), \pi_2(t) \rangle$.

Turning to sum types, the primary way of constructing objects of a sum type $A + B$ is either by taking an object x of type A and forming $\mathbf{in}_1(x)$, or by taking an object y of type B and forming $\mathbf{in}_2(y)$. (This case is trickier than the previous two in that we have multiple options here.) But is there anything that guarantees that any object t of type $A + B$ must have one of these two forms? Note that for any t of type $A + B$, it is natural to identify t with

$$\mathbf{case}(t, [u]\mathbf{in}_1(u), [u]\mathbf{in}_2(u)). \tag{51}$$

To see why, we evaluate the term (51), considering each case individually. First, if t is of the form $\mathbf{in}_1(z)$, then the whole term simplifies to $\mathbf{in}_1(u)[z/u]$, which is $\mathbf{in}_1(z)$, and so the final result is t itself. Likewise, if t is of the form $\mathbf{in}_2(z)$, then the whole term simplifies to $\mathbf{in}_2(u)[z/u]$, which is $\mathbf{in}_2(z)$, and so the final result is again t itself. So either way, the term (51) simplifies to t.

Because a term of the form $\mathbf{case}(t, [u]s_1(u), [v]s_2(v))$ has the form $s_1(z)$ or $s_2(z)$ for some z (assuming that u does not appear free in s_1 or s_2), identifying t with (51) then tells us that t has either the form $\mathbf{in}_1(z)$ or $\mathbf{in}_2(z)$ for some z, as desired. It is thus natural to introduce the new rule of η-reduction

$$\mathbf{case}(t, [u]\mathbf{in}_1(u), [v]\mathbf{in}_2(v)) \to_\eta t. \tag{52}$$

This rule guarantees that every object of type $A + B$ has either the form $\mathbf{in}_1(t)$ or $\mathbf{in}_2(t)$.

Our η-rules tell us that objects of complex types have *canonical forms*: every element of a function type $A \to B$ has the form $\lambda x(t)$, every element of a product type $A \times B$ has the form $\langle t_1, t_2 \rangle$, and every element of a sum type $A + B$ has the form $\mathbf{in}_1(t)$ or $\mathbf{in}_2(t)$. Later on when we have a notion of equality in our theory we will return to the question of the meaning of the η-rules, but the present considerations will suffice for now.

One might be tempted to generalize (52) somewhat, and write

$$\mathbf{case}(t, [u]s[\mathbf{in}_1(u)/z], [v]s[\mathbf{in}_2(v)/z]) \to_\eta s[t/z] \tag{53}$$

where s, t are λ-terms and u, v, z variables such that u and v are not free in s. Note that (52) may be obtained from (53) by letting s be the term z. This rule looks entirely reasonable. Moreover, it has the virtue of allowing us to see a sense in which λ-terms such as

$$R(\mathbf{case}(z, [x]P, [y]Q)) \quad \text{and} \quad \mathbf{case}(z, [x]R(P), [y]R(Q))$$

are equivalent (where P, Q and R are λ-terms and x, y, z are variables such that neither x nor y are free in R and z is not free in P or Q.) That these are equivalent in some sense is intuitive

– it should not matter whether we divide into cases, apply P or Q, and then input the result into R, or whether we divide into cases, then apply $R(P)$ or $R(Q)$. To see this more formally, substitute $R(\mathbf{case}(z, [x]P, [y]Q))$ for s in (53), where u and v are assumed to not appear free in $R(\mathbf{case}(z, [x]P, [y]Q))$. This gives the expression

$$\mathbf{case}(t, [u]R(\mathbf{case}(z, [x]P, [y]Q))[\mathbf{in}_1(u)/z], [v]R(\mathbf{case}(z, [x]P, [y]Q))[\mathbf{in}_2(v)/z]) \tag{54}$$

Now, because z does not appear free in P or Q, the expression $R(\mathbf{case}(z, [x]P, [y]Q))[\mathbf{in}_1(u)/z]$ is $R(\mathbf{case}(\mathbf{in}_1(u), [x]P, [y]Q))$, which β-reduces to $R(P[u/x])$. Because x is not free in R, this is $R(P)[u/x]$. Likewise, $s[\mathbf{in}_2(v)/z]$ β-reduces to $R(Q)[v/y]$. Thus (54) β-reduces to

$$\mathbf{case}(z, [u]R(P)[u/x], [v]R(Q)[v/y]).$$

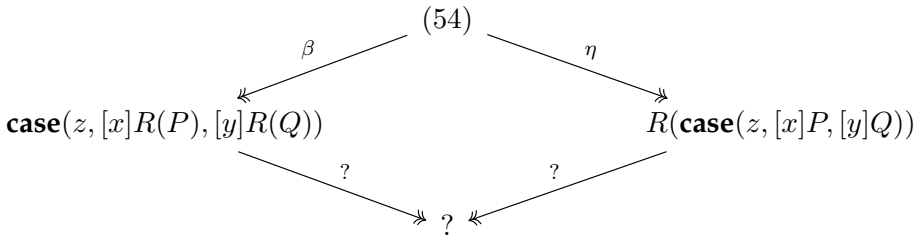
Renaming bound variables, we can rewrite this more simply as

$$\mathbf{case}(z, [x]R(P), [y]R(Q)). \tag{55}$$

But on the other hand, using the rule (53), the expression (54) η-reduces to $R(\mathbf{case}(z, [x]P, [y]Q))$. Because the same expression both β-reduces to $\mathbf{case}(z, [x]R(P), [y]R(Q))$ and also η-reduces to $R(\mathbf{case}(z, [x]P, [y]Q))$, we then have

$$R(\mathbf{case}(z, [x]P, [y]Q)) \equiv_{\beta\eta} \mathbf{case}(z, [x]R(P), [y]R(Q)).$$

On one had, this is a pleasing result. On the other hand, without further modifications this leads to a violation of the Church-Rosser property for $\beta\eta$-reductions, as there is no obvious way to complete the 'diamond'



While different approaches are possible to rectify this, the approach we will take is to prohbit the η-reduction rule (53). In some texts, rules such as (53) as re-classified as rules of 'commutation' , along with reduction rules like

$$\mathbf{case}(t, [u]P, [v]Q)(R) \to \mathbf{case}(t, [u]P(R), [v]Q(R)))$$
$$\lambda x(\mathbf{case}(t, [u]P, [v]Q)) \to \mathbf{case}(t, [u]\lambda x(P), [v]\lambda x(Q))$$

and so on (with appropriate restrictions on free variables.) In general, such rules allow us to move certain expressions in and out of binding operators. Many tricky issues arise here, which we do not pursue further. For our purposes, we allow only the η-reduction rules:

$$\lambda x(t(x)) \to_\eta t$$

$$\langle \pi_1(t), \pi_2(t) \rangle \to_\eta t$$
$$\textbf{case}(t, [u]\textbf{in}_1(u), [v]\textbf{in}_2(v)) \to_\eta t$$

and no others. More formally, we have the following definitions

Definition 2.18: η-reduction, $\beta\eta$-reduction, and $\beta\eta$-equivalence

(a) $A \to_\eta B$ iff for some expressions E and F

(i) A has the form $E\ \lambda x(t(x))\ F$ and B has the form $E\ t\ F$ (where x is not free in t), or
(ii) A has the form $E\ \langle \pi_1(t), \pi_2(t) \rangle\ F$ and B has the form $E\ t\ F$, or
(iii) A has the form $E\ \textbf{case}(t, [u]\textbf{in}_1(u), [v]\textbf{in}_2(v))\ F$ and B has the form $E\ t\ F$.

(b) $A \to_{\beta\eta} B$ iff $A \to_\beta B$ or $A \to_\eta B$.

(c) $A \twoheadrightarrow_{\beta\eta} B$ (i.e., A $\beta\eta$-reduces to B) iff for some sequence $X_0, X_1, ..., X_n$ of λ-terms we have that $A = X_0$, $B = X_n$, and $X_0 \to_{\beta\eta} X_1 ... \to_{\beta\eta} X_n$. This includes the degenerate case in which A and B are identical.

(d) X and Y are *$\beta\eta$-equivalent* (in symbols, $X =_{\beta\eta} Y$) iff one of the following holds:

(i) X and Y are the same term (up to α-equivalence), or
(ii) for some sequence of terms $X_0, X_1, ..., X_{n-1}, X_n$ with $U = X_0$ and $V = X_n$,

$$X_0 \sim_{\beta\eta} X_1,\ X_1 \sim_{\beta\eta} X_2, ...,\ X_{n-1} \sim_{\beta\eta} X_n.$$

2.13 Collecting Things Together

We have added much machinery to the untyped λ-calculus λ_0, creating a more sophisticated untyped calculus with many rules. For convenience, we present these rules together.

Definition 2.19: Rules of the untyped λ-calculus λ_1.

Rules of λ-term formation

(i) Any variable $x, y, z, ...$ is a λ-term.
(ii) If M and N are λ-terms, then so is $M(N)$.
(iii) If M is a λ-term and x a variable, then $\lambda x(M)$ is a λ-term.
(iv) If x and y are λ-terms, then so is $\langle x, y \rangle$.
(v) If x is a λ-term, then so are $\pi_1(x)$ and $\pi_2(x)$.
(vi) If x is a λ-term, then so are $\textbf{in}_1(x)$ and $\textbf{in}_2(x)$.
(vii) If R, P and Q are λ-terms and u, v variables, then $\textbf{case}(R, [u]P, [v]Q)$ is a λ-term.

(viii) If x is a λ-term, then so is $\mathbf{abs}(x)$.

Rules of β-reduction

(i) $\lambda x(M)(N) \to_\beta M[N/x]$
(ii) $\pi_1\langle M, N\rangle \to_\beta M$
(iii) $\pi_2\langle M, N\rangle \to_\beta N$
(iv) $\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]Q) \to_\beta P[R/u]$
(v) $\mathbf{case}(\mathbf{in}_2(R), [u]P, [v]Q) \to_\beta Q[R/v]$

Rules of η-reduction

(i) $\lambda x(M(x)) \to_\eta M$ where x is not free in M
(ii) $\langle \pi_1(M), \pi_2(M)\rangle \to_\eta M$
(iii) $\mathbf{case}(M, [u]\mathbf{in}_1(u), [v]\mathbf{in}_2(v)) \to_\eta M$

We call this expanded version of the untyped λ-calculus λ_1.

To the untyped λ-calculus λ_1, we append the following set of typing rules:

Definition 2.20: Rules of the typed λ-calculus $\mathbf{TR}_1$.

(i) $\Gamma, v : \tau \vdash v : \tau$

(ii) $$\frac{\Gamma \vdash x : \tau \to \sigma \qquad \Gamma \vdash y : \tau}{\Gamma \vdash x(y) : \sigma}$$

(iii) $$\frac{\Gamma, v : \tau \vdash x : \sigma}{\Gamma \vdash \lambda v(x) : \tau \to \sigma}$$

(iv) $$\frac{\Gamma \vdash x : \sigma \qquad \Gamma \vdash y : \tau}{\Gamma \vdash \langle x, y\rangle : \sigma \times \tau}$$

(v) $$\frac{\Gamma \vdash x : \sigma \times \tau}{\Gamma \vdash \pi_1(x) : \sigma} \qquad \frac{\Gamma \vdash x : \sigma \times \tau}{\Gamma \vdash \pi_2(x) : \tau}$$

(vi) $$\frac{\Gamma \vdash x : \sigma}{\Gamma \vdash \mathbf{in}_1(x) : \sigma + \tau} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash \mathbf{in}_2(x) : \sigma + \tau}$$

(vii) $$\frac{\Gamma \vdash x : \sigma + \tau \qquad \Gamma, u : \sigma \vdash y : \rho \qquad \Gamma, v : \tau \vdash z : \rho}{\Gamma \vdash \mathbf{case}(x, [u]y, [v]z) : \rho}$$

(viii) $$\frac{\Gamma \vdash x : 0}{\Gamma \vdash \mathbf{abs}(x) : \tau}$$

where v, v_1, v_2 are variables, x, y, z are terms of λ_1, and σ, τ, ρ are types.

This system of typing rules is sometimes called $\lambda(I)$. We will refer to it as $\mathbf{TR}_1$. It is an expansion of the smaller typing system $\lambda(\to)$ (i.e., $\mathbf{TR}_0$) considered earlier.

As before, there is a bound-typed version of this set of typing rules

Definition 2.21: Rules of the typed λ-calculus $\mathbf{TR}_1^{\mathbf{bt}}$.

(i) $\Gamma, v : \tau \vdash v : \tau$

(ii) $$\frac{\Gamma \vdash x : \tau \to \sigma \qquad \Gamma \vdash y : \tau}{\Gamma \vdash x(y) : \sigma}$$

(iii) $$\frac{\Gamma, v : \tau \vdash x : \sigma}{\Gamma \vdash \lambda v^\tau(x) : \tau \to \sigma}$$

(iv) $$\frac{\Gamma \vdash x : \sigma \qquad \Gamma \vdash y : \tau}{\Gamma \vdash \langle x, y\rangle : \sigma \times \tau}$$

(v) $$\frac{\Gamma \vdash x : \sigma \times \tau}{\Gamma \vdash \pi_1(x) : \sigma} \qquad \frac{\Gamma \vdash x : \sigma \times \tau}{\Gamma \vdash \pi_2(x) : \tau}$$

(vi) $$\frac{\Gamma \vdash x : \sigma}{\Gamma \vdash \mathbf{in}_1(x) : \sigma + \tau} \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash \mathbf{in}_2(x) : \sigma + \tau}$$

(vii) $$\frac{\Gamma \vdash x : \sigma + \tau \qquad \Gamma, u : \sigma \vdash y : \rho \qquad \Gamma, v : \tau \vdash z : \rho}{\Gamma \vdash \mathbf{case}(x, [u^\sigma]y, [v^\tau]z) : \rho}$$

(viii) $$\frac{\Gamma \vdash x : 0}{\Gamma \vdash \mathbf{abs}(x) : \tau}$$

where v, v_1, v_2 are variables, x, y, z are terms of λ_1, and σ, τ, ρ are types.

There is also a corresponding generalization of the Church-style λ-calculus:

Definition 2.22: Rules of the typed λ-calculus $\lambda_1^{\mathbf{type}}$.

Types

The *propositional types* are those that may be obtained from the fundamental types $\tau_1, \tau_2, ...$ and the empty type 0 under the operations taking two types A and B to the types $A \to B$, $A \times B$ and $A + B$.

Terms

The set of *simply typed λ-terms* are defined inductively as follows:

(i) For any variable v and any type τ, v^τ is a λ-term of type τ.
(ii) If $[M]^{\alpha\to\beta}$ and $[N]^\alpha$ are λ-terms, then $[[M]^{\alpha\to\beta}([N]^\alpha)]^\beta$ is a λ-term.
(iii) If $[M]^\beta$ is a λ-term and x^α a variable, then $[\lambda x^\alpha([M]^\beta)]^{\alpha\to\beta}$ is a λ-term.
(iv) If $[M]^\alpha$, $[N]^\beta$ are λ-terms, then $[\langle [M]^\alpha, [N]^\beta\rangle]^{\alpha\times\beta}$ is a λ-term.
(v) If $[M]^{\alpha\times\beta}$ is a λ-term, then $[\pi_1([M]^{\alpha\times\beta})]^\alpha$ and $[\pi_2([M]^{\alpha\times\beta})]^\beta$ are λ-terms.
(vi) If $[M]^\alpha$ and $[N]^\beta$ are λ-terms, then $[\mathbf{in}_1^{\alpha,\beta}([M]^\alpha)]^{\alpha+\beta}$ and $[\mathbf{in}_2^{\alpha,\beta}([N]^\beta)]^{\alpha+\beta}$ are λ-terms.
(vii) If $[R]^{\alpha+\beta}$, $[P]^\gamma$, and $[Q]^\gamma$ are λ-terms and u^α, v^β variables, then

$[\mathbf{case}([R]^{\alpha+\beta}, [u^\alpha][P]^\gamma, [v^\beta][Q]^\gamma)]^\gamma$ is a λ-term.

(viii) If $[M]^0$ is a λ-term, then so is $[\mathbf{abs}^\alpha([M]^0)]^\alpha$.

Rules of β-reduction

(i) $\lambda x^\sigma([M]^\tau)([N]^\sigma) \to_\beta [M]^\tau[[N]^\sigma/x^\sigma]$
(ii) $\pi_1(\langle [M]^\sigma, [N]^\tau\rangle^{\sigma\times\tau}) \to_\beta [M]^\sigma$
(iii) $\pi_2(\langle [M]^\sigma, [N]^\tau\rangle^{\sigma\times\tau}) \to_\beta [N]^\tau$
(iv) $\mathbf{case}(\mathbf{in}_1^{\alpha,\beta}([S]^\alpha)^{\alpha+\beta}, [u^\alpha][P]^\gamma, [v^\beta][Q]^\gamma) \to_\beta [P]^\gamma[[S]^\alpha/u^\alpha]$
(v) $\mathbf{case}(\mathbf{in}_2^{\alpha,\beta}([S]^\beta)^{\alpha+\beta}, [u^\alpha][P]^\gamma, [v^\beta][Q]^\gamma) \to_\beta [Q]^\gamma[[S]^\beta/v^\beta]$

Rules of η-reduction

(i) $\lambda x^\sigma([M]^{\sigma\to\tau}(x^\sigma)) \to_\eta [M]^{\sigma\to\tau}$ where x^σ is not free in M
(ii) $\langle \pi_1([M]^{\sigma\times\tau}), \pi_2([M]^{\sigma\times\tau})\rangle \to_\eta [M]^{\sigma\times\tau}$
(iii) $\mathbf{case}([M]^{\sigma+\tau}, [u^\sigma]\mathbf{in}_1^{\sigma,\tau}(u^\sigma), [v^\tau]\mathbf{in}_2^{\sigma,\tau}(v^\tau)) \to_\eta [M]^{\sigma+\tau}$

We call this version of the Church-style typed λ-calculus λ_1^{type}. Although in this definition we have written everything out in full detail, as with λ_0^{type} we will in practice omit brackets and types for clarity when they can be easily reconstructed. In particular, as before it will suffice to indicate the types of bound variables when they are first introduced, as well as the types of free variables. Note however that in our expanded version of the λ-calculus we now have *two* sources of bound variables. First, there are variables connected with λ-operators, as before. Second, we have variables that are bound within a **case** term. Recall that in

$$\mathbf{case}(x, [u]P, [v]Q),$$

$[u]$ and $[v]$ are regarded as 'binding operators' much like λ, and their scopes are all of P and Q respectively. Thus, if we are to type all bound variables at the moment the relevant binding operators are introduced, we must type the variables $[u]$ and $[v]$ in $\mathbf{case}(x, [u]P, [v]Q)$. For example, consider the open λ_1-term

$$\lambda y\ \mathbf{case}(x, [u]u, [v]vy)$$

We can type all the bound variables of this expression as follows:

$$\lambda y^B\mathbf{case}(x, [u^A]u, [v^{B\to A}]vy)$$

The free variable x obviously must have type $A + (B \to A)$ (because in $\mathbf{case}(x, [u]P, [v]Q)$ the type of x must be the sum of the types of u and v), and thus we may regard this term as a fully typed λ_1^{type}-term – explicitly writing down the type of the free variable x is in this case optional. However, where it does not produce excessive clutter, for maximum clarity we generally indicate the type of each free variable explicitly, even when it is otherwise deducible. So for the term in question we will generally write

$$\lambda y^B\mathbf{case}(x^{A+(B\to A)}, [u^A]u, [v^{B\to A}]vy)$$

That this is grammatically typed is shown by the following typing derivation of $\mathbf{TR}_1$

$$\dfrac{\dfrac{x{:}A{+}(B{\to}A), y{:}B \vdash x{:}A{+}(B{\to}A) \qquad x{:}A{+}(B{\to}A), y{:}B, u{:}A \vdash u{:}A \qquad x{:}A{+}(B{\to}A), y{:}B, v{:}B \to A \vdash \vdots\, vy{:}A}{x{:}A{+}(B{\to}A), y{:}B \vdash \mathbf{case}(x, [u]u, [v]vy){:}A}}{x{:}A{+}(B{\to}A) \vdash \lambda y\ \mathbf{case}(x, [u]u, [v]vy){:}B \to A}$$

or perhaps even more clearly by the following typing derivation of $\mathbf{TR}_1^{\mathbf{bt}}$

$$\dfrac{\dfrac{x{:}A{+}(B{\to}A), y{:}B \vdash x{:}A{+}(B{\to}A) \qquad x{:}A{+}(B{\to}A), y{:}B, u{:}A \vdash u{:}A \qquad x{:}A{+}(B{\to}A), y{:}B, v{:}B \to A \vdash \vdots\, vy{:}A}{x{:}A{+}(B{\to}A), y{:}B \vdash \mathbf{case}(x, [u^A]u, [v^{B\to A}]vy){:}A}}{x{:}A{+}(B{\to}A) \vdash \lambda y^B\ \mathbf{case}(x, [u^A]u, [v^{B\to A}]vy){:}B \to A}$$

For another example, consider the similar untyped λ_1-term

$$\lambda y\ \mathbf{case}(xz, [u]u, [v]vy)$$

as before, the bound variables may be typed as follows

$$\lambda y^B \mathbf{case}(xz, [u^A]u, [v^{B\to A}]vy)$$

However, knowing that xz has type $A + (B \to A)$ does not allow us to determine the types of the free variables x and z. In order for this term to be fully typed, we must therefore indicate the types of at least one of these terms explicitly. So for example, we regard the term

$$\lambda y^B \mathbf{case}(xz^D, [u^A]u, [v^{B\to A}]vy)$$

as a fully typed λ_1^{type}-term, as in this expression x must obviously have type $D \to (A + (B \to A))$. Just as before, for maximum clarity it is better to indicate the type of each free variable explicitly, in which case we write

$$\lambda y^B \mathbf{case}(x^{(A+(B\to A))\to D} z^D, [u^A]u, [v^{B\to A}]vy).$$

The reader is invited to write out the corresponding typing derivation in either $\mathbf{TR}_1$ or $\mathbf{TR}_1^{\mathbf{bt}}$.

Note that in addition to familiar examples of untypable terms such as $x(x)$, in λ_1 we now have a large class of new untypable terms. For example,

$$\pi_1(\lambda x(E))$$

cannot be typed for any λ-term E, because when fully typed, π_1 only acts on objects whose types have the form $A \times B$, while $\lambda x(E)$ always has a type of the form $A \to B$. In the exercises you will see examples of other terms in λ_1 that cannot be typed for similar reasons.

There are many results about λ_1, λ_1^{type} and $\mathbf{TR_1}$ that are straightforward generalizations of earlier theorems about λ_0, λ_0^{type} and $\mathbf{TR_0}$. In many cases, their proofs only involve only minor modifications of the corresponding proofs in the previous chapter. Occasionally, when a result about our expanded λ-calculus has a straightforward and essentially identical proof to that given

in the previous chapter, we simply use the result directly. (For example, the fact that every λ-term is α-equivalent to one that obeys the Baredregt variable convention remains true in λ_1, and the proof is for all intents and purposes identical to that of the corresponding fact about λ_0.) Doing so helps us to avoid having to relist a mass of theorems every time we expand the λ-calculus. We state some of the more important and less trivial generalizations here, in each leaving the proof to the the Appendix or the exercises.

Lemma 2.23:

For any λ_1-terms U, V, t and variable x, if $U \to_\beta V$ then $U[t/x] \twoheadrightarrow_\beta V[t/x]$, and if $U \to_\eta V$ then $U[t/x] \twoheadrightarrow_\eta V[t/x]$.

Theorem 2.24: The Church-Rosser Theorem for β- and $\beta\eta$-reductions in λ_1.

In the untyped λ-calculus λ_1, if $X \twoheadrightarrow_\beta Y_1$ and $X \twoheadrightarrow_\beta Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$. Likewise, if $X \twoheadrightarrow_{\beta\eta} Y_1$ and $X \twoheadrightarrow_{\beta\eta} Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_{\beta\eta} Z$ and $Y_2 \twoheadrightarrow_{\beta\eta} Z$.

Corollary 2.25

In λ_1, $U =_\beta V$ iff there is a term Z such that $U \twoheadrightarrow_\beta Z$ and $V \twoheadrightarrow_\beta Z$, and $U =_{\beta\eta} V$ iff there is a term Z such that $U \twoheadrightarrow_{\beta\eta} Z$ and $V \twoheadrightarrow_{\beta\eta} Z$.

Theorem 2.26: The Church-Rosser Theorem for β- and $\beta\eta$-reductions in $\lambda_1^{\mathbf{type}}$.

In the typed λ-calculus λ_1^{type}, if $X \twoheadrightarrow_\beta Y_1$ and $X \twoheadrightarrow_\beta Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$. Likewise, if $X \twoheadrightarrow_{\beta\eta} Y_1$ and $X \twoheadrightarrow_{\beta\eta} Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_{\beta\eta} Z$ and $Y_2 \twoheadrightarrow_{\beta\eta} Z$.

Corollary 2.27

In λ_1^{type}, $U =_\beta V$ iff there is a term Z such that $U \twoheadrightarrow_\beta Z$ and $V \twoheadrightarrow_\beta Z$, and $U =_{\beta\eta} V$ iff there is a term Z such that $U \twoheadrightarrow_{\beta\eta} Z$ and $V \twoheadrightarrow_{\beta\eta} Z$.

Theorem 2.28: Strong Normalizability of λ_1^{type} under β-reduction.

There is no infinite sequence $\tau_1, \tau_2, ...$ of terms in the typed λ-calculus λ_1^{type} for which

$$\tau_1 \to_\beta \tau_2 \to_\beta \tau_3 \to_\beta ...$$

Using η-reduction postponement (Lemma 2.34, to be discussed shortly), it can be shown as before that if a term has an infinite $\beta\eta$-reduction sequence, then it also has an infinite β-reduction sequence. (The analog for this in λ_0 was proved in the exercises for section 1.7, and the argument goes through unchanged.) The following is therefore a consequence of Theorem 2.28

Theorem 2.29: Strong Normalizability of λ_1^{type} under $\beta\eta$-reduction.

There is no infinite sequence $\tau_1, \tau_2, ...$ of terms in the typed λ-calculus λ_1^{type} for which

$$\tau_1 \to_{\beta\eta} \tau_2 \to_{\beta\eta} \tau_3 \to_{\beta\eta} ...$$

We also have

Theorem 2.30

Suppose E is a λ_1-term obeying the Barendregt variable convention, and that $\Gamma \vdash E : X$ in $\mathbf{TR_1}$. Then there is a proof of $\Gamma' \vdash E : X$ in $\mathbf{TR_1}$ for some $\Gamma' \subseteq \Gamma$ such that (i) there is no silent replacement of λ-terms by α-equivalent terms in this proof, (ii) the variables whose types are declared in Γ' are precisely the free variables of E, and (iii) all the elements of Γ' are distinct, and (iv) every variable declaration in the proof is either an element of Γ', or has the form $v : \tau$ where v is a bound variable of E.

Theorem 2.31: Soundness and Completeness of $\mathbf{TR_1}$.

Suppose E is an untyped λ_1-term whose free variables are $v_1, ..., v_n$, and let $\tau_1, ..., \tau_n$ be types. Then the following are equivalent:

(i) E can be typed into a λ_1^{type} term with overall type X in which the free variables $v_1, ..., v_n$ are assigned types $\tau_1, ..., \tau_n$ respectively.

(ii) There is a proof of $\Gamma \vdash E : X$ in $\mathbf{TR_1}$, where the context Γ includes the variable declarations $v_1 : \tau_1,\ v_2 : \tau_2, ...,\ v_n : \tau_n$.

Theorem 2.32: Soundness and Completeness of $\mathbf{TR_1^{bt}}$.

Suppose E is a bound-typed λ_1-term whose free variables are $v_1, ..., v_n$, and let $\tau_1, ..., \tau_n$ be types. Then the following are equivalent:

(i) It is possible to assign types of λ_1^{type} to all the variables of E in such a way that the free variables $v_1, ..., v_n$ of E are assigned types $\tau_1, ..., \tau_n$, and E itself has type X.

(ii) There is a proof of $\Gamma \vdash E : X$ in $\mathbf{TR_1^{bt}}$, where the context Γ includes the variable declarations $v_1 : \tau_1,\ v_2 : \tau_2, ...,\ v_n : \tau_n$.

Theorem 2.33: Subject Reduction Theorem for $\mathbf{TR_1}$.

In $\mathbf{TR_1}$, if $\Gamma \vdash M : \sigma$ and $M \twoheadrightarrow_{\beta\eta} N$, then $\Gamma \vdash N : \sigma$.

As in the previous chapter, many of these theorems have analogs for the bound-typed λ-calculus that we do not state separately, with the exception of the Soundness and Completeness Theorems 2.31 and 2.32.

One result that (perhaps surprisingly) does *not* generalize straightforwardly from λ_0 to λ_1 is the η-reduction postponement lemma (Lemma 1.24.) Consider the following reduction sequence in λ_1

$$\langle \pi_1(\lambda x(x)), \pi_2(\lambda x(x)) \rangle (x) \to_\eta \lambda x(x)(x) \to_\beta x$$

The reduction from $\langle \pi_1(\lambda x(x)), \pi_2(\lambda x(x)) \rangle (x)$ to x cannot be achieved by a sequence of β-reductions followed by a sequence of η-reductions. (Note that term $\langle \pi_1(\lambda x(x)), \pi_2(\lambda x(x)) \rangle (x)$ is actually in β-normal form.) There is therefore no simple analog of Lemma 1.24 for λ_1. However, perhaps surprisingly it turns out that although η-reduction postponement fails for the *untyped* system λ_1, it holds for the *typed* system λ_1^{type}. (Note that the expression $\langle \pi_1(\lambda x(x)), \pi_2(\lambda x(x)) \rangle$ cannot be properly typed, as $\lambda x(x)$ has the type $\tau \to \tau$ for some τ, and π_1 and π_2 can only operate on terms of type $\tau \times \sigma$.)

In particular, we have

Lemma 2.34: η-reduction postponement in λ_1^{type}.

For all λ_1^{type}-terms X, Y, Z, if $X \to_\eta Y \to_\beta Z$ in λ_1^{type}, then there is a λ_1^{type}-term Y' such that $X \to_\beta Y' \twoheadrightarrow_\eta Z$ (where $\twoheadrightarrow_\eta$ is a sequence of η-reductions.)

This is proved in the appendix. It follows from this that if a typed term can be reduced to $\beta\eta$-normal form, then there is a way of reducing it to $\beta\eta$-normal form that involves performing a sequence of β-reductions followed by a sequence of η-reductions.

Exercises for Section 2.13

1. For each of the following λ_1-terms, indicate if it can be typed or not. If it can be typed, write out the corresponding λ_1^{type} term. If it cannot, explain why not.

(i) $\pi_1(x)(y)$
(ii) $y(\pi_1(x))$
(iii) $\pi_1(\lambda x(x))$
(iv) $\langle x, y\rangle z$
(v) $z\langle x, y\rangle$
(vi) $\mathbf{in}_1(x)(y)$
(vii) $y(\mathbf{in}_1(x))$
(viii) $\mathbf{case}(\langle x, y\rangle, [u]u, [v]v)$
(ix) $\mathbf{case}(x, [u]\langle u, u\rangle, [v]\lambda z(v))$
(x) $\mathbf{case}(x, [u]w(\langle u, u\rangle), [v]v)$

2. Prove Lemma 2.23

3. Verify that Corollaries 2.25 and 2.27 do in fact follow from their respective theorems.

4. Prove Theorem 2.30.

5. Prove Theorem 2.33.

2.14 The Curry-Howard Correspondence: Version II

If we take the rules of $\mathbf{TR_1}$ and as before in each judgment $x : \tau$ erase the λ-term x and keep only the type τ, we get the following:

$$\Gamma, \tau \vdash \tau$$

$$\frac{\Gamma \vdash \tau \to \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash \sigma}$$

$$\frac{\Gamma, \tau \vdash \sigma}{\Gamma \vdash \tau \to \sigma}$$

$$\frac{\Gamma \vdash \sigma \qquad \Gamma \vdash \tau}{\Gamma \vdash \sigma \,\&\, \tau}$$

$$\frac{\Gamma \vdash \sigma \,\&\, \tau}{\Gamma \vdash \sigma} \qquad \frac{\Gamma \vdash \sigma \,\&\, \tau}{\Gamma \vdash \tau}$$

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma \vee \tau} \qquad \frac{\Gamma \vdash \tau}{\Gamma \vdash \sigma \vee \tau}$$

$$\frac{\Gamma \vdash \sigma \vee \tau \qquad \Gamma, \sigma \vdash \rho \qquad \Gamma, \tau \vdash \rho}{\Gamma \vdash \rho}$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash \tau}$$

where Γ is an (unordered) list of formulae. Here, $\times$ has been replaced by $\&$, $+$ has been replaced by $\vee$, and 0 has been replaced by $\bot$. What we have here are of course just the axioms and rules of inference for *full* intuitionistic propositional logic **NJ**.

For much of the remainder of this book, we will regard the symbols $\&$ and $\times$, the symbols $\vee$ and $+$, and the symbols 0 and $\bot$ as interchangeable. For the sake of emphasis, in the context of a typing derivation or when discussing types we will (typically) use the symbols $\times, +$ and 0, and in the context of a proof in logic we will (typically) use the symbols $\&, \vee$ and $\bot$. Thinking of these as just different ways of writing the same thing will allow us to avoid having to clutter notation with cumbersome syntactical 'translation' functions. Adopting this convention, starting with the rules of $\mathbf{TR}_1$ and erasing all λ-terms and keeping only the types, we then *literally* get the rules of **NJ**.

The relationship between $\mathbf{TR_1}$ and **NJ** is then similar to the relationship between $\mathbf{TR_0}$ and **NJ**$(\rightarrow)$. Every proof in $\mathbf{TR}_1$ contains as its 'skeleton' a proof in **NJ**, and we will see that every proof in **NJ** is the 'skeleton' of some proof in $\mathbf{TR}_0$. In fact, we have the following more general form of the Curry-Howard Correspondence:

Theorem 2.35: The Curry-Howard Correspondence for $\mathbf{TR}_1$ and NJ, Part 1.

If $\mathbf{TR_1}$ proves

$$x_1 : A_1, ..., x_n : A_n \vdash s : B$$

then **NJ** proves

$$A_1, ..., A_n \vdash B$$

Proof

In any proof in $\mathbf{TR}_1$, if one takes each assertion of the form $z : B$ and discards the term z leaving only the type B, the result is a proof of **NJ**.

Theorem 2.36: The Curry-Howard Correspondence for $\mathbf{TR}_1$ and NJ, Part 2.

If **NJ** proves

$$A_1, ..., A_n \vdash B$$

Then for any distinct variables $x_1, ..., x_n$, there is some λ_1-term s such that $\mathbf{TR_1}$ proves

$$x_1 : A_1, ..., x_n : A_n \vdash s : B$$

Proof

The proof is a straightforward induction on the construction of proofs in **NJ**, and is completely analogous to the proof of Theorem 2.8. The base case, and the case in which the final step of the proof is an application of ($\to$E) or ($\to$I) are dealt with precisely as in the proof of Theorem 2.8.

It suffices to consider the case in which the final step of the **NJ** proof is an application of one of the rules (&E), (&I), ($\vee$I), ($\vee$E) or ($\bot$E).

For example, suppose that a proof of $A_1,\ \ldots, A_n \vdash B$ has as its final step an application of (&I). This means that B has the form $C\&D$ and the proof has the form:

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ A_1,\ \ldots, A_n \vdash C & A_1,\ \ldots, A_n \vdash D\end{array}}{A_1,\ \ldots, A_n \vdash C\&D}\ (\&\text{I})$$

By inductive hypothesis we have proofs of both the following sequents:

$$x_1 : A_1,\ \ldots, x_n : A_n \vdash\ s : C \quad \text{and} \quad x_1 : A_1,\ \ldots, x_n : A_n \vdash\ t : D.$$

Combining these gives the proof

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ x_1 : A_1,\ \ldots, x_n : A_n \vdash s : C & x_1 : A_1,\ \ldots, x_n : A_n \vdash t : D\end{array}}{x_1 : A_1,\ \ldots, x_n : A_n \vdash \langle s, t\rangle : C \times D}$$

as desired.

In a similar way, proofs whose final steps are (&E), ($\vee$I), ($\vee$E), or ($\bot$E)

$$\dfrac{A_1, \ldots, A_n\ \vdash\ C\&D}{A_1, \ldots, A_n\ \vdash\ C}\ (\&\text{ E}) \qquad \dfrac{A_1, \ldots, A_n\ \vdash\ C\&D}{A_1, \ldots, A_n\ \vdash\ D}\ (\&\text{ E})$$

$$\dfrac{A_1, \ldots, A_n\ \vdash\ C}{A_1, \ldots, A_n\ \vdash\ C \vee D}\ (\vee\text{ I}) \qquad \dfrac{A_1, \ldots, A_n\ \vdash\ D}{A_1, \ldots, A_n\ \vdash\ C \vee D}\ (\vee\text{ I})$$

$$\dfrac{A_1, \ldots, A_n\ \vdash\ C \vee D \qquad A_1, \ldots, A_n, C\ \vdash\ E \qquad A_1, \ldots, A_n, D\ \vdash\ E}{A_1, \ldots, A_n\ \vdash\ E}\ (\vee\text{ E})$$

$$\dfrac{A_1, \ldots, A_n\ \vdash\ \bot}{A_1, \ldots, A_n\ \vdash\ C}\ (\bot\text{ E})$$

are transformed to $\mathbf{TR_1}$ typing derivations

$$\dfrac{\Gamma\ \vdash\ s : C \times D}{\Gamma\ \vdash\ \pi_1(s) : C}\ (\&\text{ E}) \qquad \dfrac{\Gamma\ \vdash\ s : C \times D}{\Gamma\ \vdash\ \pi_1(t) : D}\ (\&\text{ E})$$

$$\frac{\Gamma \vdash s : C}{\Gamma \vdash \mathbf{in}_1(s) : C + D}\ (\vee \text{ I}) \qquad \frac{\Gamma \vdash s : D}{\Gamma \vdash \mathbf{in}_2(s) : C + D}\ (\vee \text{ I})$$

$$\frac{\Gamma \vdash s : C + D \qquad \Gamma, y : C \vdash s' : E \qquad \Gamma, z : D \vdash s'' : E}{\Gamma \vdash \mathbf{case}(s, [y]s', [z]s'') : E}\ (\vee \text{ E})$$

$$\frac{\Gamma \vdash s : 0}{\Gamma \vdash \mathbf{abs}(s) : C}\ (\bot \text{ E})$$

where Γ is $x_1 : A_1, ..., x_n : A_n$.

These results show how we can move between proofs in intuitionistic logic **NJ** and typing derivations in $\mathbf{TR}_1$. Moving from a typing derivation in $\mathbf{TR}_1$ to a proof in **NJ** is trivial; it suffices to simply delete λ-terms and keep types. Moving from a proof in **NJ** to a typing derivation in $\mathbf{TR}_1$ is done as before, by simply assigning variables appropriately to the formulae (now viewed as types) in the leaves of the **NJ** proof, and then using the typing rules to generate the λ-terms progressively deeper in the proof.

To demonstrate this with a slightly more complex example, we have seen already that we can prove $(\neg A \vee \neg B) \to \neg(A\&B)$ in **NJ** by combining the subproofs

$$\frac{A\&B,\ \neg A \ \vdash\ \neg A \qquad \dfrac{A\&B,\ \neg A \ \vdash\ A\&B}{A\&B,\ \neg A \ \vdash\ A}\ (\&\text{ I})}{A\&B,\ \neg A \ \vdash\ \bot}\ (\to \text{ E})$$

and

$$\frac{A\&B,\ \neg B \ \vdash\ \neg B \qquad \dfrac{A\&B,\ \neg B \ \vdash\ A\&B}{A\&B,\ \neg B \ \vdash\ B}\ (\&\text{ I})}{A\&B,\ \neg B \ \vdash\ \bot}\ (\to \text{ E})$$

to give

$$\dfrac{\dfrac{\dfrac{\neg A \vee \neg B \ \vdash\ \neg A \vee \neg B \qquad \begin{matrix} \vdots \\ A\&B,\ \neg A \vdash \bot \end{matrix} \qquad \begin{matrix} \vdots \\ A\&B,\ \neg B \vdash \bot \end{matrix}}{\neg A \vee \neg B,\ A\&B \vdash \bot}\ (\vee \text{ E}')}{\neg A \vee \neg B \vdash \neg(A\&B)}\ (\to \text{ I})}{\vdash (\neg A \vee \neg B) \to \neg(A\&B)}\ (\to \text{ I})$$

The leaves of this larger proof consist of the formulae

$$\neg A \vee \neg B,\ A\&B,\ \neg A,\ \neg B$$

Let w, x, y, z be variables of these types, i.e.,

$$w : \neg A + \neg B,\ x : A \times B,\ y : \neg A,\ z : \neg B$$

The first subproof is transformed into the typing derivation

$$\dfrac{x:\ A\times B,\ y:\neg A\ \vdash\ y:\neg A \qquad \dfrac{x:\ A\times B,\ y:\neg A\ \vdash\ x:A\times B}{x:\ A\times B,\ y:\neg A\ \vdash\ \pi_1(x):A}}{x:\ A\times B,\ y:\neg A\ \vdash\ y(\pi_1(x)):\bot}$$

The second subproof is likewise transformed into the typing derivation

$$\dfrac{x:\ A\times B,\ z:\neg B\ \vdash\ z:\neg B \qquad \dfrac{x:\ A\times B,\ z:\neg B\ \vdash\ x:A\times B}{x:\ A\times B,\ z:\neg B\ \vdash\ \pi_2(x):B}}{x:\ A\times B,\ z:\neg B\ \vdash\ z(\pi_2(x)):\bot}$$

putting these together yields:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{w:\neg A+\neg B\ \vdash\ w:\neg A+\neg B} \qquad \dfrac{\vdots}{x:A\times B,y:\neg A\vdash y(\pi_1(x)):\bot} \qquad \dfrac{\vdots}{x:A\times B,z:\neg B\vdash z(\pi_2(x)):\bot}}{w:\neg A+\neg B,\ x:\ A\times B\vdash \mathbf{case}(w,[y]y(\pi_1(x)),[z]z(\pi_2(x))):\bot}}{w:\neg A+\neg B\vdash \lambda x\ \mathbf{case}(w,[y]y(\pi_1(x)),[z]z(\pi_2(x))):\neg(A\times B)}}{\vdash \lambda w\lambda x\ \mathbf{case}(w,[y]y(\pi_1(x)),[z]z(\pi_2(x))):(\neg A+\neg B)\to\neg(A\times B)}$$

where for the sake of compactness typing declarations have been omitted from contexts when not needed. This of course is a typing derivation in $\mathbf{TR}_1$.

As before, we can use this version of the Curry-Howard correspondence to solve technical problems about $\mathbf{TR}_1$ by using intuitionistic logic. For example, for arbitrary types A and B the type $(A+B)\to A$ is *not* inhabited, as $(A\vee B)\to A$ is not a theorem of intutionistic (or classical) logic. In this way, understanding the structure of **NJ** gives us a tool for understanding the structure of $\mathbf{TR}_1$.

Conversely, understanding the structure of $\mathbf{TR}_1$ can help us to understand the structure of **NJ**. We begin with a more superficial and familiar type of example. Is $A\to(B\to(A\&B))$ intuitionistically provable? This question can be answered easily enough by examining the rules of **NJ**. But let us consider this question through the lens of the Curry-Howard correspondence. By the Curry-Howard correspondence, our question is equivalent to whether we can find a term in our expanded λ-calculus that can be assigned the type $A\to(B\to(A\times B))$. In fact, it is not hard to see that $\lambda x\lambda y(\langle x,y\rangle)$ is such a term. If x has type A and y has type B, then this term has type $A\to(B\to(A\times B))$, as the following typing derivation shows:

$$\dfrac{\dfrac{\dfrac{\dfrac{x:A,y:B\ \vdash\ x:A \qquad x:A,y:B\ \vdash\ y:B}{x:A,y:B\ \vdash\ \langle x,y\rangle:A\times B}}{x:A\ \vdash\ \lambda y(\langle x,y\rangle):B\to(A\times B)}}{\vdash\ \lambda x\lambda y(\langle x,y\rangle):A\to(B\to(A\times B))}$$

Deleting λ-terms then yields the following proof of $A\to(B\to(A\&B))$ in **NJ**

$$\dfrac{\dfrac{\dfrac{A,B\ \vdash\ A \qquad A,B\ \vdash\ B}{A,B\ \vdash\ A\&B}}{A\ \vdash\ B\to(A\&B)}}{\vdash\ A\to(B\to(A\&B))}$$

We will see more substantive applications of $\mathbf{TR}_1$ to our understanding of **NJ** in the next sections when we return to the topic of proof normalization.

Exercises for Section 2.14

1. The following sentences X are all theorems of **NJ**. For each of them, first construct a proof in **NJ**. (You have already constructed proofs for some of these in earlier exercises.) Then transform each of these proofs of X into a typing derivation in $\mathbf{TR}_1$ of $\vdash M : X$ for some λ_1-term M.

(a) $p \to (p \vee q)$
(b) $\neg(p \vee q) \to \neg p$
(c) $(p \& \neg p) \to q$
(d) $((p \to r) \& (q \to r)) \to ((p \vee q) \to r)$
(e) $\neg\neg(p \vee \neg p)$

2. Are all the λ_1-terms M obtained in problem **1.(a)-(e)** in β-normal form? If not, in each case find a different **NJ** proof for which the relevant λ_1-term M is in β-normal form.

2.15 Proof Normalization Revisited

Earlier, we examined **NJ**($\to$) proofs in which an application of the ($\to$I) rule was immediately followed by an application of the ($\to$E) rule, with the conclusion of the ($\to$I) rule used as the major premise:

$$\dfrac{\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\,(\to\text{I}) \qquad \Gamma \vdash A}{\Gamma \vdash B}\,(\to\text{E})$$

The typing derivations corresponding to such proofs look like:

$$\dfrac{\dfrac{\overline{\Gamma}, u : A \vdash e : B}{\overline{\Gamma} \vdash \lambda u(e) : A \to B} \qquad \overline{\Gamma} \vdash f : A}{\overline{\Gamma} \vdash \lambda u(e)(f) : B}$$

The term $\lambda u(f)(g)$ is of course β-reducible. We concluded that typing derivations of β-normal form terms corresponded to **NJ**($\to$) proofs without the pattern of reasoning in question. This was formally stated as Theorem 2.13.

This basic idea generalizes to our other connectives. Consider a proof in which an application of the (&I) rule is immediately followed by an application of the (&E) rule:

$$\dfrac{\dfrac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \& B}\,(\&\text{I})}{\Gamma \vdash A}\,(\&\text{E})$$

A typing derivation corresponding to this proof will look like:

$$\dfrac{\dfrac{\overline{\Gamma} \vdash e : A \qquad \overline{\Gamma} \vdash f : B}{\overline{\Gamma} \vdash \langle e, f\rangle : A \times B}}{\overline{\Gamma} \vdash \pi_1(\langle e, f\rangle) : A}$$

Note that in this case too the final term $\pi_1(\langle e, f\rangle)$ is not in β-normal form.

Finally, consider a proof in which an application of the ($\vee$I) rule is immediately followed by an application of the ($\vee$E) rule, with the conclusion of the ($\vee$I) rule used as the major premise. Here there are two possibilities:

$$\dfrac{\dfrac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\,(\vee\text{I}) \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\,(\vee\text{E})$$

and

$$\dfrac{\dfrac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\,(\vee\text{I}) \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\,(\vee\text{E})$$

The typing derivations corresponding to these will look like:

$$\dfrac{\dfrac{\overline{\Gamma} \vdash e : A}{\overline{\Gamma} \vdash \mathbf{in}_1(E) : A + B} \qquad \overline{\Gamma}, u : A \vdash f : C \qquad \overline{\Gamma}, v : B \vdash g : C}{\Gamma \vdash \mathbf{case}(\mathbf{in}_1(e), [u]f, [v]g) : C}$$

and

$$\dfrac{\dfrac{\overline{\Gamma} \vdash e : B}{\overline{\Gamma} \vdash \mathbf{in}_2(E) : A + B} \qquad \overline{\Gamma}, u : A \vdash f : C \qquad \overline{\Gamma}, v : B \vdash g : C}{\Gamma \vdash \mathbf{case}(\mathbf{in}_2(e), [u]f, [v]g) : C}$$

Again, the final term $\mathbf{case}(\mathbf{in}_1(e), [u]f, [v]g)$ or $\mathbf{case}(\mathbf{in}_2(e), [u]f, [v]g)$ is not in β-normal form.

With this in mind, we consider the following definition

Definition 2.37: Normal Proof of NJ.

A proof in **NJ** is *normal* just in case

(i) it contains no occurrence of a ($\to$I) rule introducing a conditional $X \to Y$ which is immediately followed by an application of the ($\to$E) rule with this conditional $X \to Y$ as its major premise.

(ii) it contains no occurrence of a ($\&$I) rule immediately followed by an application of the ($\&$E) rule.

(iii) it contains no occurrence of a ($\vee$I) rule introducing a disjunction $X \vee Y$ which is immediately followed by an application of the ($\vee$E) rule with this disjunction $X \vee Y$ as its

major premise.

As before we have

Lemma 2.38: The Subterm Lemma for $\mathbf{TR}_1$

In any typing derivation of $\mathbf{TR}_1$ with conclusion of the form $\Gamma \vdash s : A$, if s' is a subterm of s then the typing derivation contains a sequent of the form $\Gamma' \vdash s' : B$, where $\Gamma' \subseteq \Gamma$.

and

Theorem 2.39

Given a typing derivation T in $\mathbf{TR}_1$ whose conclusion is a sequent of the form $\Delta \vdash g : C$, let P be the **NJ** proof obtained from T by deleting all terms and keeping types. Then P is a normal proof iff g is in β-normal form.

As before, we may conclude

Theorem 2.40: Proof Normalization Theorem for NJ

If $A_1, ..., A_n \vdash B$ has a proof in **NJ**, then it has a normal proof in **NJ**.

Proof

Suppose P_1 is a non-normal proof of $A_1, ..., A_n \vdash B$. By the Curry-Howard Correspondence (in particular, Theorem 2.36), this may be transformed into a typing derivation with conclusion

$$x_1 : A_1, ..., x_n : A_n \vdash s : B$$

for some λ-term s. By Theorem 2.39, s is not in β-normal form. Let s' be the β-normal form of s. By Theorem 2.33 (the Subject Reduction Theorem) there is a typing derivation of

$$x_1 : A_1, ..., x_n : A_n \vdash s' : B$$

in $\mathbf{TR}_1$. Because s' is in β-normal form, it follows from Theorem 2.39 that the **NJ** proof obtained by taking this typing derivation and removing all terms and keeping types is a normal proof of $A_1, ..., A_n \vdash B$.

As before, this gives us an algorithm for proof normalization. Consider for example the following non-normal proof of $A \to (B \to A)$ in **NJ**

$$\dfrac{\dfrac{\dfrac{\dfrac{A,\ B\ \vdash\ A}{A\ \vdash\ B \to A}\,(\to \text{I}) \qquad A\ \vdash\ A}{A\ \vdash\ (B \to A)\&A}\,(\&\ \text{I})}{A\ \vdash\ B \to A}\,(\&\ \text{E})}{\vdash\ A \to (B \to A)}\,(\to \text{I})$$

This is non-normal because it involves the (& I) rule immediately followed by the (& E) rule. This has the corresponding typing derivation

$$\dfrac{\dfrac{\dfrac{\dfrac{x : A,\ y : B\ \vdash\ x : A}{x : A\ \vdash\ \lambda y(x) : B \to A} \qquad x : A\ \vdash\ x : A}{x : A\ \vdash\ \langle \lambda y(x), x\rangle : (B \to A) \times A}}{x : A\ \vdash\ \pi_1\langle \lambda y(x), x\rangle : B \to A}}{\vdash\ \lambda x(\pi_1\langle \lambda y(x), x\rangle) : A \to (B \to A)}$$

The term $\lambda x(\pi_1\langle \lambda y(x), x\rangle)$ in the conclusion is not in β-normal form. It β-reduces to $\lambda x \lambda y(x)$, which is in β-normal form. The term $\lambda x \lambda y(x)$ may be given the type $A \to (B \to A)$ as follows

$$\dfrac{\dfrac{x : A,\ y : B\ \vdash\ x : A}{x : A\ \vdash\ \lambda y(x) : B \to A}}{\vdash\ \lambda x \lambda y(x) : A \to (B \to A)}$$

which corresponds to the **NJ** proof

$$\dfrac{\dfrac{A,\ B\ \vdash\ A}{A\ \vdash\ B \to A}\,(\to \text{I})}{\vdash\ A \to (B \to A)}\,(\to \text{I})$$

This proof is indeed normal.

Let us turn to an examination of the structure of normal proofs. Recall the definition of a *major branch*, which requires no changes

Definition 2.41: Major Branch

A *major branch* of a proof is a branch of the proof with the property that if the conclusion of an inference lies on the branch and that inference has a major premise, then that major premise also lies on in the branch.

As before, we easily see that each proof has at least one major branch. We also have the following:

Lemma 2.42: The Major Branch Lemma.

In any normal proof in **NJ**, a major brach will consist of a (possibly empty) sequence of elimination rules (i.e., ($\rightarrow$ E), ($\&$ E), ($\vee$ E) and ($\bot$ E) rules) followed by a (possibly empty) sequence of introduction rules (i.e., ($\rightarrow$ I), ($\&$ I) and ($\vee$ I) rules.)

Proof

We first argue that along a major branch in a normal proof, a ($\rightarrow$ I) rule cannot be immediately followed by *any* elimination rule. To see why, note that because the conclusion of a ($\rightarrow$ I) rule has the form $\Delta \vdash \phi \rightarrow \psi$, it cannot be immediately followed by a ($\&$ E) or a ($\bot$ E) rule. Because the only premise of a ($\vee$ E) rule that appears on a major branch is the major premise, which always has the form $\Gamma \vdash \phi \vee \phi$, it cannot be followed by a ($\vee$ E) rule either. If it is followed by an elimination rule, it must therefore be followed by a ($\rightarrow$ E) rule. But this happening would contradict the fact that the proof is normal. So a ($\rightarrow$ I) rule cannot be followed by *any* elimination rule.

In a precisely similar way, it may be argued that along a major branch, a ($\&$ I) rule cannot be followed by any elimination rule, and that a ($\vee$ I) rule cannot be followed by any elimination rule. It thus follows that along a major branch of a normal proof, an introduction rule can only be followed by another introduction rule. It follows that any major brach of a normal must consist of a (possibly empty) sequence of elimination rules followed by a (possibly empty) sequence of introduction rules.

As before, the Major Branch Lemma gives a wealth of information about what is provable in **NJ**. For example:

Theorem 2.43: Consistency of NJ

No atomic sentence X is provable in **NJ**.

Recall that the only atomic sentences in our language are the propositional variables $X, Y, Z, ...$ and the sentence $\bot$. The proof is essentially identical to the consistency proof given for **NJ**($\rightarrow$) (i.e., Theorem 2.17).

Proof

Suppose **NJ** proves S where S is either a propositional variable or the symbol $\bot$. We then know from Theorem 2.40 that there is a normal proof P whose conclusion is $\vdash S$. The last step of P cannot be an introduction rule, as each introduction rule introduces a non-atomic

sentence into the conclusion. The last step of the proof is therefore an elimination rule. Using the Major Branch Lemma (Lemma 2.42), it then follows that any major branch of P must consist solely of elimination rules. But along a major branch, the elimination rules all leave the left hand side of sequents unchanged. Because the left hand side of every leaf is non-empty, it follows that the conclusion of P must have a non-empty left hand side, which is a contradiction.

The following result is also of interest

Theorem 2.44

The only atomic sentence X for which $\neg X$ provable in **NJ** is $\bot$.

Proof

Suppose **NJ** proves $\neg X$ where X is an atomic sentence distinct from $\bot$. Consider a normal proof of $\vdash \neg X$. A major branch of this proof could not consist entirely of elimination rules, because the left hand side of $\vdash \neg X$ is empty. Therefore the proof must end in an introduction rule. Because $\neg X$ is a conditional, the only possible introduction rule the proof could end in is ($\rightarrow$ I). The proof therefore has the form:

$$\dfrac{\begin{array}{c}\vdots\\ X \vdash \bot\end{array}}{\vdash \neg X}\ (\rightarrow \text{I})$$

The sequent $X \vdash \bot$ cannot be the conclusion of an introduction rule, and so must be the conclusion of an elimination rule. Thus, any major branch must consist entirely of elimination rules, followed by the ($\rightarrow$ I) inference at the end of the proof. Because elimination rules do not change the left hand side of a sequent along a major branch, the proof must have the form

$$\dfrac{\begin{array}{c}X \vdash X\\ \vdots\\ X \vdash \bot\end{array}}{\vdash \neg X}\ (\rightarrow \text{I})$$

For atomic sentences X, the only case in which an elimination rule can be performed on the leaf $X \vdash X$ on a major branch is the case in which X is $\bot$ (in which case a ($\bot$ E) can be performed). Because we are assuming that X is distinct from $\bot$, it follows that no elimination rule can be performed on the leaf $X \vdash X$, and thus it is impossible to derive $X \vdash \bot$ solely using elimination rules. On the other hand, if X is $\bot$, then

$$\dfrac{\bot \vdash \bot}{\vdash \neg \bot}\ (\rightarrow \text{I})$$

is a derivation of $\neg X$ in **NJ**.

From this, we easily have

Theorem 2.45: Underivability of Double Negation Elimination

If X is atomic sentence distinct from $\bot$, then $\neg\neg X \to X$ is not provable in **NJ**.

Proof

Suppose to the contrary that $\neg\neg X \to X$ is provable in **NJ** for some atomic sentence X distinct from $\bot$, and consider a normal proof of $\vdash \neg\neg X \to X$. The last step in this proof cannot be an elimination rule, as this would mean that any major branch of this proof would consist entirely of elimination rules, and thus the concluding sequent would have non-empty left hand side. So the final rule must be an introduction rule. Given that $\neg\neg X \to X$ is neither a conjunction nor a disjunction, the only possibility is that the last rule is $(\to \text{I})$:

$$\frac{\neg\neg X \vdash X}{\vdash \neg\neg X \to X}\,(\to \text{I})$$

Because the right hand side of the sequent $\neg\neg X \vdash X$ is atomic, it cannot be the result of an introduction rule, and must be the result of an elimination rule. Any major branch of the proof must therefore consist entirely of elimination rules followed by a single $(\to \text{I})$. Because along a major branch the left hand side of a sequent remains unchanged when performing elimination rules, any major branch must therefore have the form

$$\dfrac{\begin{array}{c}\neg\neg X \vdash \neg\neg X \\ \vdots \\ \neg\neg X \vdash X\end{array}}{\vdash \neg\neg X \to X}\,(\to \text{I})$$

The only elimination rule that can be applied to the leaf $\neg\neg X \vdash \neg\neg X$ along a major branch is $(\to \text{E})$, which means that $\neg\neg X$ (i.e., $(X \to \bot) \to \bot$) must be the major premise, and the proof has the form

$$\dfrac{\begin{array}{c}\dfrac{\neg\neg X \vdash \neg\neg X \qquad \begin{array}{c}\vdots \\ \neg\neg X \vdash \neg X\end{array}}{\neg\neg X \vdash \bot}\,(\to \text{E}) \\ \vdots \\ \neg\neg X \vdash X\end{array}}{\vdash \neg\neg X \to X}\,(\to \text{I})$$

Thus we have that $\neg\neg X \vdash \bot$ is derivable in **NJ**. Using ($\to$ I) we then have that $\vdash \neg\neg\neg X$ is derivable in **NJ**, and using the fact that $\vdash \neg\neg\neg X \to \neg X$ is derivable in **NJ** and ($\to$ E), we finally have that $\vdash \neg X$ is derivable in **NJ**. This however contradicts Theorem 2.44.

Note that the sequent $\neg\neg\ \bot\to\bot$ is indeed provable in **NJ**. The proof is

$$\dfrac{\dfrac{\neg\neg\ \bot \vdash \neg\neg\ \bot \qquad \dfrac{\neg\neg\ \bot, \bot \vdash\ \bot}{\neg\neg\ \bot \vdash \neg\ \bot}\,(\to \text{I})}{\neg\neg\ \bot \vdash\ \bot}\,(\to \text{E})}{\vdash\ \neg\neg\ \bot\to\bot}\,(\to \text{I})$$

Some further applications of the Major Branch Lemma will be given in the exercises. One of the most interesting things about our proofs of Theorems 2.43, 2.44 and 2.45 is that they show the unprovability of certain sequents on purely syntactic grounds. No semantic considerations of any sort are invoked. That is to say, we do not construct 'models' in which certain sentences are false, thereby somehow showing that these sentences cannot be derived. Instead, our proofs merely consider the syntactic form of the sentences in question to argue for their unprovability.

Exercises for Section 2.15

1. Each of the following proofs in **NJ** is *not* normal. Use the techniques of this section to transform them into normal proofs.

(a)

$$\dfrac{\dfrac{B,\ C,\ C \to A\ \vdash\ C \qquad \dfrac{A, B,\ C,\ C \to A\ \vdash\ C}{B,\ C,\ C \to A\ \vdash\ C \to A}\,(\to \text{I})}{B,\ C,\ C \to A\ \vdash\ A}\,(\to \text{E}) \qquad B,\ C,\ C \to A\ \vdash\ B}{\dfrac{B,\ C,\ C \to A\ \vdash\ A\&B}{B,\ C,\ C \to A\ \vdash\ A}\,(\&\text{E})}\,(\&\text{I})$$

(b)

$$\dfrac{\dfrac{A\&B\ \vdash\ A\&B}{A\&B\ \vdash\ A}\,(\&\text{E})}{A\&B\ \vdash\ A \vee B}\,(\vee\text{I}) \qquad \dfrac{A\&B, A\ \vdash\ A\&B}{A\&B, A\ \vdash\ B}\,(\&\text{E}) \qquad \dfrac{\dfrac{A\&B, B\ \vdash\ B}{A\&B, B\ \vdash\ A\&B}\,(\&\text{I})}{A\&B, B\ \vdash\ B}\,(\&\text{E})}{A\&B\ \vdash\ B}\,(\vee\text{E})$$

2. Using the Major Branch Lemma, show that the following sequents are not derivable in **NJ**, where X and Y are *distinct* atomic sentences, with neither equal to $\bot$.

(i) $X \vdash Y$

(ii) $\vdash X \vee \neg X$

(iii) $\neg(X\&Y) \vdash X$, and likewise $\neg(X\&Y) \vdash Y$. (Hint: consider the normal proof with the fewest number of sequents.)

(iv) $\neg(X\&Y), X \vdash Y$

(v) $\neg(X\&Y), X \vdash \bot$, and likewise $\neg(X\&Y), Y \vdash \bot$
(vi) $\vdash \neg(X\&Y) \to (\neg X \vee \neg Y)$

3. By using the Major Branch Lemma, show that Pierce's Rule is not derivable in **NJ**. (You should be able to adapt the proof that it is not provable in **NJ**$(\to)$.)

4. Show that if X is a sentence constructed from atomic sentences (including $\bot$) using only the connective $\to$, then if **NJ** proves $\vdash X$, then **NJ**$(\to)$ proves $\vdash X$.

2.16 The BHK Interpretation Revisited

We finally return to topic of the **BHK** conditions for intutionistic logic. Armed with everything we have presented so far, we will see that there is a sense in which these laws can be vindicated in a concrete and natural way.

Recall the four **BHK** conditions:

BHK1 : To prove $A\&B$ is to prove A and to prove B.
BHK2 : To prove $A \vee B$ is to prove A or to prove B.
BHK3 : To prove $A \to B$ is to construct a way of transforming a proof of A into a proof of B.
BHK4 : There is no proof of $\bot$.

There are two main obstacles in making rigorous sense of these claims. To see the first, let us consider **BHK1**. It is *not* true that in intuitionistic logic, any proof of $A\&B$ literally consists of a proof of A and a proof of B, conjoined into a proof of $A\&B$ by the (&I) rule. This is because we can introduce unnecessary detours into a proof. For example, a proof of A and a proof of B can lead to a proof of $A\&B$ as follows:

$$\dfrac{\dfrac{\dfrac{\vdash A \qquad \vdash B}{\vdash A\&B}\,(\&\,\mathrm{I})}{\vdash (A\&B)\&A}\,(\&\,\mathrm{E})}{\vdash A\&B}\,(\&\,\mathrm{I})$$

The problem with this proof of course is that it is not normal. A natural idea then is to think of the **BHK** conditions as claims about *normal* proofs.

A second obstacle in making sense of the **BHK** conditions revolves around **BHK3**. This claim tells us that proofs of conditionals are *functions* of a certain sort. But **NJ** proofs are not functions in any literal sense – they are trees of sequents. We turn to the question of how to make sense of **BHK3** shortly.

We begin with the following theorem

Theorem 2.46: BHK Theorem

The following are all true

(i) If a normal proof in **NJ** has a conclusion of the form $\vdash X\&Y$, then it must have the form

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ \vdash X & \vdash Y\end{array}}{\vdash X\&Y}\ (\&\ \mathrm{I})$$

where the proofs of the sequents $\vdash X$ and $\vdash Y$ are also normal.

(ii) If a normal proof in **NJ** has a conclusion of the form $\vdash X \vee Y$, then it must have one of the forms:

$$\dfrac{\begin{array}{c}\vdots \\ \vdash X\end{array}}{\vdash X \vee Y}\ (\vee\ \mathrm{I}) \qquad \text{or} \qquad \dfrac{\begin{array}{c}\vdots \\ \vdash Y\end{array}}{\vdash X \vee Y}\ (\vee\ \mathrm{I})$$

where the proof of the sequent $\vdash X$ or $\vdash Y$ is also normal.

(iii) If a normal proof in **NJ** has a conclusion of the form $\vdash X \to Y$, then it must have the form

$$\dfrac{\begin{array}{c}\vdots \\ X \vdash Y\end{array}}{\vdash X \to Y}\ (\to\ \mathrm{I})$$

where the proof of the sequent $X \vdash Y$ is also normal.

(iv) No normal proof in **NJ** has the form

$$\begin{array}{c}\vdots \\ \vdash \bot\end{array}$$

Proof

For (i), suppose a normal proof in **NJ** has a conclusion of the form $\vdash X\&Y$, and consider a major branch of this proof. This major branch cannot consist entirely of elimination rules, as the conclusion $\vdash X\&Y$ has an empty left hand side. Therefore the final rule of inference in this branch must be an introduction rule. Because $X\&Y$ has the form of a conjunction, the only introduction rule which could prove the sequent $\vdash X\&Y$ is the (& I) rule. The proof must therefore have the form

$$\dfrac{\begin{array}{cc}\vdots & \vdots \\ \vdash X & \vdash Y\end{array}}{\vdash X\&Y}\ (\&\ \mathrm{I})$$

Because subproofs of normal proofs are also normal proofs, the proofs of the sequents $\vdash X$ and $\vdash Y$ are also normal.

The proofs of (ii) and (iii) involve essentially the same form of argument. That (iv) is true follows from Theorem 2.43.

Theorem 2.46 shows that we may view **BHK1**, **BHK2** and **BHK4** as true claims about *normal* proofs.

Let us turn to condition **BHK3**. Is there a sense in which a normal proof of $A \to B$ is a function that takes us from a normal proof of A to a normal proof of B? While a normal proof of $A \to B$ is not *literally* a function from normal proofs of A to normal proofs of B, it is easily seen to encode such a function. For given any proofs of $A \to B$ and A in **NJ**, we can produce a proof of B in **NJ** by invoking ($\to$ E):

$$\dfrac{\vdash\ A \to B \qquad \vdash\ A}{\vdash\ B}\ (\to \mathrm{E})$$

This proof need not be normal, but normalizing it in the way described in the previous sections yields a normal proof. Thus, given any proof of $A \to B$, we have an algorithm that takes a normal proof of A to a normal proof of B. In this sense, **BHK3** is also vindicated.

Thus we see that all of **BHK1-4** may be interpreted as true claims about normal proofs of **NJ**. In this sense, the proof system **NJ** does indeed capture the ideas behind the **BHK** interpretation, and we are therefore right to think of **NJ** as a formalization of 'constructive' or 'intuitionistic' reasoning. One advantage (among many) of the formalism of type theory and the Curry-Howard correspondence is the way in which it allows us to see all of this particularly clearly.

Exercises for Section 2.16

1. Show the following.

(i) It is *not* true in general that if **NJ** proves a sequent of the form $\Delta \vdash X \vee Y$, then it must prove either $\Delta \vdash X$ or $\Delta \vdash Y$.

(ii) If each sentence in Δ has the form $\neg Z$, then if **NJ** proves $\Delta \vdash X \vee Y$, it must also prove either $\Delta \vdash X$ or $\Delta \vdash Y$.

2.17 Other Resources

For a good discussion of propositional intuitionistic logic and the Curry-Howard Correspondence, see chapters 2 and 4 of Sorenson and Urzyczyn's *Lectures on the Curry-Howard Isomorphism* [10] and chapter 6 of Selinger's *Lecture Notes on the Lambda Calculus* [9]. Another good reference focusing

mainly on the connection with intuitionistic logic is Mints' *A Short Introduction to Intuitionistic Logic* [7]. For a discussion of the themes of this chapter more from the point of view of computer science, chapters 8 through 14 of Pierce's *Types and Programming Languages* [8] is a good introduction.

2.18 Appendix

In this section, we provide proofs for several results in the main text.

Lemma 2.34: η-reduction postponement in λ_1.

For all λ_1^{type}-terms X, Y, Z, if $X \to_\eta Y \to_\beta Z$ in λ_1^{type}, then there is a λ_1^{type}-term Y' such that $X \twoheadrightarrow_\beta Y' \twoheadrightarrow_\eta Z$ (where $\twoheadrightarrow_\eta$ is a sequence of η-reductions.)

Proof of Lemma 2.34.

As very little of the argument explicitly uses the fact that we are working in the typed λ-calculus λ_1^{type}, we will omit typing superscripts in what follows for ease of readability, instead explicitly drawing attention to facts about typing when needed.

Suppose that we have λ_1^{type}-terms X, Y, Z with $X \to_\eta Y \to_\beta Z$. Then X can be written in the form $Et'F$ and Y in the form EtF where E and F are sequences of symbols, and t' is an η-redex – i.e., a term with one of the forms $\lambda x(t(x))$ (with x not free in t), $\langle \pi_1(t), \pi_2(t) \rangle$, or $\mathbf{case}(t, [u]\mathbf{in}_1(u), [v]\mathbf{in}_2(v))$. Thus we write

$$Et'F \to_\eta EtF \to_\beta Z$$

where Z is a λ-term.

The β-reduction $EtF \to_\beta Z$ occurs by reducing some β-redex s which occurs a subexpression of EtF. We then have either that (i) s is a subexpression of t (where this includes the possibility that $s = t$), (ii) t is a proper subexpression of s, or (iii) s and t are disjoint.

Consider first case (i). Assume that when the β-reduction in question is performed on the subexpression s of t, we get $t \to_\beta t''$. We can then represent our reduction sequence as

$$Et'F \to_\eta EtF \to_\beta Et''F$$

Because $t \to_\beta t''$, we may then reverse the order of the β- and η-reductions for each of our 3 possible types of η-reduction as follows

$$Et'F = E \begin{array}{c} \lambda x(t(x)) \\ \langle \pi_1(t), \pi_2(t) \rangle \\ \mathbf{case}(t, [u]\mathbf{in}_1(u), [v]\mathbf{in}_2(v)) \end{array} F \twoheadrightarrow_\beta E \begin{array}{c} \lambda x(t''(x)) \\ \langle \pi_1(t''), \pi_2(t'') \rangle \\ \mathbf{case}(t'', [u]\mathbf{in}_1(u), [v]\mathbf{in}_2(v)) \end{array} F \twoheadrightarrow_\eta Et''F$$

In the case in which t' is $\lambda x(t(x))$, we must use the fact that x is not free in t''. This is true because x is not free in t and no β-reduction introduces a new free variable.

Consider next case (iii). If s and t are disjoint, then s is either a subexpression of E or a subexpression of F. Suppose without loss of generality that s is a subexpression of E, so that E can be written as GsH. Then our reduction sequence has the form

$$GsHt'F \to_\eta GsHtF \to_\beta Gs'HtF$$

where $s \to_\beta s'$ and $t' \to_\eta t$. In this case, the order of the β- and η-reductions may easily be reversed as follows

$$GsHt'F \to_\beta Gs'Ht'F \to_\eta Gs'HtF$$

Case (ii) (t is a proper subexpression of s) requires the most care. In this case we have

$$Et'F \to_\eta EtF \to_\beta Z$$

where the β-redex s being reduced in $EtF \to_\beta Z$ has t as a proper subexpression. We consider each of the possible form of β-redex in turn.

First, suppose s has the form $\lambda z(M)(N)$. Because t is a proper subexpression of s, we have that (a) t is $\lambda z(M)$, (b) t is a subexpression of M, or (c) t is a subexpression of N.

In case (a) we use some facts about typing. (This is one of the three points in the proof that relies on the fact that we are working in the *typed* setting.) If t is $\lambda z(M)$, then the type of t has the form $\alpha \to \beta$. The only η-reduction that returns an object of such a type is the reduction rule $\lambda x(E(x)) \to_\eta E$. Thus, t' has the form $\lambda x(\lambda z(M)(x))$ and the reduction sequence has the form

$$E\lambda x(\lambda z(M)(x))F \to_\eta E\lambda z(M)F \to_\beta Z$$

where F has the form $(N)F'$, i.e.,

$$E\lambda x(\lambda z(M)(x))(N)F' \to_\eta E\lambda z(M)(N)F' \to_\beta EM[N/z]F'$$

where x does not occur free in $\lambda z(M)$. Assuming as we may that x and z are distinct variables, this means that x does not occur free in M. But then we also have the reduction sequence

$$E\lambda x(\lambda z(M)(x))(N)F' \to_\beta E\lambda x(M[x/z])(N)F' \to_\beta EM[x/z][N/x]F' = EM[N/z]F',$$

where $EM[x/z][N/x]F' = EM[N/z]F'$ follows from the fact that x does not occur free in M. This is a sequence of β-reductions followed by an (empty) sequence of η-reductions, as desired.

In case (b), because M contains t as a subexpression, we can write M as $G[t/u]$, where u is a completely new variable. Our reduction sequence then has the form

$$E\lambda z(G[t'/u])(N)F' \to_\eta E\lambda z(G[t/u])(N)F' \to_\beta EG[t/u][N/z]F'$$

which can be rewritten

$$E\lambda z(G[t'/u])(N)F' \to_\beta EG[t'/u][N/z]F' \to_\beta EG[t/u][N/z]F'$$

where we have used Lemma 2.23 to get $G[t'/u][N/z] \to_\beta G[t/u][N/z]$, using the fact that $G[t'/u] \to_\beta G[t/u]$.

In case (c), using similar considerations our reduction sequence can be written in the form

$$E\lambda z(M)(G[t'/u])F' \to_\eta E\lambda z(M)(G[t/u])F' \to_\beta EM[G[t/u]/z]F'$$

which can be rewritten

$$E\lambda z(M)(G[t'/u])F' \to_\beta EM[G[t'/u]/z]F' \to_\eta EM[G[t/u]/z]F'$$

This completes the argument for case (iii) in which s has the form $\lambda z(M)(N)$.

Next, suppose in case (ii) that s has the form $\pi_1\langle M, N\rangle$. Because t is a proper subexpression of s, we have that (a) t is $\langle M, N\rangle$, (b) t is a subexpression of M, or (c) t is a subexpression of N.

In case (a) we again invoke the fact that we are working in the typed setting. If t is $\langle M, N\rangle$, then the type of t has the form $\alpha \times \beta$. The only η-reduction that returns an object of such a type is the reduction rule $\langle \pi_1(S), \pi_2(S)\rangle \to_\eta S$. Thus, t' has the form $\langle \pi_1\langle M, N\rangle, \pi_2\langle M, N\rangle\rangle$ and the reduction sequence has the form

$$E\ \pi_1\langle \pi_1\langle M, N\rangle, \pi_2\langle M, N\rangle\rangle F \to_\eta E\pi_1\langle M, N\rangle F \to_\beta EMF$$

But then we also have the reduction sequence

$$E\ \pi_1\langle \pi_1\langle M, N\rangle, \pi_2\langle M, N\rangle\rangle F \twoheadrightarrow_\beta E\pi_1\langle M, N\rangle F \to_\beta EMF$$

This is a sequence of β-reductions followed by an (empty) sequence of η-reductions, as desired.

In case (b), our reduction sequence can be written in the form

$$E'\pi_1\langle G[t/u], N\rangle F' \to_\eta E'\pi_1\langle G[t/u], N\rangle F' \to_\beta E'G[t/u]F'$$

which may be rewritten

$$E'\pi_1\langle G[t/u], N\rangle F' \to_\beta E'G[t/u]F' \to_\eta E'G[t/u]F'.$$

Case (c) is completely symmetric. The case in which s has the form $\pi_2\langle M, N\rangle$ is also completely symmetric.

Next, suppose in case (ii) that s has the form $\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]Q)$. Because t is a proper subexpression of s, we have that (a) t is the subexpression $\mathbf{in}_1(R)$, (b) t is a subexpression of R, (c) t is a subexpression of P, or (d) t is a subexpression of Q.

In case (a), we again use some facts about typing. If t is $\mathbf{in}_1(R)$, then the type of t must have the form $\alpha + \beta$. The only η-reduction that returns an object of such a type is the reduction rule $\mathbf{case}(t, [x]\mathbf{in}_1(x), [y]\mathbf{in}_2(y)) \to_\eta t$, and thus we know that t' has the form $\mathbf{case}(\mathbf{in}_1(R), [x]\mathbf{in}_1(x), [y]\mathbf{in}_2(y))$, which both η and β reduces to $\mathbf{in}_1(R)$. The reduction sequence thus has the form

$$E'\mathbf{case}(\mathbf{case}(\mathbf{in}_1(R), [x]\mathbf{in}_1(x), [y]\mathbf{in}_2(y)), [u]P, [v]Q)F' \\ \to_\eta E'\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]Q)F' \to_\beta E'P[R/u]F'$$

which may be rewritten

$$E'\mathbf{case}(\mathbf{case}(\mathbf{in}_1(R), [x]\mathbf{in}_1(x), [y]\mathbf{in}_2(y)), [u]P, [v]Q)F' \\ \to_\beta E'\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]Q)F' \to_\beta E'P[R/u]F'.$$

This is a sequence of β-reductions followed by an (empty) sequence of η-reductions, as desired.

In case (b), we may rewrite R as $G[t/w]$ (where w is a new variable), and the reduction sequence has the form

$$E'\mathbf{case}(\mathbf{in}_1(G[t'/w]), [u]P, [v]Q)F' \to_\eta E'\mathbf{case}(\mathbf{in}_1(G[t/w]), [u]P, [v]Q)F' \to_\beta E'P[G[t/w]/u]F'$$

which can be rewritten

$$E'\mathbf{case}(\mathbf{in}_1(G[t'/w]), [u]P, [v]Q)F' \to_\beta E'P[G[t'/w]/u]F' \to_\eta E'P[G[t/w]/u]F'$$

In case (c), our reduction sequence then has the form

$$E'\mathbf{case}(\mathbf{in}_1(R), [u]G[t'/w], [v]Q)F' \to_\eta E'\mathbf{case}(\mathbf{in}_1(R), [u]G[t/w], [v]Q)F' \to_\beta E'G[t/w][R/u]F'$$

which may be rewritten

$$E'\mathbf{case}(\mathbf{in}_1(R), [u]G[t'/w], [v]Q)F' \to_\beta E'G[t'/w][R/u]F' \to_\eta E'G[t/w][R/u]F'$$

where again we have used Lemma 2.23 to get $G[t'/w][R/u] \to_\eta G[t/w][R/u]$, using the fact that $G[t'/w] \to_\eta G[t/w]$.

In case (d) our reduction sequence has the form

$$E'\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]G[t'/w])F' \to_\eta E'\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]G[t/w])F' \to_\beta E'P[R/u]F'$$

which may be rewritten

$$E'\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]G[t'/w])F' \to_\beta E'P[R/u]F'.$$

This completes the argument for case (iii) in which s has the form $\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]Q)$. The final case in which the redex s has the form $\mathbf{case}(\mathbf{in}_2(R), [u]P, [v]Q)$ is completely symmetric.

Theorem 2.24: The Church-Rosser Theorem for β- and $\beta\eta$-reductions in λ_1.

In the untyped λ-calculus λ_1, if $X \twoheadrightarrow_\beta Y_1$ and $X \twoheadrightarrow_\beta Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$. Likewise, if $X \twoheadrightarrow_{\beta\eta} Y_1$ and $X \twoheadrightarrow_{\beta\eta} Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_{\beta\eta} Z$ and $Y_2 \twoheadrightarrow_{\beta\eta} Z$.

Proof of Theorem 2.24.

In this proof, all λ-terms are understood to be terms of the untyped λ-calculus λ_0. As before, we assume all terms obey the Barendregt variable convention. The proof follows the same strategy as the proofs of Theorems 1.13 and 1.20, with which familiarity is assumed.

For ease of reference, we list all redexes and their corresponding rules of $\beta\eta$-reduction together:

(i) the redex $\lambda x(M)(N)$ reduces to $M[N/x]$
(ii) the redex $\pi_1\langle M, N\rangle$ reduces to M
(iii) the redex $\pi_2\langle M, N\rangle$ reduces to N
(iv) the redex $\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]Q)$ reduces to $P[R/u]$
(v) the redex $\mathbf{case}(\mathbf{in}_2(R), [u]P, [v]Q)$ reduces to $Q[R/v]$
(vi) the redex $\lambda x(M(x))$ reduces to M where x is not free in M
(vii) the redex $\langle \pi_1(M), \pi_2(M)\rangle$ reduces to M
(viii) the redex $\mathbf{case}(M, [u]\mathbf{in}_1(u), [v]\mathbf{in}_2(v))$ reduces to M

We use the term 'type (n) redex' to refer to the redexes in this list. (For example, a type (vii) redex is one of the form $\langle \pi_1(M), \pi_2(M)\rangle$.)

As before, we define a more general notion of $\beta\eta$-reduction that we denote $\Rightarrow_{\beta\eta}$. To begin, note that as before for a given λ-term e the set of β- or η-redexes contained in e form a finite set of trees when ordered under the relation of inclusion (with the innermost redexes located at the uppermost leaves of these trees.) We would like it to be the case that whenever one redex r lies higher than another redex r' in one of these trees, then when the redex r' is written in the appropriate form (i)-(viii), we have that r is a subexpression of M, N, P, Q or R. This guarantees that if r is reduced, r' still maintains its original form as one of our redex types.

As in the proof of Theorem 1.20, the problem is that there are redexes r' containing redexes r as subexpressions such that r itself is not necessarily a subexpression of the relevant M, N, P, Q or R. To help us systematically identify all such cases, we begin by enumerating all subexpressions of redexes that are not subexpressions of the relevant M, N, P, Q or R.

(1) $\lambda x(M)$ is a proper subexpression of a type (i) redex
(2) $\langle M, N\rangle$ is a proper subexpression of a type (ii) or (iii) redex
(3) $\mathbf{in}_1(R)$ and $\mathbf{in}_2(R)$ are proper subexpressions of type (iv) and (v) redexes
(4) $M(x)$ is a proper subexpression of a type (vi) redex
(5) $\pi_1(M)$ and $\pi_2(M)$ are proper subexpression of a type (vii) redex
(6) $\mathbf{in}_1(u)$ and $\mathbf{in}_2(u)$ are proper subexpression of a type (viii) redex

We consider each of these cases in turn. In case (1), the only way $\lambda x(M)$ can be a redex is if it is a type (vi) redex. In case (2), the only way a term $\langle M, N\rangle$ can be a redex is if it is a type (vii) redex. In case (3), the terms $\mathbf{in}_1(R)$ and $\mathbf{in}_2(R)$ never have the form of a redex. In case (4), the only way a term $M(x)$ can be a redex is if it is a type (i) redex. In case (5), the only way $\pi_1(M)$ or $\pi_2(M)$ can be a redex is if it is a type (ii) or (iii) redex. And in case (6), $\mathbf{in}_1(u)$ and $\mathbf{in}_2(u)$ can never have the form of a redex.

Thus, the redexes and corresponding subredexes we need to pay special attention to are the following, listed with their types and with the problematic subredexes explicitly indicated

(a) the type (i) redex $\quad \underbrace{\lambda x(\overline{M}(x))}_{\eta\text{-redex}}(N)$

(b) the type (ii) or (iii) redex $\quad \pi_i \underbrace{\langle \pi_1(M), \pi_2(M)\rangle}_{\eta\text{-redex}}$

(c) the type (vi) redex $\quad \lambda x(\underbrace{\lambda z(\overline{M})(x)}_{\beta\text{-redex}})$

(d) the type (vii) redex $\quad \langle \underbrace{\pi_1\langle M, N\rangle}_{\beta\text{-redex}}, \underbrace{\pi_2\langle M, N\rangle}_{\beta\text{-redex}}\rangle$

In **(a)**, it is assumed that x is not free in $\overline{M}$, and in **(c)** it is assumed that x is not free in $\lambda z(\overline{M})$. These redexes correspond to redex trees that contain the following structures

$$\begin{array}{cccc} \lambda x(\overline{M}(x)) & \langle \pi_1(M), \pi_2(M)\rangle & \lambda x(\overline{M}(x)) & \pi_1\langle M, N\rangle \quad \pi_2\langle M, N\rangle \\ | & | & | & \searrow \quad \swarrow \\ \lambda x(\overline{M}(x))(N) & \pi_i\langle \pi_1(M), \pi_2(M)\rangle & \lambda x(\overline{M}(x))(N) & \langle \pi_1\langle M, N\rangle, \pi_2\langle M, N\rangle\rangle \end{array}$$

In analogy with the proof of Theorem 1.20, we declare redexes of the form **(a)-(d)** *prohibited*. If we perform a 1-step reduction on any non-prohibited redex, then any other non-prohibited redex below it maintains its form as a redex. We can therefore perform a 1-step β- or η-reduction on as many (or as few) of the non-prohibited redexes contained in the redex trees

of e that we want, working from the innermost redexes outwards – that is, working from the uppermost redexes downwards. As before, if doing so produces a term e', we will say that $e \Rightarrow_\beta e'$. If in this way we reduce *all* the non-prohibited redexes contained in e, we call the resulting term e^* (or $[e]^*$), which we call the *complete reduction* of e. (Again, the order in which we perform β-reductions associated with disjoint redexes does not matter.) By definition, we then have $e \Rightarrow_\beta e^*$.

We now have a sequence of lemmas to those contained in the proofs of Theorem 1.13 and 1.20. Recall that $s \rightarrow_{\beta\eta} t$ means $s \rightarrow_\beta t$ or $s \rightarrow_\eta t$.

Lemma 1: For any λ-terms s and t, $s \rightarrow_{\beta\eta} t$ entails $s \Rightarrow_{\beta\eta} t$.

Proof: The reduction $s \rightarrow_{\beta\eta} t$ corresponds to the case in which a single redex r in the redex trees of s is reduced, with the result being t. If the redex r is not prohibited, then it is immediate that $s \Rightarrow_{\beta\eta} t$.

In order to deal with the case in which r is prohibited, we argue for the following slightly modified version of (23) used in the proof of Theorem 1.20

> If $s \rightarrow_{\beta\eta} t$ via the reduction of a prohibited redex r, then there are disjoint redexes $r'_1, ..., r'_n$ properly contained in r such that $s \rightarrow_{\beta\eta} t$ via the reduction of the redexes $r'_1, ..., r'_n$. (56)

To argue for this principle, it suffices to consider the four possible forms **(a)**, **(b)**, **(c)** or **(d)** that the prohibited redex r might have. In the case in which r has the form **(a)** or **(c)**, the analysis given in the discussion of (23) in the proof of Theorem 1.20 shows that (56) holds.

In case **(b)**, r has the form of the β-redex $\pi_i\langle\pi_1(M), \pi_2(M)\rangle$. This β-reduces to $\pi_i(M)$. Note however that if in r the η-redex $r' = \langle\pi_1(M), \pi_2(M)\rangle$ is η-reduced, the result is also $\pi_i(M)$, and so (56) holds.

In case **(d)**, r has the form of the η-redex $\langle\pi_1\langle M, N\rangle, \pi_2\langle M, N\rangle\rangle$. This η-reduces to $\langle M, N\rangle$. Note however that if in r the disjoint β-redexes $r'_1 = \pi_1\langle M, N\rangle$ and $r'_2 = \pi_2\langle M, N\rangle$ are β-reduced, the result is also $\langle M, N\rangle$, and so (56) holds.

Thus, (56) holds in general. Of course, any one of the redexes $r'_1, ..., r'_n$ given in (56) may also be prohibited. In this case, the principle (56) can be applied again to any such prohibited r'_i. Because we cannot have an infinite sequence $u, u', u'', ...$ of redexes, each a subredex of the previous one on the list, it follows that if $s \rightarrow_{\beta\eta} t$ via the reduction of a prohibited redex r, then there must be a set of disjoint, non-prohibited redexes $r^*_1, ..., r^*_m$ properly contained in r such that $s \rightarrow_{\beta\eta} t$ via the reduction of the redexes $r^*_1, ..., r^*_m$. Thus, $s \Rightarrow_{\beta\eta} t$. □

Lemma 2: For any λ-terms s and t, $s \Rightarrow_{\beta\eta} t$ entails $s \twoheadrightarrow_{\beta\eta} t$.

Proof: Identical to that given in the proof of Theorem 1.13 □

Lemma 3: For any λ-terms s, s', t, t', if $s \Rightarrow_{\beta\eta} s'$ and $t \Rightarrow_{\beta\eta} t'$, then $s[t/v] \Rightarrow_{\beta\eta} s'[t'/v]$.

Proof: Identical to that given in the proof of Theorem 1.20 □

Lemma 4: For any λ-terms s and t, if $s \Rightarrow_{\beta\eta} t$ then $t \Rightarrow_{\beta\eta} s^*$.

Proof: As in the proof of Theorem 1.20, we use the notation $W(X, Y, ...)$ to compactly denote

redexes and $W^{\mathbf{r}}(X, Y, ...)$ their reductions as follows

type **(i)** redex:	$W(M,N) = \lambda x(M)(N)$	$W^{\mathbf{r}}(M,N) = M[N/x]$
type **(ii)** redex:	$W(M,N) = \pi_1\langle M,N\rangle$	$W^{\mathbf{r}}(M,N) = M$
type **(iii)** redex:	$W(M,N) = \pi_2\langle M,N\rangle$	$W^{\mathbf{r}}(M,N) = N$
type **(iv)** redex:	$W(P,Q,R) = \mathbf{case}(\mathbf{in}_1(R),[u]P,[v]Q)$	$W^{\mathbf{r}}(P,Q,R) = P[R/u]$
type **(v)** redex:	$W(P,Q,R) = \mathbf{case}(\mathbf{in}_2(R),[u]P,[v]Q)$	$W^{\mathbf{r}}(P,Q,R) = Q[R/v]$
type **(vi)** redex:	$W(M) = \lambda x(M(x))$	$W^{\mathbf{r}}(M,N) = M$
type **(vii)** redex:	$W(M) = \langle\pi_1(M),\pi_2(M)\rangle$	$W^{\mathbf{r}}(M,N) = M$
type **(viii)** redex:	$W(M) = \mathbf{case}(M,[u]\mathbf{in}_1(u),[v]\mathbf{in}_2(v))$	$W^{\mathbf{r}}(M,N) = M$

in type **(vi)** redexes, x is of course assumed not to be free in M.

First we show

(P1) If a redex $r = W(X, Y, ...)$ is not prohibited and another redex r' is properly contained in r, then r' is properly contained in one of $X, Y, ...$

(P2) For any redex $W(X, Y, ...)$, if $X \Rightarrow_{\beta\eta} X', Y \Rightarrow_{\beta\eta} Y', ...$ then

$$W^{\mathbf{r}}(X, Y, ...) \Rightarrow_{\beta\eta} W^{\mathbf{r}}(X', Y', ...).$$

The property **(P1)** follows from our characterization of prohibited redexes. For a type **(i)** redex, property **(P2)** amounts to the claim that if $M \Rightarrow_{\beta\eta} M'$ and $N \Rightarrow_{\beta\eta} N'$, then $M[N/x] \Rightarrow_{\beta\eta} M'[N'/x]$, which follows from Lemma 3. Likewise for type **(iv)** and **(v)** redexes. For a type **(ii)** redex, property **(P2)** amounts to the trivial claim that if $M \Rightarrow_{\beta\eta} M'$ and $N \Rightarrow_{\beta\eta} N'$, then $M \Rightarrow_{\beta\eta} M'$. Likewise for type **(iii)**, **(vi) (vii)**, and **(viii)** redexes. So both **(P1)** and **(P2)** are true in general. Having established these, the remainder of the proof of Lemma 4 is identical to that given in Theorem 1.20. □

Given Lemma 4, the proof of Lemma 5 and remainder of the proof is the same as before, with only trivial notational changes. □

Theorem 2.26: The Church-Rosser Theorem for β- and $\beta\eta$-reductions in $\lambda_1^{\mathbf{type}}$.

In the typed λ-calculus λ_1^{type}, if $X \twoheadrightarrow_\beta Y_1$ and $X \twoheadrightarrow_\beta Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_\beta Z$ and $Y_2 \twoheadrightarrow_\beta Z$. Likewise, if $X \twoheadrightarrow_{\beta\eta} Y_1$ and $X \twoheadrightarrow_{\beta\eta} Y_2$, then there is a term Z such that $Y_1 \twoheadrightarrow_{\beta\eta} Z$ and $Y_2 \twoheadrightarrow_{\beta\eta} Z$.

Proof of Theorem 2.26.

To prove Theorems 2.26, it suffices to use the proof of Theorems 2.24 and and note that noth-

ing more than notational alterations are required in the typed case. In particular, every term used in these proofs can be typed appropriately if required. □

Theorem 2.28: Strong Normalizability of λ_1^{type}.

There is no infinite sequence $\tau_1, \tau_2, ...$ of terms in the typed λ-calculus λ_1^{type} for which

$$\tau_1 \to_\beta \tau_2 \to_\beta \tau_3 \to_\beta ...$$

Proof of Theorem 2.28.

This proof is adapted from Wojdyga [11]. We show that if there were an infinite sequence of terms $\tau_1, \tau_2, ...$ in the typed λ-calculus λ_1^{type} for which $\tau_1 \to_\beta \tau_2 \to_\beta \tau_3 \to_\beta ...$, then there would also be an infinite sequence of terms $\tau_1', \tau_2', ...$ in the typed λ-calculus λ_0^{type} for which $\tau_1' \to_{\beta\eta} \tau_2' \to_{\beta\eta} \tau_3' \to_{\beta\eta} ...$, thereby contradicting the strong normalizability result for λ_0^{type} proved in Theorem 1.33.

Pick a fundamental simple type of λ_0^{type} and denote it $\perp$. To each type P of λ_1^{type} (i.e., each propositional type) we associate a type $\|P\|$ of λ_0^{type} (i.e., a simple type) by applying the following rules:

- $\|P\| = \perp$ if M is a fundamental type or 0
- $\|P_1 \to P_2\| = \|P_1\| \to \|P_2\|$
- $\|P_1 \times P_2\| = (\|P_1\| \to (\|P_2\| \to \perp)) \to \perp$
- $\|P_1 + P_2\| = (\|P_1\| \to \perp) \to ((\|P_2\| \to \perp) \to \perp)$

We then have

Lemma 1: For each type P of λ_1^{type}, $\|P\|$ can be written uniquely in the form

$$P_1 \to (P_2 \to (... \to (P_n \to \perp)...)).$$

Proof: A straightforward induction on the construction of P. □

To each λ-term τ of λ_1^{type}, we associate a λ-term $\|\tau\|$ of λ_0^{type} by applying the following rules

(1) $\|v^\tau\| = v^{\|\tau\|}$
(2) $\|M^{\alpha\to\beta}(N^\alpha)\| = \|M^{\alpha\to\beta}\|(\|N^\alpha\|)$
(3) $\|\lambda x^\alpha(M^\beta)\| = \lambda x^{\|\alpha\|}(\|M^\beta\|)$
(4) $\|\langle M^\alpha, N^\beta\rangle\| = \lambda z^{\|\alpha\|\to(\|\beta\|\to\perp)}(z(\|M^\alpha\|)(\|N^\beta\|))$
(5) $\|\pi_1(M^{\alpha\times\beta})\| = \lambda z_1^{\alpha_1}...\lambda z_n^{\alpha_n}(\|M^{\alpha\times\beta}\|(\lambda u^{\|\alpha\|}\lambda v^{\|\beta\|}(uz_1...z_n)))$
where $\|\alpha\| = \alpha_1 \to (\alpha_2 \to (... \to (\alpha_n \to \perp)...))$.

(6) $\|\pi_2(M^{\alpha\times\beta})\| = \lambda z_1^{\beta_1}...\lambda z_n^{\beta_n}(\|M^{\alpha\times\beta}\|(\lambda u^{\|\alpha\|}\lambda v^{\|\beta\|}(vz_1...z_n)))$
where $\|\beta\| = \beta_1 \to (\beta_2 \to (... \to (\beta_n \to \bot)...))$.
(7) $\|\mathbf{in}_1^{\alpha,\beta}(M^\alpha)\| = \lambda u^{\|\alpha\|\to\bot}\lambda v^{\|\beta\|\to\bot}(u\|M^\alpha\|)$
(8) $\|\mathbf{in}_2^{\alpha,\beta}(N^\beta)\| = \lambda u^{\|\alpha\|\to\bot}\lambda v^{\|\beta\|\to\bot}(v\|N^\beta\|)$
(9) $\|\mathbf{case}(R^{\alpha+\beta},[u^\alpha][P]^\gamma,[v^\beta][Q]^\gamma)\| =$
$\lambda z_1^{\gamma_1}...\lambda z_n^{\gamma_n}(\|R^{\alpha+\beta}\|(\lambda u^{\|\alpha\|}(\|P^\gamma\|z_1...z_n))(\lambda v^{\|\beta\|}(\|Q^\gamma\|z_1...z_n)))$
where $\|\gamma\| = \gamma_1 \to (\gamma_2 \to (... \to (\gamma_n \to \bot)...))$.
(10) $\|\mathbf{abs}^\alpha([M]^0)\| = \lambda z_1^{\alpha_1}...\lambda z_n^{\alpha_n}(\|M^0\|)$
where $\|\alpha\| = \alpha_1 \to (\alpha_2 \to (... \to (\alpha_n \to \bot)...))$.

where the bound variables $z, z_1, ..., z_n, u, v$ introduced on the right hand side of these expressions are assumed to not appear anywhere on the left hand side. (Again, we do not write all the superscripts in these typed λ-terms, but include enough superscripts that the type of any subexpression may be easily determined.)

We then have

Lemma 2: For each λ-term P of λ_1^{type} of type τ, $\|P\|$ is a term of λ_0^{type} of type $\|\tau\|$.

Proof: By induction on the construction of P. There are 10 cases to consider, corresponding to each of the 10 cases in the definition of $\|\tau\|$ just given. The arguments are all straightforward.

For an easy example, in case **(2)** we must show that $\|M^{\alpha\to\beta}(N^\alpha)\|$ is a λ_0^{type} term of type $\|\beta\|$. By inductive hypothesis $\|M^{\alpha\to\beta}\|$ and $\|N^\alpha\|$ are λ_0^{type} terms of type $\|\alpha \to \beta\|$ and $\|\beta\|$ respectively. The type $\|\alpha \to \beta\|$ is by definition $\|\alpha\| \to \|\beta\|$. It follows immediately that $\|M^{\alpha\to\beta}(N^\alpha)\| = \|M^{\alpha\to\beta}\|(\|N^\alpha\|)$ is a λ_0^{type} term of type $\|\beta\|$.

For a harder example, consider case **(5)**. We must show that $\|\pi_1(M^{\alpha\times\beta})\|$ is a λ_0^{type} term of type $\|\alpha\|$. If $\|\alpha\| = \alpha_1 \to (\alpha_2 \to (... \to (\alpha_n \to \bot)...))$, then if u has type $\|\alpha\|$ and $z_1, ..., z_n$ have types $\alpha_1, ..., \alpha_n$, then $uz_1...z_n$ is a term of type $\bot$. Thus $\lambda u^{\|\alpha\|}\lambda v^{\|\beta\|}(uz_1...z_n))$ is a term of type $\|\alpha\| \to (\|\beta\| \to \bot)$. By inductive hypothesis, $\|M^{\alpha\times\beta}\|$ has type $\|\alpha \times \beta\|$ which is by definition $(\|\alpha\| \to (\|\beta\| \to \bot)) \to \bot$, and so $\|M^{\alpha\times\beta}\|(\lambda u^{\|\alpha\|}\lambda v^{\|\beta\|}(uz_1...z_n))$ has type $\bot$. Because $z_1, ..., z_n$ have types $\alpha_1, ..., \alpha_n$, it then follows that $\lambda z_1^{\alpha_1}...\lambda z_n^{\alpha_n}(\|M^{\alpha\times\beta}\|(\lambda u^{\|\alpha\|}\lambda v^{\|\beta\|}(uz_1...z_n)))$ has type $\alpha_1 \to (\alpha_2 \to (... \to (\alpha_n \to \bot)...))$, i.e., type $\|\alpha\|$. Thus $\|\pi_1(M^{\alpha\times\beta})\|$ is a λ_0^{type} term of type $\|\alpha\|$.

Straightforward reasoning of this sort suffices for verifying the remaining 8 cases. □

The following lemma also holds

Lemma 3: If M^τ, N^σ are λ_1^{type} terms and x^σ a variable, then $\|M^\tau\|[\|N^\sigma\|/x^{\|\sigma\|}] = \|M^\tau[N^\sigma/x^\sigma]\|$.

Proof: Straightforward induction on the construction of the λ-term M^τ. □.

If σ and τ are λ-terms of λ_0^{type}, we write $\sigma \twoheadrightarrow_{\beta\eta}^+ \tau$ just in case there is a $\beta\eta$-reduction of σ to τ involving at least one step (i.e., there is one such reduction which is not a trivial 0-step reduction.) We then have

Lemma 4: For terms M, N of λ_1^{type}, if $M \to_\beta N$, then $\|M\| \twoheadrightarrow_{\beta\eta}^+ \|N\|$.

Proof: We consider each of the 5 β-reduction rules of λ_1^{type} in turn. First, consider the reduction rule

$$\lambda x^\sigma(M^\tau)(N^\sigma) \to_\beta M^\tau[N^\sigma/x^\sigma].$$

Then

$$\|\lambda x^{\sigma}(M^{\tau})(N^{\sigma})\| = \|\lambda x^{\sigma}(M^{\tau})\|(\|N^{\sigma}\|) = \lambda x^{\|\sigma\|}(\|M^{\tau}\|)(\|N^{\sigma}\|)$$
$$\rightarrow_{\beta} \|M^{\tau}\|[\|N^{\sigma}\|/x] = \|M^{\tau}[N^{\sigma}/x]\|$$

as desired. Next consider the reduction rule

$$\pi_1(\langle M^{\sigma}, N^{\tau}\rangle^{\sigma\times\tau}) \rightarrow_{\beta} M^{\sigma}$$

Then

$$\begin{aligned}\|\pi_1(\langle M^{\sigma}, N^{\tau}\rangle^{\sigma\times\tau})\| &= \lambda z_1^{\sigma_1}...\lambda z_n^{\sigma_n}(\|\langle M^{\sigma}, N^{\tau}\rangle\|(\lambda u^{\|\sigma\|}\lambda v^{\|\tau\|}(uz_1...z_n)))\\ &= \lambda z_1^{\sigma_1}...\lambda z_n^{\sigma_n}(\lambda z^{\|\sigma\|\rightarrow(\|\tau\|\rightarrow\perp)}(z(\|M^{\sigma}\|)(\|N^{\tau}\|))(\lambda u^{\|\sigma\|}\lambda v^{\|\tau\|}(uz_1...z_n)))\\ &\rightarrow_{\beta} \lambda z_1^{\sigma_1}...\lambda z_n^{\sigma_n}(\lambda u^{\|\sigma\|}\lambda v^{\|\tau\|}(uz_1...z_n)(\|M^{\sigma}\|)(\|N^{\tau}\|))\\ &\twoheadrightarrow_{\beta} \lambda z_1^{\sigma_1}...\lambda z_n^{\sigma_n}(\|M^{\sigma}\|z_1...z_n) \rightarrow_{\eta} \lambda z_1^{\sigma_1}...\lambda z_{n-1}^{\sigma_{n-1}}(\|M^{\sigma}\|z_1...z_{n-1}) \rightarrow_{\eta} ... \rightarrow_{\eta} \|M^{\sigma}\|\end{aligned}$$

where $\|\sigma\| = \sigma_1 \rightarrow (\sigma_2 \rightarrow (... \rightarrow (\sigma_n \rightarrow \perp)...))$, as desired. The argument for the reduction rule

$$\pi_2(\langle M^{\sigma}, N^{\tau}\rangle^{\sigma\times\tau}) \rightarrow_{\beta} N^{\tau}$$

is symmetric. Next consider the reduction rule

$$\mathbf{case}(\mathbf{in}_1^{\alpha,\beta}(S^{\alpha})^{\alpha+\beta}, [u^{\alpha}]P^{\gamma}, [v^{\beta}]Q^{\gamma}) \rightarrow_{\beta} P^{\gamma}[S^{\alpha}/u^{\alpha}]$$

Then

$$\begin{aligned}&\|\mathbf{case}(\mathbf{in}_1^{\alpha,\beta}(S^{\alpha})^{\alpha+\beta}, [u^{\alpha}]P^{\gamma}, [v^{\beta}]Q^{\gamma})\|\\ &\quad= \lambda z_1^{\gamma_1}...\lambda z_n^{\gamma_n}(\|\mathbf{in}_1^{\alpha,\beta}(S^{\alpha})^{\alpha+\beta}\|(\lambda u^{\|\alpha\|}(\|P^{\gamma}\|z_1...z_n))(\lambda v^{\|\beta\|}(\|Q^{\gamma}\|z_1...z_n)))\\ &= \lambda z_1^{\gamma_1}...\lambda z_n^{\gamma_n}(\lambda \overline{u}^{\|\alpha\|\rightarrow\perp}\lambda \overline{v}^{\|\beta\|\rightarrow\perp}(\overline{u}\|S^{\alpha}\|)(\lambda u^{\|\alpha\|}(\|P^{\gamma}\|z_1...z_n))(\lambda v^{\|\beta\|}(\|Q^{\gamma}\|z_1...z_n)))\\ &\twoheadrightarrow_{\beta} \lambda z_1^{\gamma_1}...\lambda z_n^{\gamma_n}(\lambda u^{\|\alpha\|}(\|P^{\gamma}\|z_1...z_n)\|S^{\alpha}\|) \rightarrow_{\beta} \lambda z_1^{\gamma_1}...\lambda z_n^{\gamma_n}(\|P^{\gamma}\|[\|S^{\alpha}\|/u^{\|\alpha\|}]z_1...z_n)\\ &\qquad\qquad\twoheadrightarrow_{\eta} \|P^{\gamma}\|[\|S^{\alpha}\|/u^{\|\alpha\|}] = \|P^{\gamma}[S^{\alpha}/u^{\alpha}]\|\end{aligned}$$

where $\|\gamma\| = \gamma_1 \rightarrow (\gamma_2 \rightarrow (... \rightarrow (\gamma_n \rightarrow \perp)...))$, as desired. The argument for the reduction rule

$$\mathbf{case}(\mathbf{in}_2^{\alpha,\beta}(S^{\beta})^{\alpha+\beta}, [u^{\alpha}]P^{\gamma}, [v^{\beta}]Q^{\gamma}) \rightarrow_{\beta} Q^{\gamma}[S^{\beta}/v^{\beta}]$$

is symmetric. □

To prove the strong normalizability of λ_1^{type} under β-reductions, suppose to the contrary that there is an infinite sequence $\tau_1, \tau_2, ...$ of simply typed λ-terms such that

$$\tau_1 \to_\beta \tau_2 \to_\beta \tau_3 \to_\beta ...$$

Then by Lemma 4,

$$\|\tau_1\| \twoheadrightarrow^+_{\beta\eta} \|\tau_2\| \twoheadrightarrow^+_{\beta\eta} \|\tau_3\| \twoheadrightarrow^+_{\beta\eta} ...$$

contradicting the strong normalizability result for λ_0^{type} proved in Theorem 1.33.

Theorem 2.31: Soundness and Completeness of $\mathbf{TR_1}$.

Suppose E is an untyped λ_1-term whose free variables are $v_1, ..., v_n$, and let $\tau_1, ..., \tau_n$ be types. Then the following are equivalent:

(i) It is possible to assign types of λ_1^{type} to all the variables of E in such a way that the free variables $v_1, ..., v_n$ of E are assigned types $\tau_1, ..., \tau_n$, and E itself has type X.

(ii) There is a proof of $\Gamma \vdash E : X$ in $\mathbf{TR_1}$, where the context Γ includes the variable declarations $v_1 : \tau_1,\ v_2 : \tau_2, ...,\ v_n : \tau_n$.

Proof of Theorem 2.31.

First we prove **(i)** $\to$ **(ii)**. The proof is by induction on the construction of the λ-term E. If E is a variable, or has one of the forms $E_1(E_2)$ or $\lambda v(E)$, then the argument given in the proof of Theorem 1.43 suffices.

Suppose E has the form $\langle E_1, E_2\rangle$, and that by assigning the free variables $v_1, ..., v_n$ of $\langle E_1, E_2\rangle$ the types $\tau_1, ..., \tau_n$ respectively, E can be assigned the type $X \times Y$. Then by inductive hypothesis we have

$$\Gamma_1 \vdash E_1 : X \quad \text{and} \quad \Gamma_2 \vdash E_2 : Y$$

for some contexts Γ_1, Γ_2 such that for any j, if v_j is free in E_1 (resp. E_2), then $v_j : \tau_j$ appears in Γ_1 (resp. Γ_2). Letting $\overline{E}_1$ and $\overline{E}_2$ be λ-terms obeying the Barendregt variable convention and α-equivalent to E_1 and E_2, by Theorem 2.30 there are proofs

$$\Gamma_1' \vdash \overline{E}_1 : X \quad \text{and} \quad \Gamma_1' \vdash \overline{E}_2 : Y$$

such that Γ_1' (resp. Γ_2') is just the set of typing declarations $v_j : \tau_j$ for the v_j that are free in $\overline{E}_1$ (resp. $\overline{E}_2$.) Let Γ be the context

$$v_1 : \tau_1, ..., v_n : \tau_n.$$

By adding additional variable declarations to the contexts of these proofs, we have

$$\Gamma \vdash \overline{E}_1 : X \quad \text{and} \quad \Gamma \vdash \overline{E}_2 : Y.$$

from which we may infer $\Gamma \vdash \langle E_1, E_2 \rangle : X \times Y$ (where this may involve silent renaming of variables). So **(i)** $\rightarrow$ **(ii)** holds in this case.

Suppose E has the form $\pi_1(E')$, and that by assigning the free variables $v_1, ..., v_n$ of $\pi_1(E')$ the types $\tau_1, ..., \tau_n$ respectively, $\pi_1(E')$ can be assigned type X, with E' being assigned type $X \times Y$. Then by inductive hypothesis, we have

$$\Gamma \vdash E' : X \times Y$$

where Γ includes the variable declarations $v_1 : \tau_1, ..., v_n : \tau_n$. We then immediately have

$$\Gamma \vdash \pi_1(E') : X$$

as desired, and so **(i)** $\rightarrow$ **(ii)** holds in this case.

Similarly straightforward arguments of this sort can be used to deal with the cases in which E has the form $\pi_2(E'), \mathbf{in}_1(E'), \mathbf{in}_2(E')$ or $\mathbf{abs}(E')$.

Suppose next that E has the form $\mathbf{case}(\mathbf{in}_1(R), [u]P, [v]Q)$. Without loss of generality, we assume that E obeys the Barendregt variable convention. Assume that by assigning the free variables $v_1, ..., v_n$ of E the types $\tau_1, ..., \tau_n$ respectively, E can be assigned the type Z. Suppose that in this case R is assigned type $X + Y$, P is assigned type Z, and Q is assigned type Z. Then by inductive hypothesis and Theorem 2.30 we have

$$\Gamma_1 \vdash R : X + Y \quad \text{and} \quad \Gamma_2, u : X \vdash P : Z \quad \text{and} \quad \Gamma_3, v : Y \vdash Q : Z$$

where each of Γ_1, Γ_2 and Γ_3 consists of the variable declarations $v_j : \tau_j$ for any v_j that appears free in R, P and Q respectively. Because E (and thus all its subterms) obeys the Barendregt variable convention, neither u or v appear among the v_j, and so the contexts $\Gamma_2, u : X$ and $\Gamma_3, v : Y$ are grammatical. We thus obtain

$$\Gamma_1, \Gamma_2, \Gamma_3 \vdash \mathbf{case}(\mathbf{in}_1(R), [u]P, [v]Q) : Z$$

as desired. The case in which E has the form $\mathbf{case}(\mathbf{in}_2(R), [u]P, [v]Q)$ is of course symmetrical. This completes the argument that **(i)** $\rightarrow$ **(ii)**.

Now we prove **(ii)** $\rightarrow$ **(i)**. The proof is by induction on the construction of proofs. If $\Gamma \vdash E : X$ has the form $\Gamma', v : X \vdash v : X$, or is the result of an (App) or (Abs) inference, then the argument given in the proof of Theorem 1.43 suffices. The proofs for the remaining cases are all similar, and we simply consider the most complex case in which the proof is the result of a **case** inference

$$\frac{\Gamma \vdash R : \sigma + \tau \qquad \Gamma, u : \sigma \vdash P : \rho \qquad \Gamma, v : \tau \vdash Q : \rho}{\Gamma \vdash \mathbf{case}(R, [u^\sigma]P, [v^\tau]Q) : \rho}$$

Assume then that we have

$$\Gamma \vdash \mathbf{case}(R, [u^\sigma]P, [v^\tau]Q) : \rho,$$

where if $v_1, ..., v_n$ are the free variables of $\mathbf{case}(R, [u^\sigma]P, [v^\tau]Q)$, then Γ includes typing declarations $v_1 : \tau_1, ..., v_n : \tau_n$. Suppose that the last step in the proof of this sequent is indeed the **case** rule, so that we have

$$\Gamma \vdash R : \sigma + \tau \quad \text{and} \quad \Gamma, u : \sigma \vdash P : \rho \quad \text{and} \quad \Gamma, v : \tau \vdash Q : \rho$$

Then by inductive hypothesis, consistent with the free variables of R being assigned types in accordance with Γ, theterm R may be assigned the type $\sigma + \tau$, consistent with the free variables of P being assigned types in accordance with $\Gamma, u : \sigma$, the term P may be assigned the type ρ, and consistent with the free variables of Q being assigned types in accordance with $\Gamma, v : \tau$, the term Q may be assigned the type ρ. It follows that consistent with the free variables of $\mathbf{case}(R, [u^\sigma]P, [v^\tau]Q)$ being assigned types in accordance with Γ, this term may be assigned the type ρ, as desired. Thus **(ii)** $\rightarrow$ **(i)** holds.

Theorem 2.32 Soundness and Completeness of $\mathbf{TR_1^{bt}}$.

Suppose E is a bound-typed λ_1-term whose free variables are $v_1, ..., v_n$, and let $\tau_1, ..., \tau_n$ be types. Then the following are equivalent:

(i) If the variables $v_1, v_2, ..., v_n$ are assigned types $\tau_1, \tau_2, ..., \tau_n$ respectively, then E must be assigned type σ.

(ii) There is a proof of $\Gamma \vdash E : X$ in $\mathbf{TR_1^{bt}}$, where the context Γ includes the variable declarations $v_1 : \tau_1,\ v_2 : \tau_2, ...,\ v_n : \tau_n$.

Proof of Theorem 2.32.

This proof is largely analogous to that of Theorem 2.31 and so is omitted.

Bibliography

[1] H. Barendregt. *The Lambda Calculus, its Syntax and Semantics*. North-Holland, 1984.

[2] H. Barendregt. The impact of the lambda calculus in logic and computer science. *The Bulletin of Symbolic Logic*, 3(2):181–215, Jun 1997.

[3] G. Boolos, J. Burgess, and R Jeffrey. *Computability and Logic*. Cambridge University Press, 2007.

[4] J. Hindley and J. Seldin. *Lambda-Calculus and Combinators, An Introduction*. Cambridge University Press, 2008.

[5] Hindley J. *Basic Simple Type Theory*. Cambridge University Press, 1997.

[6] G. Michaelson. *Functional Programming Through Lambda Calculus*. Dover Publications, 1989.

[7] G. Mints. *A Short Introduction to Intuitionistic Logic*. Kluwer Academic Publishers, 2000.

[8] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[9] P. Selinger. *Lecture Notes on the Lambda Calculus*. Lulu.com, 2018.

[10] M. Sorensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*. Elsevier Science, 2006.

[11] A. Wojdyga. Short proofs of strong normalization. In E. Ochmański and J. Tyszkiewicz, editors, *Mathematical Foundations of Computer Science 2008. Lecture Notes in Computer Science*, volume 5162, pages 612–623, 2008.