Introduction to Lambda Calculus

Aaron Stump
Computer Science
The University of Iowa

April 12, 2024

Contents

Part I

Untyped Lambda Calculus

Chapter 1

Introduction

The formal system known as lambda calculus was invented by Alonzo Church, and published first in his paper "A Set of Postulates for the Foundation of Logic" [4]. As that title suggests, Church's motivation for devising lambda calculus was to create a formal foundation for logic and mathematics. This was in response to the crisis in foundations of mathematics that occurred in the early 20th century, with the discovery of paradoxes in proposed foundational theories. Bertrand Russell's discovery, in 1901, of a contradiction in the foundational theory being developed by Gottlob Frege was a prime and motivating example [13]. Church's own theory was quickly discovered to be inconsistent, as, sadly, was even a revised version [5].

One may take these failures as as a cautionary tale of the difficulty of creating consistent foundational theories. Or, more inspiringly, one can understand them as showing that many good results can come from endeavors that fall short of their objectives. For from these early systems of Church, and the work he and his brilliant graduate students carried out consequently, has arisen a remarkable line of inquiry, with tremendous theoretical and practical impact, on the subjects of typed and untyped lambda calculus. For an engaging exposition of this history, see the paper by Cardone and Hindley [3].

Church subsequently published a research monograph focused on the lambda calculus as a formal notion of computation, rather than a foundation for mathematics [6]. I will take this monograph to be his definitive presentation of lambda calculus.

1.1 Why this book

There are a number of very impressive books on lambda calculus currently available. For example, Hendrik Barendregt's book remains an authoritative source for many deep topics in the theory of untyped lambda calculus [1], and his more recent book, co-authored with Will Dekkers and Richard Statman, is a similar source for certain topics in typed lambda calculus [2]. But these are reference works, which are far too advanced to serve as textbooks. One book on lambda calculus that is at an appropriate level for university instruction is the one by J. Roger Hindley and Jonathan Seldin [8]. But this book has a more mathematical perspective on the subject, and puts less emphasis on certain more computational points. So in my opinion, there is currently no available textbook, for students at the late undergraduate or early graduate level, on lambda calculus from the perspective of Computer Science. And that is what the current volume seeks to supply.

Chapter 2

Syntax and Reduction Semantics

2.1 The syntax of lambda terms

The untyped lambda calculus has a very small syntax, an appealing quality for both theoretical study and practical use. There are just two syntactic categories: variables and terms. We assume variables are taken from some countably infinite set of mathematical objects (like the natural numbers) disjoint from the other forms of term, and generally avoid specifying them further. One stipulation, though is that whatever variables are, we should be able to test effectively whether or not two variables are equal. Such an equality test will be needed when defining substitution. We will use x as a meta-variable for variables, and adopt the convention that in any particular meta-linguistic discussion, different such meta-variables refer to different variables. So if we write x and y, please understand that these meta-variables refer to different actual variables.

Terms are then certain labeled binary trees. We will use the meta-variable t (and decorated versions like t_1, t', etc.) for terms. They are either leaf nodes labeled with variables x, application nodes with subtrees t_1 and t_2, or lambda nodes with subtree labeled x and subtree t'. Pictorially, these are shown in Figure 2.1. An application node is said to apply t_1, as a function, to t_2, as an argument to that function. For a lambda node with subtree labeled x and subtree t', the term t' is called the *body* of the λ-node. A term whose root is labeled λ is called a λ-abstraction.

While understanding the tree structure for terms is essential for the study of lambda calculus, it is typical to present terms not as trees, but textually. For this, we use the following context-free grammar, together with two parsing conventions:

$$\textit{terms}\ t\ ::=\ x \mid t_1\ t_2 \mid \lambda\, x.\, t \mid (t)$$

The parsing conventions are:

1. Application is left-associative. So $a\ b\ c$ should be interpreted as $(a\ b)\ c$, as opposed to $a\ (b\ c)$.

2. The scope of $\lambda\, x$ is as far to the right as possible. So $\lambda\, x.\, x\ x$ should be interpreted as $\lambda\, x.\, (x\ x)$, because this grouping shows that the $\lambda\, x$ part of the term governs the application $x\ x$. This disambiguation is used, instead of $(\lambda\, x.\, x)\ x$.

Parentheses are just used for disambiguation and do not correspond to any node in the tree syntax (of Figure 2.1).

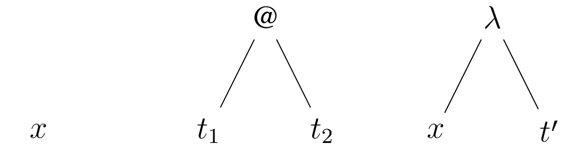


Figure 2.1: Graphical depiction of the syntax of terms t

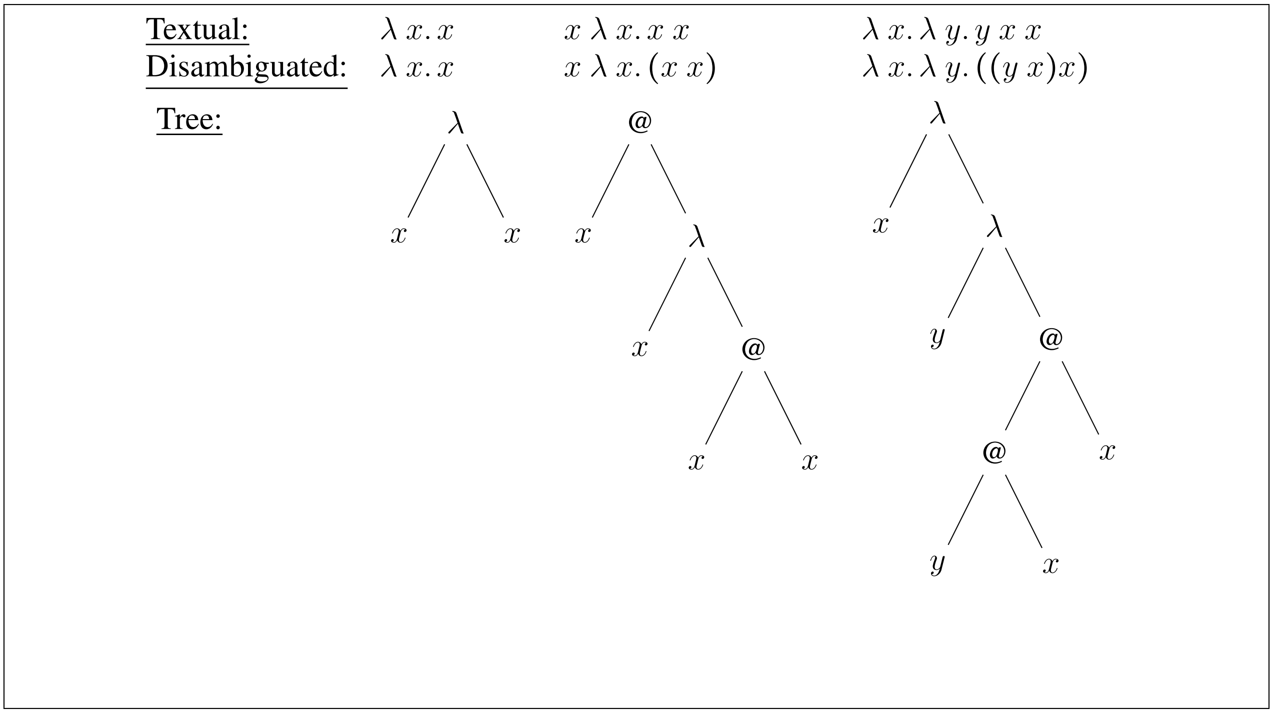

Textual:
$\lambda\, x.\, x$
$x\, \lambda\, x.\, x\, x$
$\lambda\, x.\, \lambda\, y.\, y\, x\, x$
Disambiguated:
$\lambda\, x.\, x$
$x\, \lambda\, x.\, (x\, x)$
$\lambda\, x.\, \lambda\, y.\, ((y\, x) x)$
Tree:
λ
x
x
@
x
λ
x
@
x
x
λ
x
λ
y
@
@
x
y
x

Figure 2.2: Some example terms, in textual form (where parsing conventions must be applied), then disambiguated textual form (no parsing conventions needed), and finally tree form

Figure 2.2 shows several example terms in both textual form and tree form. Exercises below (Section **??**) require you to translate between these forms for some other examples.

Definition 2.1.1 (subterm)**.** *A subterm of a term t is just some subtree (possibly t itself) of t.*

2.2 Binding, bound, and free variable occurrences

As will become more apparent shortly, the λ symbol introduces its variable x *locally* in the body. This means that this x used in the body of a λ-abstraction is semantically different from the same variable x used outside the body. In Computer Science terms, λ introduces x as a local variable, whose scope (i.e., the part of the expression where this variable introduction is in force) is the body of the λ-abstraction.

We have defined terms as certain finite labeled trees. An occurrence of a variable x in such a term is a node of the tree that is labeled with x. The following basic terminology is then used for occurrences.

Definition 2.2.1 (Binding occurrence)**.** *An occurrence of x in t is called binding iff it is the left child of a node labeled with λ.*

Definition 2.2.2 (Bound occurrence)**.** *An occurrence of x in t is called bound iff it occurs somewhere in the right subtree of a node N labeled with λ, where the left child of N is labeled x.*

Definition 2.2.3 (Free occurrence)**.** *An occurrence of x in t that is neither binding nor bound is called free.*

Figure 2.3 gives examples of this terminology, for the term $(\lambda\, x.\, x\; (x\; y))\; (x\; y)$.

Definition 2.2.4 (Free variable)**.** *A variable x is free in t iff it has a free occurrence in t. The set of free variables of t is denoted FV(t). .*

The free variables of the term shown in Figure 2.3 are x and y, because each of these has a free occurrence (boxed node) in the tree.

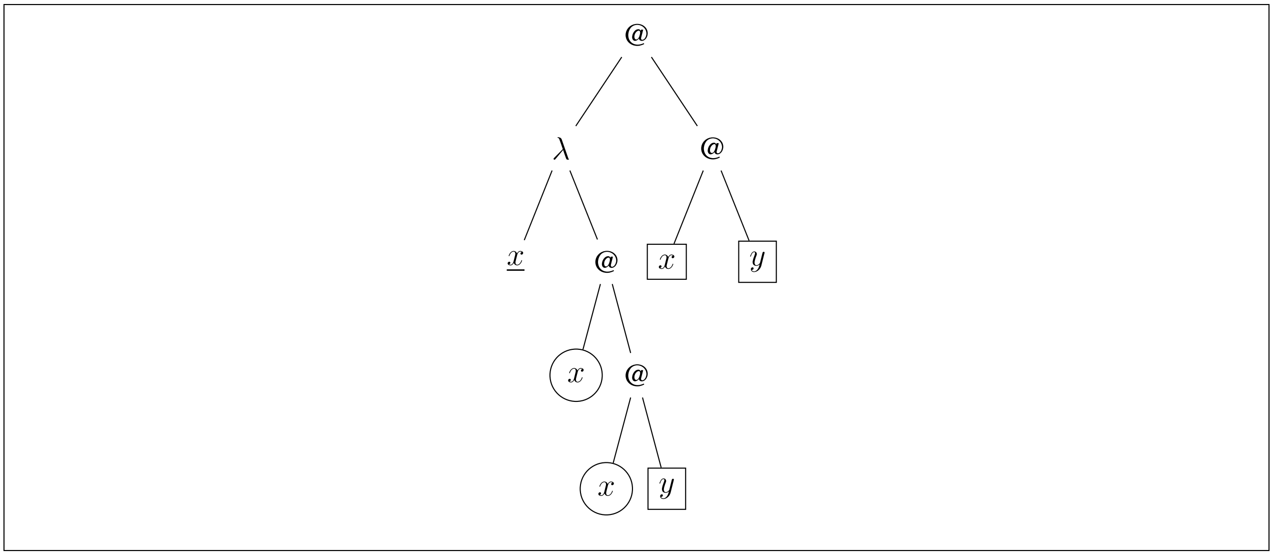


Figure 2.3: Example term illustrating the concepts of binding, bound, and free variable occurrence. The underlined node is a binding occurrence, circled nodes are bound occurrences (bound by that sole underlined binding occurrence), and boxed nodes are free occurrences.

Definition 2.2.5 (Open and closed terms)**.** *A term with at least one free variable occurrence is called* open*, and a term with no free variable occurrences is called* closed*.*

The term in Figure 2.3 is open, because it has a non-empty set of free variables. In contrast, the term $\lambda\, x.\, x\; x$ is closed, because its sole variable x occurs only binding or bound.

2.3 Positions and subterms

It is sometimes useful to be able to refer precisely to subterms of terms, including variable occurrences, using *positions*. A position π is a finite sequence of 0s and 1s. Let us write ϵ for the empty sequence, and $i\pi$ for the sequence that begins with $i \in \{0, 1\}$ and then continues with sequence π. Let us elide ϵ in nonempty sequences.

Definition 2.3.1 (Subterm at a position)**.** *The subterm $t|_\pi$ of term t at position π is defined recursively by:*

$$\begin{array}{rcl} t|_\epsilon & = & t \\ (t_0\; t_1)|_{i\pi} & = & t_i|_\pi \\ (\lambda\, x.\, t)|_0 & = & x \\ (\lambda\, x.\, t)|_{1\pi} & = & t|_\pi \end{array}$$

For example, consider again the term $(\lambda\, x.\, x\; (x\; y))\; (x\; y)$ shown in Figure 2.3. We have:

- $((\lambda\, x.\, x\; (x\; y))\; (x\; y))|_{21} = (x\; y)|_1 = x|_\epsilon = x$
- $((\lambda\, x.\, x\; (x\; y))\; (x\; y))|_{11} = (\lambda\, x.\, x\; (x\; y))|_1 = x$
- $((\lambda\, x.\, x\; (x\; y))\; (x\; y))|_{122} = (\lambda\, x.\, x\; (x\; y))|_{22} = (x\; (x\; y))|_2 = (x\; y)|_\epsilon = x\; y$

The subterm of t at a position can be undefined, if the position extends beyond the leaves of tree t. For example, $((\lambda\, x.\, x\; (x\; y))\; (x\; y))|_{111}$ is undefined, because the position 111 points past the binding occurrence of variable x.

2.4 Capture-avoiding substitution

To define how λ-terms compute (Section 2.5 below), we need a notion of substitution, where one term t' is substituted for the free occurrences of a variable x in another term t. We will use the notation $[t'/x]t$ to denote the result of this substitution, if defined. Substitution is used to define how lambda abstractions reduce (i.e., compute) when applied to arguments. We will need to substitute arguments t' for input variables x in bodies t of λ-abstractions. Note that the notation $[t'/x]t$ is part of our meta-language discussion of λ-calculus, and not new object-language syntax within the language of λ-calculus. Also, it should be mentioned that one finds numerous other notations for substitution in the literature. For example, Church writes $\mathsf{S}^{x}_{t'}t$ where we are instead writing $[t'/x]t$.

Defining substitution is, arguably, the central technical issue in the definition of the lambda calculus. The problem is essentially one of the proper maintenance of scoping of variable, as we will consider next.

2.4.1 Variable capture

The main problem in defining substitution is to ensure that a substitution $[t'/x]t$ avoids *variable capture*, where some free occurrences of a variable y in the term t' get captured by a λ-abstraction of y occurring in t. Such a capture would represent a change of scoping of those occurrences in t': before the subsitution, they were not bound by any λ-abstraction in t, but after the substitution they are. This change of scoping is to be prevented.

The simplest example of the problem is $[y/x]\lambda\, y.\, x$. A naive (and scope-incorrect) approach to substitution would produce $\lambda\, y.\, y$. But then the occurrence of y that is being substituted has changed its scoping. Before the substitution, it is not bound by the displayed $\lambda\, y$, but after the substitution, it is. So it has been captured.

It is common practice, in many research works, to deal with the problem of variable capture by assuming that variables are implicitly renamed to avoid capture. So for this example, the common practice would be to say that the result of $[y/x]\lambda\, y.\, x$ is $\lambda\, w.\, y$, for some variable w different from x and y. This is the approach taken in Hindley and Seldin's book, where it is even specified which variable w (from the countably infinite supply of variables) is to be used [8]. So substitution, in Hindley and Seldin, is indeed a function; not all authors are so careful. Additionally, most works assume what is known as Barendregt's variable convention: in discussing some finite set of lambda terms, we assume that no variable occurs both free in one of the terms and bound in one of the terms [1, Definition 2.1.13], and further assume that variables are implicitly renamed to ensure this.

In this book, I will follow a different approach, adopting a modified version of Church's original proposal for dealing with renaming of variables [6]. During reduction, substitution is not allowed in case it would lead to capture, and variables must be explicitly renamed first in additional reduction steps. I have several reasons for pursuing this approach. First, as variable binding is one of the central technical issues of lambda calculus, it is better, certainly when first learning the theory, not too try to ignore the problem by assuming things are arranged so that it never arises. Second, variable binding turns out to be one of the most tricky aspects both for implementation of languages incorporating lambda calculus, and for formalizing the meta-theory of such language in computer theorem-proving systems. So again, dealing with the problem head-on seems best, as it may encourage development of the theory in a way that minimizes, rather than ignores, the issue. Perhaps we will find better ways to formalize lambda calculus if we isolate the places where renaming is needed, for example. And finally, some research is directly concerned with issues of renaming in lambda calculus, and thus needs to be completely explicit about the issue. An example is works seeking to analyze when renaming can always be avoided [7].

2.4.2 Substitution as a partial function

Figure 2.4 gives the definition of substitution as a partial function. Recall the meta-variable convention that x and y are assumed to refer to different object-language variables. In Equations 3 and 4 it is intended (by the "otherwise" at the end of Equation 2) that $x \in FV(t_1\ t_2)$ and $x \in FV(\lambda\, y.\, t)$, respectively. This ensures that at most one equation can be instantiated to obtain a fact about substitution $[t_1/x]t_2$, for any particular t_1, x, and t_2. For Equations 3 and 4, let us understand that if the right-hand side of the equation is undefined (in some instance of the equation), then the left-hand side is, too.

$$\begin{array}{lllll}
1. & [t'/x]x & = & t' \\
2. & [t'/x]t & = & t, \text{if } x \notin FV(t); \text{otherwise:} \\
3. & [t'/x](t_1\ t_2) & = & ([t'/x]t_1)\ [t'/x]t_2 \\
4. & [t'/x]\lambda\, y.\, t & = & \lambda\, y.\, [t'/x]t, \text{if } y \notin FV(t')
\end{array}$$

Figure 2.4: Recursive definition of capture-avoiding substitution as a partial function. If a recursive call is undefined then the outer call is also considered undefined. The case that is similar to the last clause of the definition but where $y \in FV(t')$ is the basic undefined case. The clauses (equations) of the definition are numbered for reference later.

2.4.3 Examples

Here are some examples of capture-avoiding substitution.

1. $[\lambda\, x.\, x/y]\lambda\, z.\, y\ z = \lambda\, z.\, (\lambda\, x.\, x)\ z$. In detail, labeling the equality symbol with the number of the clause from Figure 2.4, and underlining the part of the term that is being changed (just for clarity), the calculation is:

$$\begin{array}{ll}
\underline{[\lambda\, x.\, x/y]\lambda\, z.\, y\ z} & =_4 \\
\lambda\, z.\, \underline{[\lambda\, x.\, x/y](y\ z)} & =_3 \\
\lambda\, z.\, \underline{([\lambda\, x.\, x/y]y)}\ [\lambda\, x.\, x/y]z & =_1 \\
\lambda\, z.\, (\lambda\, x.\, x)\ \underline{[\lambda\, x.\, x/y]z} & =_2 \\
\lambda\, z.\, (\lambda\, x.\, x)\ z &
\end{array}$$

2. $[(x\ x)/y]\lambda\, y.\, x\ y = \lambda\, y.\, x\ y$. This is just by clause 2 of Figure 2.4, because we are substituting for variable y in a λ-abstraction which binds y. The y for which we are substituting cannot possibly occur free in a λ-abstraction of y, so substitution stops, returning the term (into which we are trying to substitute) unchanged.

3. $[\lambda\, y.\, x\ y/z](z\ \lambda\, x.\, x) = (\lambda\, y.\, x\ y)\ \lambda\, x.\, x$. In detail, we have

$$\begin{array}{ll}
\underline{[\lambda\, y.\, x\ y/z](z\ \lambda\, x.\, x)} & =_3 \\
\underline{([\lambda\, y.\, x\ y/z]z)}\ [\lambda\, y.\, x\ y/z]\lambda\, x.\, x & =_1 \\
(\lambda\, y.\, x\ y)\ \underline{[\lambda\, y.\, x\ y/z]\lambda\, x.\, x} & =_2 \\
(\lambda\, y.\, x\ y)\ \lambda\, x.\, x &
\end{array}$$

4. The substitution $[x\ x/y]\lambda\, x.\, y\ y$ is undefined, because to push the substitution inside a λ-abstraction, clause 4 of Figure 2.4 requires that the λ-bound variable (in this case x) is not free in the term we are substituting (which here is $x\ x$). Since x is free in $x\ x$, this means we cannot apply any of the equations of Figure 2.4 and the substitution is undefined.

2.4.4 Some properties of capture-avoiding substitution

The following lemmas are used below. The proofs are a bit technical, carefully applying the definition of substitution from Figure 2.4. Dealing with the possibility that substitutions are undefined is a somewhat tedious necessity. Each lemma is presented with an example, before its proof.

Lemma 2.4.1. *If t' is closed, then $[t'/x]t$ is defined.*

Example. $[\lambda\, x.\, x/x]\lambda\, y.\, x\ y$ is defined and equals $\lambda\, y.\, (\lambda\, x.\, x)\ y$.

Proof. As t' is closed, the condition in Equation 4 of Figure 2.4 can never be violated, and hence the substitution is defined. □

Lemma 2.4.2. *Suppose that $x \notin FV(t_1)$, and $[t_1/x]t_2$ is defined. Then $x \notin FV([t_1/x]t_2)$.*

Intuitive idea. Substitution of a term t_1 for a variable x in term t_2 results in a term with no free occurrences of x (since they have all been replaced by substitution), as long as x is not free in the substituted term t_1.

Example. Take t_1 to be $\lambda\ y.\, y$, and t_2 to be $x\ \lambda\ z.\, z$. The conditions of the lemma are satisfied, and the result of applying the substitution is $(\lambda\ y.\, y)\ \lambda\ z.\, z$. As stated in the lemma, x does not occur free in this term.

Example where the first condition does not hold. If we take t_1 to be $\lambda\ y.\, x$, and t_2 to be $x\ \lambda\ z.\, z$, then the result of applying the substitution is $(\lambda\ y.\, x)\ \lambda\ z.\, z$, which does contain x free. Hence, the first condition is needed.

Proof. The proof is by induction on t_2. If $x \notin FV(t_2)$, then by Equation 2 of Figure 2.4, $[t_1/x]t_2 = t_2$, and hence $x \notin FV([t_1/x]t_2)$ (since $[t_1/x]t_2 = t_2$ and we are assuming $x \notin FV(t_2)$). So suppose $x \in FV(t_2)$. If t_2 is x, then $[t_1/x]t_2 = t_1$, and the result follows by the assumption that $x \notin FV(t_1)$. If t_2 is $t_a\ t_b$ for some terms t_a and t_b, then by the induction hypothesis, $x \notin FV([t_1/x]t_a)$ and $x \notin FV([t_1/x]t_b)$. So $x \notin FV([t_1/x](t_a\ t_b))$, since $[t_1/x](t_a\ t_b) = ([t_1/x]t_a)\ [t_1/x]t_b$ by Equation 3 of Figure 2.4. If t_2 is $\lambda\ z.\, t$ for some t and some z different from x, then by the induction hypothesis, $x \notin FV([t_1/x]t)$. So $x \notin FV(\lambda\ z.\, [t_1/x]t)$. The latter expression equals $[t_1/x]\lambda\ z.\, t_2$, since the substitution is assumed to be defined. □

Lemma 2.4.3. *Suppose that $y \notin FV(t)$, and $[y/x]t$ is defined. Then $[x/y][y/x]t$ is defined and equals t.*

Intuitive idea. Substituting variable y for variable x and then reversing that (to substitute x for y) leaves the term unchanged, as long as it is legal to replace x with y (avoiding capture), and as long as the original term does not have any free y (since then substituting x for y would replace those free occurrences of y with free occurrences of x, and the term would be different in the end).

Example. If we substitute y for x in $x\ \lambda\ x.\, x$, we get $y\ \lambda\ x.\, x$. Then substituting x for y restores the original term $x\ \lambda\ x.\, x$.

Proof. The proof is by induction on t. If t is x, then $[x/y][y/x]x = x$. If $x \notin FV(t)$, then $[x/y][y/x]t = [x/y]t$. Furthermore, since $y \notin FV(t)$ by assumption, we have $[x/y]t = t$. For the rest of the proof, then, suppose $x \in FV(t)$.

If t is $t_1\ t_2$ for some t_1 and t_2, then by the semantics of Figure 2.4, the expressions $[x/y][y/x](t_1\ t_2)$ and $([x/y][y/x]t_1)\ [x/y][y/x]t_2$ are either (a) both undefined or else (b) both defined and equal. Since $[y/x](t_1\ t_2)$ is defined, so are $[y/x]t_1$ and $[y/x]t_2$. By the induction hypothesis, then, $[x/y][y/x]t_1$ and $[x/y][y/x]t_2$ are defined and equal t_1 and t_2, respectively. So $[x/y][y/x]t_1\ [x/y][y/x]t_2$ is defined and equals $t_1\ t_2$. This means that it must have been option (b). So $[x/y][y/x](t_1\ t_2)$ must be defined and equal to that same value, namely $t_1\ t_2$.

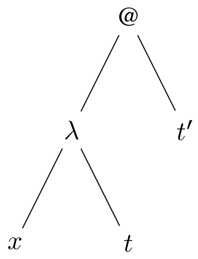
Now suppose t is a λ-abstraction of some variable different from x (as the case where it is an abstraction of x is covered above, by the reasoning when $x \notin FV(t)$). It is also not possible for t to be $\lambda\ y.\, t'$ for any t', because if the bound variable is y, the substitution $[y/x]\lambda\ y.\, t'$ is undefined, So the only case we must consider is where t is $\lambda\ z.\, t'$ for some z (different from x and y) and t'.

We wish to apply the induction hypothesis to conclude that $[x/y][y/x]t'$ is defined and equals t'. For this, we need to know first that $y \notin FV(t')$. But this follows from the facts that $y \notin FV(\lambda\ z.\, t')$ and $y \neq z$. Second, we need to know that $[y/x]t'$ is defined. But this follows since $[y/x]\lambda\ z.\, t'$ is defined and equals $\lambda\ z.\, [y/x]t'$. Since that expression is defined, so also must its subexpression $[y/x]t'$ be defined. So we can indeed apply the induction hypothesis to conclude that $[x/y][y/x]t'$ is defined and equals t'.

Now again by the semantics of Figure 2.4, since z is different from x and y, either $[x/y][y/x]\lambda\ z.\, t'$ and $\lambda\ z.\, [x/y][y/x]t'$ are both undefined, or else both defined and equal. But the reasoning of the previous paragraph shows that $\lambda\ z.\, [x/y][y/x]t'$ is defined (since we concluded that $[x/y][y/x]t'$ is defined) and equals $\lambda\ z.\, t'$ (since we concluded $[x/y][y/x]t' = t'$ by induction hypothesis). Hence, $[x/y][y/x]\lambda\ z.\, t'$ is also defined, and equals $\lambda\ z.\, t'$, as required. □

2.5 Single-step beta-reduction

The central computational concept in λ-calculus is β*-reduction*, which explains how to evaluate function calls. A function call is an application of a λ-abstraction to an argument. So as a tree, it looks like:



In textual form, it is $(\lambda\ x.\, t)\ t'$. The β-axiom says that such a term reduces to $[t'/x]t$; i.e., the result of substituting t' for x in t, as discussed in Section 2.4 above. This is only allowed, however, if the substitution is defined. Terms of the form $(\lambda\ x.\, t)\ t'$ are called β-redexes (for "β-reducible expressions"). Operationally, this reduction of the β-redex to the result of a substitution is called *contracting* the redex; the result of substitution is called the *contractum* . Note that the requirement that the substitution is defined is a particularity of the approach we take in this book.

To give a formal definition of β-reduction, we will first define what it means to reduce a single β-redex, and then, in Section 2.8 below, define β-reduction with multiple steps. In both cases, we define relations between terms. For single-step β-reduction, the relation is denoted $\rightsquigarrow_\beta$.

You may recall that set theoretically, a relation is just a set of ordered pairs, and we may equivalently write $(t, t') \in \rightsquigarrow_\beta$ or $t \rightsquigarrow_\beta t'$ to indicate that t β-reduces to t'. There are several ways to give the definition. First, we can define the β-reduction relation using a set of inference rules. Such rules are of the form

$$\frac{\textit{premise}_1 \ \cdots \ \textit{premise}_n}{\textit{conclusion}}$$

It is allowed for n to be 0, in which case there are no premises and the rule is called an *axiom*. Inference rules are to be understood as universally quantified implications: the conjunction of the premises implies the conclusion, for all instantiations of the meta-variables used. A *derivation* is a kind of tree built by instantiating the inference rules in various ways, and then using the conclusion of one inference as the premise of another. Such instantiated inference rules are called *inferences*. A derivation is *open* if there are some premises that are not the conclusion of any inference. Otherwise, it is called *closed*. We further stipulate that we are only interested in facts that can be proved using finite derivations.

A definition of β-reduction using inference rules is given in Figure 2.5. The leftmost rule in the figure is the β rule, where we require that $[t'/x]t$ is defined in order to use the rule for inferences in a derivation. The other three rules express the idea that a reduction can take place anywhere in a term. Figure 2.6 gives some example derivations. For some other examples:

- The term $(\lambda\ x.\, x\ x)\ y$ single-step β-reduces to $y\ y$, using just the β axiom. This is because substituting y for x in the body $x\ x$ of the λ-abstraction results in $y\ y$.
- On the other hand, the term $(\lambda\ x.\, \lambda\ y.\, x)\ y$ does not β-reduce, because the substitution $[x/y]\lambda\ y.\, x$, according to our definition of substitution, is undefined: replacing x with y in $\lambda\ y.\, x$ would result in variable capture.

Definition 2.5.1 (Single-step β-reduction (rules))**.** *The relation $\rightsquigarrow_\beta$ is the set consisting of exactly those pairs (t, t') where t $\rightsquigarrow_\beta t'$ is derivable (via a finite derivation) using the rules of Figure 2.5. We write applications of the relation in infix notation (as in those rules). If t $\rightsquigarrow_\beta t'$ we say that t β-reduces to t$'$.*

Definition 2.5.2 (β-redex)**.** *Any term of the form $(\lambda\ x.\, t)\ t'$ is called a β-redex. If $[t'/x]$t is defined, let us call it a live β-redex. Otherwise, let us call it a* stuck *β-redex.*

Definition 2.5.3 (nested redex)**.** *Suppose that a redex R_1 is a subterm of some other redex R_2. Then we say that R_1 is a nested redex (nested within R_2).*

Definition 2.5.4 (β-expansion)**.** *The inverse of the $\rightsquigarrow_\beta$ relation is called β-expansion. If t $\rightsquigarrow_\beta t'$, we say that t$'$ β-expands to t.*

The following definition is generic, for any $R \subseteq (A \times A)$. We will call such a set a (binary) relation on A.

$$\frac{}{(\lambda\, x.\, t)\ t' \ \rightsquigarrow_\beta\ [t'/x]t} \qquad \frac{t\ \rightsquigarrow_\beta\ t'}{\lambda\, x.\, t\ \rightsquigarrow_\beta\ \lambda\, x.\, t'} \qquad \frac{t_1\ \rightsquigarrow_\beta\ t_1'}{t_1\ t_2\ \rightsquigarrow_\beta\ t_1'\ t_2} \qquad \frac{t_2\ \rightsquigarrow_\beta\ t_2'}{t_1\ t_2\ \rightsquigarrow_\beta\ t_1\ t_2'}$$

Figure 2.5: Inference rules defining the β-reduction relation. It is required that the substitution in the leftmost rule be defined, in order to use the rule.

$$\frac{}{(\lambda\, x.\, x\ x)\ \lambda\, y.\, y \rightsquigarrow_\beta (\lambda\, y.\, y)\ \lambda\, y.\, y} \qquad \frac{\frac{}{(\lambda\, y.\, y)\ x \rightsquigarrow_\beta x}}{\lambda\, x.\, (\lambda\, y.\, y)\ x \rightsquigarrow_\beta \lambda\, x.\, x} \qquad \frac{\frac{\frac{}{(\lambda\, x.\, x)\ z \rightsquigarrow_\beta z}}{y\ ((\lambda\, x.\, x)\ z) \rightsquigarrow_\beta y\ z}}{x\ (y\ ((\lambda\, x.\, x)\ z)) \rightsquigarrow_\beta x\ (y\ z)}$$

Figure 2.6: Example derivations using the rules of Figure 2.5 for β-reduction.

Definition 2.5.5 (determinism)**.** *An element x is said to be deterministic with respect to some relation R on a set A iff for all y and y', if xRy and xRy' then $y = y'$. R itself is called deterministic iff every element of A is deterministic with respect to R. Mathematically, being deterministic is the same as being a functional relation. A nondeterministic relation is then simply one which fails to be deterministic for at least one x.*

Lemma 2.5.6. *The $\rightsquigarrow_\beta$ relation is nondeterministic.*

Proof. Any term containing two non-nested β-redexes will have two distinct contracta. For example, $(\lambda\, x.\, x\ y)\ (\lambda\, x.\, x\ z)$ has distinct contracta $y\ (\lambda\, x.\, x\ z)$ and $(\lambda\, x.\, x\ y)\ z$. Hence, $\rightsquigarrow_\beta$ is nondeterministic. □

2.5.1 An alternative definition using contexts

Another way to define the same $\rightsquigarrow_\beta$ relation is with contexts. Let us first introduce the concept of *grafting*, which is just substitution that does not avoid capture. We will write $\langle t/x\rangle t'$ for the grafting relation (there does not seem to be a generally adopted notation for grafting). The definition, given in Figure 2.7, is essentially the same as the one for substitution, except that the last clause does not impose any requirements on *FV*(t').

Definition 2.5.7 (context)**.** *A term t is called a context iff it contains exactly one free occurrence of a special fixed variable q. If t is a context, then we write $\langle t'/q\rangle t$ more briefly as $\langle t'\rangle t$. The variable q is called the hole of the context.*

Using contexts, we may give the following alternative definition of the β-reduction relation:

Definition 2.5.8 (Single-step β-reduction (contexts))**.** *The single-step β-reduction relation $\rightsquigarrow_\beta$ is alternatively defined to be the set of all ordered pairs with first component $\langle(\lambda\, x.\, t)\ t'\rangle t''$ and second component $\langle[t'/x]t\rangle t''$, for some x, t, t', and t'' (with t'' a context and $[t'/x]t$ defined).*

The idea of this definition is to express that (in operational terms) if you find a β-redex somewhere in some possibly bigger term t_1, then you may reduce t_1 by contracting that β-redex and then rebuilding the rest of the term t_1 around the contractum. The definition expresses finding $(\lambda\, x.\, t)\ t'$ in possibly bigger term t_1 by writing $\langle(\lambda\, x.\, t)\ t'\rangle t''$ for t_1. In other words, you have some term t_1 that contains a designated occurrence of the redex, because t_1 is what you get when you graft the redex in for the single occurrence of special variable q in some context t''. The reason that we use

$$\begin{array}{lcl} \langle t'/x\rangle x & = & t' \\ \langle t'/x\rangle t & = & t, \text{if } x \notin \mathit{FV}(t); \text{otherwise:} \\ \langle t'/x\rangle(t_1\ t_2) & = & (\langle t'/x\rangle t_1)\ \langle t'/x\rangle t_2 \\ \langle t'/x\rangle\lambda\, y.\, t & = & \lambda\, y.\, \langle t'/x\rangle t \end{array}$$

Figure 2.7: Recursive definition of grafting, a total function similar to substitution but intentionally allowing variable capture.

grafting for this definition instead of substitution is to allow contraction of redexes that contain free variables that are bound in t_1. We will discuss this point further in the examples next.

Examples

1. $(\lambda\, x.\, x\; x)\; \lambda\, y.\, y$ reduces to $(\lambda\, y.\, y)\; \lambda\, y.\, y$. For this, the meta-variables of Definition 2.5.8 are instantiated thus:

   - x is instantiated with x,
   - t with $x\; x$,
   - t' with $\lambda\, y.\, y$
   - t'' with q (the special context variable)

2. $\lambda\, x.\, (\lambda\, y.\, y)\; x$ reduces to $\lambda\, x.\, x$. Here the instantiations for Definition 2.5.8 are:

   - x is instantiated with y,
   - t with y,
   - t' with x
   - t'' with $\lambda\, x.\, q$.

   Notice that here we really need the idea of grafting, because our redex contains x free, which is bound in t''. We want to allow reduction of redexes that contain variables bound outside the redex, and so we use grafting to specify that the x in the redex may be bound in t''. If we used substitution instead of grafting, this reduction would not be allowed by Definition 2.5.8, because $[(\lambda\, y.\, y)\; x/q]\lambda\, x.\, q$ is undefined (as the x in $(\lambda\, y.\, y)\; x$ would be captured pushing the substitution into $\lambda\, x.\, q$).

3. $(\lambda\, x.\, x\; x)\; \lambda\, x.\, x\; x$ reduces to that very same term. The instantiations for Definition 2.5.8 are:

   - x is instantiated with x,
   - t with $x\; x$,
   - t' with $\lambda\, x.\, x\; x$
   - t'' with q.

   We calculate the substitution $[t'/x]t$ as follows (referencing clause numbers from Figure 2.4):

$$\begin{array}{ll}
\underline{[\lambda\, x.\, x\; x/x](x\; x)} & =_3 \\
(\underline{[\lambda\, x.\, x\; x/x]x})\; [\lambda\, x.\, x\; x/x]x & =_1 \\
(\lambda\, x.\, x\; x)\; \underline{[\lambda\, x.\, x\; x/x]x} & =_1 \\
(\lambda\, x.\, x\; x)\; \lambda\, x.\, x\; x &
\end{array}$$

   This is an important basic example, because it shows that terms can reduce to themselves, and hence can give rise to infinite reductions (to be defined just below).

2.5.2 One more alternative: replacement at a position

Yet one further alternative definition of single-step β-reduction is using the notion of replacement of a subterm at a position.

Definition 2.5.9 (Replacement at a position). *The term $t[t']_\pi$ replacing the subterm of t at position π by t' is defined recursively by:*

$$\begin{array}{rcl}
t[t']_\epsilon & = & t' \\
(t_1\; t_2)[t']_{1\pi} & = & (t_1[t']_\pi)\; t_2 \\
(t_1\; t_2)[t']_{2\pi} & = & t_1\; (t_2[t']_\pi) \\
(\lambda\, x.\, t)[t']_{2\pi} & = & \lambda\, x..(t[t']_\pi)
\end{array}$$

$$\frac{t\ R\ t'}{t\ \mathcal{T}[R]\ t'} \qquad \frac{t\ \mathcal{T}[R]\ t'}{\lambda\, x.\, t\ \mathcal{T}[R]\ \lambda\, x.\, t'} \qquad \frac{t_1\ \mathcal{T}[R]\ t_1'}{t_1\ t_2\ \mathcal{T}[R]\ t_1'\ t_2} \qquad \frac{t_2\ \mathcal{T}[R]\ t_2'}{t_1\ t_2\ \mathcal{T}[R]\ t_1\ t_2'}$$

Figure 2.8: Definition of compatible closure of R.

$$\overline{(\lambda\, x.\, t)\ t'\ \ \beta\ \ [t'/x]t}$$

Figure 2.9: Definition of the bare β relation, from which $\rightsquigarrow_\beta$ may then be defined via compatible closure. This again presupposes the substitution is defined.

For an example:

$$\begin{array}{ll}
(\lambda\, x.\, \lambda\, y.\, x\ y)[y\ \lambda\, z.\, z]_{22} & = \\
\lambda\, x.\, ((\lambda\, y.\, x\ y)[y\ \lambda\, z.\, z]_{2}) & = \\
\lambda\, x.\, \lambda\, y.\, ((x\ y)[y\ \lambda\, z.\, z]_{\epsilon}) & = \\
\lambda\, x.\, \lambda\, y.\, (y\ \lambda\, z.\, z) &
\end{array}$$

Note that replacement of binding occurrences of variables is not allowed by this definition.

Using replacement, we can give yet another definition for single-step β-reduction:

Definition 2.5.10 (Single-step β-reduction (replacement at position))**.** *The single-step β-reduction relation $\rightsquigarrow_\beta$ is alternatively defined to be the set of all ordered pairs with first component $t''[(\lambda\ x.\, t)\ t']_\pi$ and second component $t''[[t'/x]t]_\pi$, for some x, t, t', t'', and π, where π is a legal position of t'' and $[t'/x]t$ defined.*

This definition uses a position to name where a β-redex is to be found and contracted, instead of using the hole of a context to show where the contraction takes place.

2.6 Definitions using closure operators

In what follows, yet another way of defining single-step β-reduction will prove illuminating, which is via **closure operators** on relations. Such an operator takes a relation R and produces some new relation (let us call it R') such that $R \subseteq R'$ and R' satisfies some desired property. We will generally define closure operators using rules.

An important example in our setting is the **compatible closure**, which we will denote $\mathcal{T}[R]$, of R. The rules for this are given in Figure 2.8. Given any relation R on terms, $\mathcal{T}[R]$ is also a relation on terms, which contains R; i.e., if two terms are related by R, then thanks to the first rule of Figure 2.8, they are also related by $\mathcal{T}[R]$. We may call this the inclusion rule for the compatible closure. The property it satisfies is that is closed under the syntactic constructs of lambda calculus, in the sense expressed by the second, third, and fourth rules of Figure 2.8.

We may now see that if we just define the bare β relation as in Figure 2.9, then we can obtain $\rightsquigarrow_\beta$ as the compatible closure of β. We may easily observe that $\rightsquigarrow_\beta$ as defined this way and as defined via the rules of Figure 2.5 are the same. The definition using compatible closure and bare β just decomposes the rules of Figure 2.5 into two parts, but is otherwise essentially the same, except for bare β inferences, which we will not generally write (thus preferring the slightly more compact rules of Figure 2.5 for presenting examples).

Definition 2.6.1 (symmetric closure)**.** *If R is a relation, then its symmetric closure is the union of R with R^{-1}, its inverse (i.e., the relation consisting of those pairs (y, x) where $(x, y) \in R$). When R is denoted in our meta-language using some arrow symbol like $\rightarrow$, the symmetric closure is conveniently denoted by adding an arrowhead, to get something like $\leftrightarrow$.*

A final closure operator commonly used in Computer Science is the reflexive transitive closure R^* of relation R, defined in Figure 2.10. We will use this in the definition of multi-step β-reduction below. The first rule may be called the inclusion rule, the second the reflexivity rule, and the third the transitivity rule. Note that we are not taking multi-step β-reduction to be just $\rightsquigarrow_\beta^*$, as this relation does not allow renaming of local variables, the subject we turn to in Section 2.7.

$$\frac{t\,R\,t'}{t\,R^*\,t'} \qquad \frac{}{t\,R^*\,t} \qquad \frac{t_1\,R^*\,t_2 \quad t_2\,R^*\,t_3}{t_1\,R^*\,t_3}$$

Figure 2.10: Definition of the reflexive, transitive closure R^* of relation R.

2.6.1 Some properties of the closure operators

In some of the proofs later in the book, we will make use of some properties of the closure operators above.

Lemma 2.6.2 (monotonicity of star)**.** *For relations R and S on set A, if $R \subseteq S$, then $R^* \subseteq S^*$.*

Proof. Assume $t\ R^*\ t'$, and show $t\ S^*\ t'$. The proof is by induction on the derivation of $t\ R^*\ t'$. Case :

$$\frac{t\ R\ t'}{t\ R^*\ t'}$$

Since $R \subseteq S$, we have $t\ S\ t'$ from $t\ R\ t'$, and then obtain $t\ S^*\ t'$ by applying this same inclusion rule.

Case :

$$\frac{}{t\ R^*\ t}$$

Note that in this case, the inference used forces t' to equal t. Using this same reflexivity rule, we derive $t\ S^*\ t$.

Case :

$$\frac{t\ R^*\ t'' \quad t''\ R^*\ t'}{t\ R^*\ t'}$$

We may construct this derivation, where uses of the induction hypothesis are indicated with inferences labeled *IH*. These are not inferences by a rule of Figure 2.10, but rather indicate that invocation of the induction hypothesis is legal for the fact above the bar and produces the result shown below the bar.

$$\frac{\dfrac{t\ R^*\ t''}{t\ S^*\ t''}\ \textit{IH} \quad \dfrac{t''\ R^*\ t'}{t''\ S^*\ t'}\ \textit{IH}}{t\ S^*\ t'}$$

□

Lemma 2.6.3 (compatible closure preserves symmetry)**.** *If R is symmetric, then $\mathcal{T}[R]$ is also.*

Proof. Given symmetric R, assume $t\mathcal{T}[R]t'$ and show $t'\mathcal{T}[R]t$. The proof is by induction on the assumed derivation of $t\mathcal{T}[R]t'$ (using the rules of Figure 2.8).

Case :

$$\frac{t\ R\ t'}{t\mathcal{T}[R]t'}$$

We construct this derivation, where $t'\ R\ t$ is deduced by symmetry of R:

$$\frac{\dfrac{t\ R\ t'}{t'\ R\ t}}{t'\mathcal{T}[R]t}$$

Case :

$$\frac{t\ \mathcal{T}[R]\ t'}{\lambda\, x.t\ \mathcal{T}[R]\ \lambda\, x.t'}$$

We construct:

$$\frac{\dfrac{t\ \mathcal{T}[R]\ t'}{t'\ \mathcal{T}[R]\ t}\ \textit{IH}}{\lambda\, x.t'\ \mathcal{T}[R]\ \lambda\, x.t}$$

Case :

$$\frac{t_1\,\mathcal{T}[R]\,t_1'}{t_1\,t_2\,\mathcal{T}[R]\,t_1'\,t_2}$$

We construct:

$$\frac{\dfrac{t_1\,\mathcal{T}[R]\,t_1'}{t_1'\,\mathcal{T}[R]\,t_1}\;\mathit{IH}}{t_1'\,t_2\,\mathcal{T}[R]\,t_1\,t_2}$$

Case :

$$\frac{t_2\,\mathcal{T}[R]\,t_2'}{t_1\,t_2\,\mathcal{T}[R]\,t_1\,t_2'}$$

We construct:

$$\frac{\dfrac{t_2\,\mathcal{T}[R]\,t_2'}{t_2'\,\mathcal{T}[R]\,t_2}\;\mathit{IH}}{t_1\,t_2'\,\mathcal{T}[R]\,t_1\,t_2}$$

□

Lemma 2.6.4 (reflexive-transitive closure preserves symmetry)**.** *If R is symmetric, then R^* is also.*

Proof. Given symmetric R, assume $t\,R^*\,t'$ and show $t'\,R^*\,t$. The proof is by induction on the assumed derivation of $t\,R^*\,t'$ (using the rules of Figure 2.10). Case :

$$\frac{t\,R\,t'}{t\,R^*\,t'}$$

We construct this derivation, where $t'\,R\,t$ is deduced by symmetry of R:

$$\frac{\dfrac{t\,R\,t'}{t'\,R\,t}}{t'\,R^*\,t}$$

Case :

$$\frac{}{t\,R^*\,t}$$

In this case $t = t'$ and we thus have $t'\,R^*\,t$ by this very inference.

Case :

$$\frac{t_1\,R^*\,t_2 \qquad t_2\,R^*\,t_3}{t_1\,R^*\,t_3}$$

We construct:

$$\frac{\dfrac{t_2\,R^*\,t_3}{t_3\,R^*\,t_2}\;\mathit{IH} \qquad \dfrac{t_1\,R^*\,t_2}{t_2\,R^*\,t_1}\;\mathit{IH}}{t_3\,R^*\,t_1}$$

□

Lemma 2.6.5 (reflexive-transitive closure is compatible with λ-abstraction)**.** *Suppose R is a relation on terms. If $t\,\mathcal{T}[R]^*\,t'$, then $\lambda\,x.\,t\,\mathcal{T}[R]^*\,\lambda\,x.\,t'$.*

Proof. We proceed by induction on the assumed derivation, again following the three rules of Figure 2.10.

Case :

$$\frac{t\,\mathcal{T}[R]\,t'}{t\,\mathcal{T}[R]^*\,t'}$$

We construct

$$\dfrac{\dfrac{t\ \mathcal{T}[R]\ t'}{\lambda\, x.\, t\ \mathcal{T}[R]\ \lambda\, x.\, t'}}{\lambda\, x.\, t\ \mathcal{T}[R]^*\ \lambda\, x.\, t'}$$

Case :

$$\dfrac{}{t\ \mathcal{T}[R]^*\ t}$$

This case forces $t' = t$. We construct the following, again applying the reflexivity rule:

$$\dfrac{}{\lambda\, x.\, t\ \mathcal{T}[R]^*\ \lambda\, x.\, t}$$

Case :

$$\dfrac{t\ \mathcal{T}[R]^*\ t'' \quad t''\ \mathcal{T}[R]^*\ t'}{t\ \mathcal{T}[R]^*\ t'}$$

We construct the following, applying the transitivity rule:

$$\dfrac{\dfrac{t\ \mathcal{T}[R]^*\ t''}{\lambda\, x.\, t\ \mathcal{T}[R]^*\ \lambda\, x.\, t''}\ \textit{IH} \quad \dfrac{t''\ \mathcal{T}[R]^*\ t'}{\lambda\, x.\, t''\ \mathcal{T}[R]^*\ \lambda\, x.\, t'}\ \textit{IH}}{\lambda\, x.\, t\ \mathcal{T}[R]^*\ \lambda\, x.\, t'}$$

□

Lemma 2.6.6 (reflexive-transitive closure is compatible with application)**.** *Suppose R is a relation on terms. If $t_1\ \mathcal{T}[R]^*\ t_1'$, then $t_1\ t_2\ \mathcal{T}[R]^*\ t_1'\ t_2$. Also, if $t_2\ \mathcal{T}[R]^*\ t_2'$, then $t_1\ t_2\ \mathcal{T}[R]^*\ t_1\ t_2'$.*

Proof. The proof is quite similar to that of Lemma 2.6.5, so it is omitted. □

Lemma 2.6.7 (reflexive-transitive closure preserves compatibility)**.** *Suppose R is a relation on terms, and consider $\mathcal{T}[R]^*$. Then the relations $\mathcal{T}[R]^*$ and its compatible closure $\mathcal{T}[\mathcal{T}[R]^*]$ are the same.*

Proof. Since $\mathcal{T}[R]^* \subseteq \mathcal{T}[\mathcal{T}[R]^*]$ by the inclusion rule of Figure 2.8, it suffices to show $\mathcal{T}[\mathcal{T}[R]^*] \subseteq \mathcal{T}[R]^*$. So assume $t\ \mathcal{T}[\mathcal{T}[R]^*]\ t'$ and show $t\ \mathcal{T}[R]^*\ t'$. The proof is by induction on the assumed derivation (with the rules of Figure 2.8).

Case :

$$\dfrac{t\ \mathcal{T}[R]^*\ t'}{t\ \mathcal{T}[\mathcal{T}[R]^*]\ t'}$$

The desired conclusion for this inclusion inference is its premise: $t\ \mathcal{T}[R]^*\ t'$.

Case :

$$\dfrac{t\ \mathcal{T}[\mathcal{T}[R]^*]\ t'}{\lambda\, x.\, t\ \mathcal{T}[\mathcal{T}[R]^*]\ \lambda\, x.\, t'}$$

By the induction hypothesis, we have $t\ \mathcal{T}[R]^*\ t'$. The result then follows by Lemma 2.6.5.

Case :

$$\dfrac{t_1\ \mathcal{T}[\mathcal{T}[R]^*]\ t_1'}{t_1\ t_2\ \mathcal{T}[\mathcal{T}[R]^*]\ t_1'\ t_2}$$

By the induction hypothesis, we have $t_1\ \mathcal{T}[R]^*\ t_1'$. The result then follows by Lemma 2.6.6.

Case :

$$\dfrac{t_2\ \mathcal{T}[\mathcal{T}[R]^*]\ t_2'}{t_1\ t_2\ \mathcal{T}[\mathcal{T}[R]^*]\ t_1\ t_2'}$$

Similar to the previous case.

□

$$\frac{}{\lambda\, x.\, t \;\;\alpha\;\; \lambda\, y.\, [y/x]t}\; y \notin FV(t)$$

Figure 2.11: Definition of the bare α relation, from which $\leadsto_\alpha$ is then defined via compatible closure. This presupposes the substitution in the rule is defined.

2.7 Alpha-equivalence

A λ-abstraction introduces a variable with local scope, to refer to input arguments. Terms that are the same except for choice of these local variables intuitively should be equivalent in some way. In this section, we define this notion of equivalence, which historically is called α-equivalence. The intention is that two terms t_1 and t_2 are α-equivalent iff one can perform safe renamings to different λ-subterms of t_1 to obtain t_2. A safe renaming of a subterm $\lambda\, x.\, t$ is $\lambda\, y.\, [y/x]t$ where $y \notin FV(t)$ and the substitution is defined. Safe renamings change binding and their corresponding bound occurrences of x into y, where y is not free in the body t. By requiring y not to be free in t, we ensure that we cannot accidentally capture free occurrences of y in t, which would be an example of the scope confusion we are trying to avoid with capture-avoiding substitution.

To define α-equivalence, we begin with the bare α relation of Figure 2.11. This allows us to rename the variable x bound by $\lambda\, x.\, t$ to any y which is not free in t, and for which the substitution $[y/x]t$ is defined. If y does not have any occurrences whatsoever in t, then it satisfies these two conditions. Those conditions are required to ensure that we do not rename x to some variable which either would capture some free variable of t or which would itself be captured when replacing x with it.

Next we apply the compatible closure (Figure 2.8), to get $\leadsto_\alpha$. So $\leadsto_\alpha$ is defined to be $\mathcal{T}[\alpha]$. This relation allows us to perform such a renaming anywhere we want in a term. Finally, we take the reflexive transitive closure, so that we can perform any finite sequence of renamings. This gives us the final definition:

Definition 2.7.1 (α-equivalence)**.** *The relation $=_\alpha$, called α-equivalence, is $\mathcal{T}[\alpha]^*$.*

2.7.1 Examples

1. $\lambda\, x.\, x \leadsto_\alpha \lambda\, y.\, y$, for any y different from x. In general, we can see that $\leadsto_\alpha$ is nondeterministic.

2. $\lambda\, x.\, \lambda\, y.\, z\; x$ is α-equivalent to $\lambda\, y.\, \lambda\, w.\, z\; y$, by combining (using the rules of Figure 2.10) the following $\leadsto_\alpha$ steps:

   - $\lambda\, x.\, \lambda\, y.\, z\; x \leadsto_\alpha \lambda\, x.\, \lambda\, w.\, z\; x$, proved with this derivation (using the rules of Figure 2.8 and Figure 2.11):

   $$\cfrac{\cfrac{\overline{\lambda\, y.\, z\; x\; \alpha\; \lambda\, w.\, [w/y](z\; x)}}{\lambda\, y.\, z\; x \mathcal{T}[\alpha] \lambda\, w.\, [w/y](z\; x)}}{\lambda\, x.\, \lambda\, y.\, z\; x \mathcal{T}[\alpha] \lambda\, x.\, \lambda\, w.\, z\; x}$$

   - $\lambda\, x.\, \lambda\, w.\, z\; x \leadsto_\alpha \lambda\, y.\, \lambda\, w.\, z\; y.$

   We combine those derivations into a single derivation for the α-equivalence in Figure 2.12.

3. $\lambda\, x.\, \lambda\, y.\, y\; x =_\alpha \lambda\, y.\, \lambda\, x.\, x\; y$, but this is slightly tricky. It is similar to the problem of swapping the values of two variables in an imperative programming language, and uses the same solution: introduce an auxiliary variable. So we have these $\leadsto_\alpha$ steps:

   - $\lambda\, x.\, \underline{\lambda\, y.\, y\; x} \leadsto_\alpha \lambda\, x.\, \lambda\, w.\, w\; x$
   - $\underline{\lambda\, x.\, \lambda\, w.\, w\; x} \leadsto_\alpha \lambda\, y.\, \lambda\, w.\, w\; y$
   - $\lambda\, y.\, \underline{\lambda\, w.\, w\; y} \leadsto_\alpha \lambda\, x.\, \lambda\, y.\, y\; x$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\lambda\, y.\, z\, x\ \alpha\ \lambda\, w.\, [w/y]z\, x}}{\lambda\, y.\, z\, x \leadsto_\alpha \lambda\, w.\, [w/y]z\, x}}{\lambda\, x.\, \lambda\, y.\, z\, x \leadsto_\alpha \lambda\, x.\, \lambda\, w.\, z\, x}}{\lambda\, x.\, \lambda\, y.\, z\, x \leadsto_\alpha^* \lambda\, x.\, \lambda\, w.\, z\, x} \quad \dfrac{\dfrac{\dfrac{}{\lambda\, x.\, \lambda\, w.\, z\, x\ \alpha\ \lambda\, y.\, \lambda\, w.\, [y/x](z\, x)}}{\lambda\, x.\, \lambda\, w.\, z\, x \leadsto_\alpha \lambda\, y.\, \lambda\, w.\, [y/x](z\, x)}}{\lambda\, x.\, \lambda\, w.\, z\, x \leadsto_\alpha^* \lambda\, y.\, \lambda\, w.\, z\, y}}{\lambda\, x.\, \lambda\, y.\, z\, x \leadsto_\alpha^* \lambda\, y.\, \lambda\, w.\, z\, y}$$

Figure 2.12: Example derivation of an α-equivalence, using the rules of Figures 2.10, Figure 2.8, and 2.11. Naturally, the passage from an α step to a $\leadsto_\alpha$ step at the top parts of the derivation is rather redundant, and we may safely omit the bare α inferences in other examples.

2.7.2 Properties of α-equivalence

Lemma 2.7.2. *α is symmetric.*

Proof. Suppose we have $t_1\ \alpha\ t_2$. The only way this can happen, with the sole rule for the bare α relation (Figure 2.11) is if t_1 is of the form $\lambda\, x.\, t$ for some variable x and term t, and t_2 is of the form $\lambda\, y.\, [y/x]t$ with $y \notin FV(t)$ and $[y/x]t$ defined. We must show that under these conditions, $t_2\ \alpha\ t_1$. This can be proved using the rule for bare α by instantiating the meta-variables in that rule as follows:

- x is instantiated with y
- y is intantiated with x
- t is instantiated with $[y/x]t$

With those instantiations, we obtain this fact from the rule, if the several conditions required by the rule hold (which we will check next):

$$\lambda\, y.\, [y/x]t\ \ \alpha\ \lambda\, x.\, [x/y][y/x]t$$

By Lemma 2.4.3, $\lambda\, x.\, [x/y][y/x]t$ is defined and equal to $\lambda\, x.\, t$, so we indeed have $t_2\ \alpha\ t_1$. That lemma requires that $y \notin FV(t)$ and that $[y/x]t$ is defined; both of these hold by assumption from the original application of the rule for bare α. The requirement, on this new bare-α inference, that $x \notin FV([y/x]t)$ follows by Lemma 2.4.2, since $x \notin FV(y) = \{y\}$ (because $x \neq y$ by our convention on meta-variables for variables).

□

Corollary 2.7.3. *$=_\alpha$ is symmetric, so $=_\alpha$ is indeed an equivalence relation.*

Proof. Since bare α is symmetric (Lemma 2.7.2), we may apply Lemma 2.6.3 to conclude that $\leadsto_\alpha$ is symmetric. Then we may apply Lemma 2.6.4 to conclude that $(\leadsto_\alpha)^*$ (which we defined α-equivalence to be in Definition 2.7.1) is symmetric. □

2.8 Multi-step beta-reduction

Having defined single-step β-reduction (Definition 2.5.1, or alternatively Definition 2.5.8) and α-equivalence (Definition 2.7.1), we will combine these two concepts for a relation expressing the idea of computation: that is, performing a sequence of β-reductions, where safe renaming of variables is allowed between steps to enable computation to proceed (where it could otherwise be stuck due to undefinedness of a substitution). This can be done concisely using the closure operators introduced above. First, though, we should recall the definition of relational composition:

Definition 2.8.1 (relational composition)**.** *If R and S are (binary) relations, then by RS we denote their composition, namely,*

$$\{(x, z) \mid \exists\, y.\ (x, y) \in R\ \wedge\ (y, z) \in S\}$$

That is, the set of pairs (x, z) such that there exists some y with $(x, y) \in R$ and $(y, z) \in S$.

Definition 2.8.2 (single step β-reduction with renaming)**.** *For compact notation below, let us write $\leadsto$ for $=_\alpha\leadsto_\beta$ (i.e., the relational composition of α-equivalence, followed by single-step β-reduction). This is called single-step β-reduction with renaming.*

A single β-reduction step with renaming allows one to perform some (possibly zero) renamings, then take a $\leadsto_\beta$-step, and then perform another set of renamings. We are generally interested in taking multiple steps of $\leadsto$, using $\leadsto^*$, which we will call multi-step β-reduction with renaming. It is often useful to identify a sequence of terms underlying a multi-step β-reduction with renaming, as follows:

Definition 2.8.3 ($\leadsto$-reduction sequence)**.** *A $\leadsto$-reduction sequence is a finite list of terms $t_1, \ldots, t_k$, such that for each $i \in \{1, \ldots, k-1\}$, $t_i \leadsto t_{i+1}$. We may write such a sequence as*

$$t_1 \leadsto \ldots \leadsto t_k$$

We say this is a $\leadsto$-reduction sequence for $t_1 \leadsto^* t_k$.

Note that there may be more than one $\leadsto$-reduction sequence for a given multi-step reduction, because different intermediate renamings may be possible.

2.8.1 Examples

1. The following shows that $(\lambda\, x.\, \lambda\, y.\, x\, x)\; \lambda\, x.\, y$ multi-step β-reduces to $\lambda\, w.\, y$. For the $=_\alpha$ step, I am underlining the underlying α-step that has been taken, and similarly for the $\leadsto_\beta$ steps, I am underlining the underlying β-step that has been taken (with α and β of Figures 2.11 and 2.9).

$$\begin{array}{ll} (\lambda\, x.\, \underline{\lambda\, y.\, x\, x})\; \lambda\, x.\, y & =_\alpha \\ \underline{(\lambda\, x.\, \lambda\, w.\, x\, x)\; \lambda\, x.\, y} & \leadsto_\beta \\ \lambda\, w.\, \underline{(\lambda\, x.\, y)\; \lambda\, x.\, y} & \leadsto_\beta \\ \lambda\, w.\, y & \end{array}$$

A $\leadsto$-reduction sequence (as in Definition 2.8.3) for this is

$$\begin{array}{ll} (\lambda\, x.\, \underline{\lambda\, y.\, x\, x})\; \lambda\, x.\, y & \leadsto \\ \lambda\, w.\, \underline{(\lambda\, x.\, y)\; \lambda\, x.\, y} & \leadsto \\ \lambda\, w.\, y & \end{array}$$

The following reduction sequence is different, because in the second line the first λ-bound variable is p instead of w:

$$\begin{array}{ll} (\lambda\, x.\, \underline{\lambda\, y.\, x\, x})\; \lambda\, x.\, y & \leadsto \\ \lambda\, p.\, \underline{(\lambda\, x.\, y)\; \lambda\, x.\, y} & \leadsto \\ \lambda\, w.\, y & \end{array}$$

Note that we may reasonably generalize this notion of $\leadsto$-reduction sequence to other relations besides $\leadsto$. We can call these *relational sequences*. For example, a $=_\alpha$-sequence is a list of terms where consecutive terms are related by $=_\alpha$. Or a bare β sequence would be a list of terms where consecutive terms are related by the bare β relation of Figure 2.9.

Definition 2.8.4 (maximal relational sequence)**.** *An R-sequence $t_1\; R \cdots R\; t_n$ is called maximal iff there is no t' with $t_n\; R\; t'$.*

A maximal β-reduction sequence is one that ends in a term which cannot be single-step β-reduced (with renaming). Such a term is called a β-normal form:

Definition 2.8.5 (β-normal form)**.** *A term t is in β-normal form iff there is no t' such that $t \leadsto t'$. This is sometimes denoted $t \not\leadsto$. One sometimes writes $t \leadsto^! t'$ to mean that $t \leadsto^* t'$ where t' is β-normal.*

$$\frac{}{S\,t_1\,t_2\,t_3 \rightsquigarrow_\beta t_1\,t_3\,(t_2\,t_3)} \qquad \frac{}{K\,t_1\,t_2 \rightsquigarrow_\beta t_1} \qquad \frac{t_1 \rightsquigarrow_\beta t_1'}{t_1\,t_2 \rightsquigarrow_\beta t_1'\,t_2} \qquad \frac{t_2 \rightsquigarrow_\beta t_2'}{t_1\,t_2 \rightsquigarrow_\beta t_1\,t_2'}$$

Figure 2.13: Inference rules defining the reduction relation for combinators.

As we have been doing, we may generalize this notion for any relation. So a bare β-normal form, for example, is a term which is not a live β-redex. And a $\rightsquigarrow_\alpha$-normal form is a term which cannot be renamed; i.e., a term containing no λ-abstraction (since all λ-abstractions can be renamed).

Definition 2.8.6 (normalizing)**.** *A term t is R-normalizing, denoted $t \downarrow_R$, iff there is an R-normal form t' with $t\ R^*\ t'$. If an R-reduction sequence ends in an R-normal form, we call the sequence R-normalizing as well. If R is left off, we assume it is $\rightsquigarrow$.*

So a normalizing term t is one which has a multi-step β-reduction to a term which then does not reduce (with $\rightsquigarrow$).

Definition 2.8.7 (non-normalizing)**.** *If term t is not R-normalizing, we call it R-non-normalizing, denoted $t \uparrow_R$. As above, if R is left off, we assume it is $\rightsquigarrow$.*

We will see a well-known example of a non-normalizing term in the next chapter (Ω, Section 3.1). Finally, it is also sometimes of interest to consider the equivalence closure of β-reduction with renaming:

Definition 2.8.8 (β-equivalence)**.** *We define $\approx$ as the equivalence closure of $\rightsquigarrow$; i.e., $=_{\rightsquigarrow}.\approx$*

2.9 Combinators

Haskell Curry developed an alternative approach to defining computable functions, without any bound variables. The syntax of these combinators is:

$$\textit{combinators}\ c\ ::=\ S \mid K \mid x \mid c\,c'$$

where free variables x are included to make it easier to express the translation (below) from λ-calculus to combinators.

The semantics of combinators is given in Figure 2.13, where we use the same symbol $\rightsquigarrow$ as we did for β-reduction with renaming. Here, though, as there are no bound variables, there is no renaming.

2.10 Exercises

2.10.1 Basic syntax

1. For each of the following terms, add parentheses to disambiguate between possible parses following the parsing convention (as in Figure 2.2), and then draw the term in tree form.

   (a) $\lambda\, y.\, y\ y\ y$

   (b) $\lambda\, x.\, x\ a\ \lambda\, q.\, x\ q$

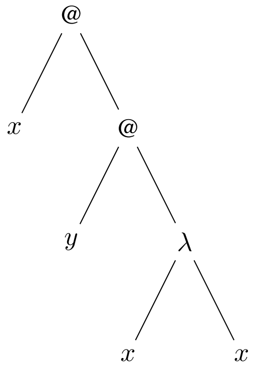
   (c) $(\lambda\, x.\, x\ x)\ \lambda\, x.\, x\ x\ x$

   (d) $\lambda\, f.\, \lambda\, a.\, f\ (f\ (f\ a))$

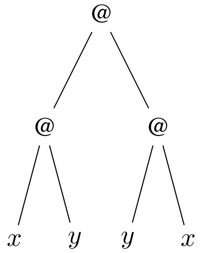
   (e) $(\lambda\, x.\, x\ x)\ \lambda\, y.\, x$

2. For each term shown in tree form below, write that term in textual form, using at least those parentheses (more are fine if you wish) required by our parsing conventions to describe the tree structure correctly.

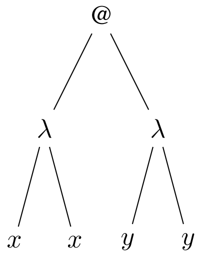
(a)



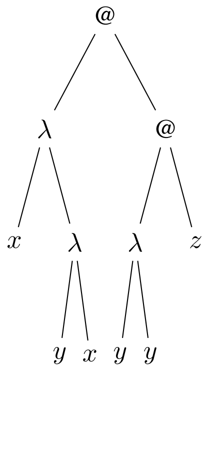
(b)



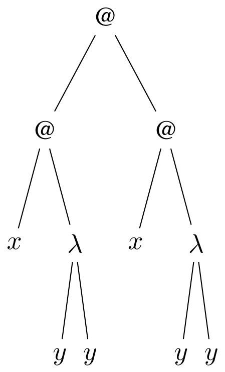
(c)



(d)



(e)



2.10.2 Kinds of variable occurrences

For each of the given terms, draw them in tree form and then indicate, in the same way as in Figure 2.3, which variable occurrences are binding (please underline), which are bound (please circle), and which are free (please box):

1. y
2. $\lambda y.\, y$
3. $(\lambda x.\, x\ x)\ y$
4. $\lambda x.\, (\lambda y.\, x)\ y$
5. $\lambda y.\, \lambda z.\, x\ y\ y\ (w\ z)$

2.10.3 Capture-avoiding substitution

For each of the following, write the result of the substitution or that it is undefined.

1. $[x/y]\lambda\, z.\, y\ y$
2. $[(x\ x)/y]\lambda\, z.\, z\ (y\ y)$
3. $[(x\ x)/y]\lambda\, y.\, z\ y$
4. $[\lambda\, x.\, x/y]\lambda\, z.\, y\ \lambda\, y.\, y\ z$
5. $[\lambda\, x.\, y/z]\lambda\, y.\, y\ z$

2.10.4 Single-step β-reduction

1. Each of the following reductions is allowed by Definition 2.5.8. For each reduction, indicate the instantiations of the meta-variables x, t, t', and t'' of Definition 2.5.8 (as in the examples in Section 2.5.1).

   (a) $\lambda\, x.\, (\lambda\, y.\, y)\; \lambda\, z.\, z \;\rightsquigarrow_\beta\; \lambda\, x.\, \lambda\, z.\, z$

   (b) $(\lambda\, y.\, y)\; (z\; z)\; z \;\rightsquigarrow_\beta\; z\; z\; z$

   (c) $z\; (\lambda\, y.\, (\lambda\, z.\, z\; z)\; (y\; y)) \rightsquigarrow_\beta z\; \lambda\, y.\, y\; y\; (y\; y)$

   (d) $(\lambda\, x.\, \lambda\, y.\, x\; x)\; (z\; \lambda\, z.\, z) \rightsquigarrow_\beta \lambda\, y.\, z\; (\lambda\, z.\, z)\; (z\; \lambda\, z.\, z)$

   (e) $(\lambda\, x.\, \lambda\, y.\, y)\; z \rightsquigarrow_\beta \lambda\, y.\, y$

2. Write derivations using the rules of Figure 2.5 for the β-reductions of parts (a), (b), (c) of the previous problem.

2.10.5 α-equivalence

For each pair of terms, indicate whether or not they are α-equivalent:

1. $\lambda\, x.\, \lambda\, y.\, y\; x$ and $\lambda\, x.\, \lambda\, x.\, y\; x$

2. $x\; \lambda\, y.\, x$ and $x\; \lambda\, w.\, x$

3. $x\; \lambda\, y.\, x$ and $y\; \lambda\, x.\, y$

4. $\lambda\, z.\, (\lambda\, x.\, x\; z)\; \lambda\, y.\, z\; y$ and $\lambda\, q.\, (\lambda\, y.\, y\; q)\; \lambda\, y.\, q\; y$

2.10.6 Multi-step β-reduction

1. Write maximal $\rightsquigarrow$-reduction sequences starting with the given term. Be careful with your renamings!

   (a) $\lambda\, y.\, (\lambda\, x.\, \lambda\, y.\, x\; (x\; y))\; \lambda\, x.\, y\; x$

   (b) $\lambda\, x.\, \lambda\, y.\, (\lambda\, z.\, \lambda\, y.\, \lambda\, x.\, z\; z)\; \lambda\, z.\, x\; y$

2. Find an example of a term t and number n of β-steps where

   - there exists a term t' such that $t(=_\alpha \rightsquigarrow_\beta^n =_\alpha \rightsquigarrow_\beta)t'$, but
   - there does not exist a term t' such that $t(=_\alpha \rightsquigarrow_\beta^{n+1})t'$.

   In other words, this problem asks you to find an example of a term where a second α-equivalence step is required in order to complete a sequence of β-reductions. As a hint (because the problem is a bit tricky):

- If all bound and free variables are distinct from each other, then it is never necessary to rename to perform a single β-reduction, so it might seem like an initial $=_\alpha$-step would suffice to obviate any subsequent renamings...
- ...but it is not! Ask yourself if there is a way that starting with a term where all bound and free variables are distinct from each other, one could arrive at a term that does not have that property; and then use this to construct a term where a second $=_\alpha$-step is required.

Chapter 3

Programming in Lambda Calculus

3.1 Basic functions

Let us consider a few basic lambda terms that are useful for programming in lambda calculus. We will give names to the terms we consider, as meta-linguistic abbreviations. Let us use the syntax $N := t$ to indicate (in our meta-language) that we wish to use name N as an abbreviation for term t. In all cases, our choice of names will be justified by the behavior of the term when applied to various arguments. By behavior, I mean the term's β-reductions.

The identity function.

$$\mathit{id} := \lambda\, x.\, x$$

This term really does behave like the mathematical (set-theoretic, let us say) identity function, since if we apply *id* to anything, we just get back that same value. We have

$$\mathit{id}\, t \quad \leadsto^* \quad t$$

Self-application operator.

$$\delta := \lambda\, x.\, x\; x$$

This operator applies input x to x. We also have

$$\Omega := \delta\; \delta$$

This term has no normal form, reducing forever to itself:

$$\begin{array}{ll} \underline{\delta}\; \delta & = \\ \underline{(\lambda\, x.\, x\; x)\; \delta} & \leadsto \\ \delta\; \delta & \end{array}$$

Composition.

$$\mathit{compose} := \lambda\, f.\, \lambda\, g.\, \lambda\, x.\, f\; (g\; x)$$

This term can be applied to any terms f and g, and it will return a term that behaves like their composition: it applies f after g to its input x. It is nice to borrow mathematical notation for function composition and write $f \circ g$ for *compose* $f\; g$. Here are a few examples using *compose*:

- $\mathit{id} \circ \mathit{id} \quad \leadsto^* \quad \mathit{id}$

- $\delta \circ \delta \;\; \leadsto^* \;\; \lambda\, x.\, x\; x\; (x\; x)$:

$$\begin{array}{ll}
\underline{\delta \circ \delta} & = \\
\underline{\textit{compose}}\; \delta\; \delta & = \\
\underline{(\lambda\, f.\, \lambda\, g.\, \lambda\, x.\, f\; (g\; x))\; \delta}\; \delta & \leadsto \\
\underline{(\lambda\, g.\, \lambda\, x.\, \delta\; (g\; x))\; \delta} & \leadsto \\
\lambda\, x.\, \delta\; (\delta\; x) & \leadsto \\
\lambda\, x.\, \delta\; \underline{(\delta\; x)} & \leadsto \\
\lambda\, x.\, \underline{\delta\; (x\; x)} & \leadsto \\
\lambda\, x.\, (x\; x)\; (x\; x) &
\end{array}$$

Application operator.

$$\textit{app} \coloneqq \lambda\, x.\, \lambda\, y.\, x\; y$$

This term takes in inputs x and y and returns the result of applying x to y. So it is a term which acts like the term construct of application.

3.2 Representing numbers with the Church encoding

For Church's original goal of a foundation for mathematics, it is paramount that there is some way to represent natural numbers, and the intuitively computable operations on them, as lambda terms. Happily, there are several such *lambda encodings* for representing data as lambda terms. Here we see the first, which is Church's own encoding.

Any lambda encoding must represent data as lambda terms implementing some behavioral interface. That is, data are defined by what they do, not what they are. The idea of the Church encoding more specifically is to define numbers as their own iteration functions: functions which take in another function f and starting point x, and repeatedly call f starting with x. Let us write $\ulcorner n \urcorner$ to mean the lambda term representing $n \in \mathbb{N}$. Then the Church encoding defines:

$$\ulcorner n \urcorner \;\; = \;\; \lambda\, f.\, \lambda\, x.\, \underbrace{f\; (\cdots (f}_{n}\;\; x))$$

So we have these concrete examples:

$$\begin{array}{lcl}
0 & \coloneqq & \lambda\, f.\, \lambda\, x.\, x \\
1 & \coloneqq & \lambda\, f.\, \lambda\, x.\, f\; x \\
2 & \coloneqq & \lambda\, f.\, \lambda\, a.\, f\; (f\; x) \\
3 & \coloneqq & \lambda\, f.\, \lambda\, a.\, f\; (f\; (f\; x)) \\
\cdots & &
\end{array}$$

In passing, we may observe that 1 and *app* are α-equivalent terms. When representing data as lambda terms, such coincidences sometimes occur.

3.3 Operations on Church-encoded natural numbers

Let us see now how to define some basic operations on Church-encoded natural numbers.

Successor. The mathematical successor operation on $\mathbb{N}$ takes in n and returns $n+1$ (i.e., the next number). Here is the definition for Church-encoded naturals:

$$\textit{succ} \;\coloneqq\; \lambda\, n.\, \lambda\, f.\, \lambda\, x.\, f\; (n\; f\; x)$$

To understand this, let us first see an example:

$$\begin{array}{ll}
\underline{\mathit{succ}}\ 2 & = \\
\underline{(\lambda\, n.\, \lambda\, f.\, \lambda\, x.\, f\ (n\ f\ x))\ 2} & \leadsto \\
\lambda\, f.\, \lambda\, x.\, f\ (\underline{2}\ f\ x) & = \\
\lambda\, f.\, \lambda\, x.\, f\ (\underline{\lambda\, f.\, (\lambda\, x.\, f\ (f\ x))\ f}\ x) & \leadsto \\
\lambda\, f.\, \lambda\, x.\, f\ (\underline{(\lambda\, x.\, f\ (f\ x))\ x}) & \leadsto \\
\lambda\, f.\, \lambda\, x.\, f\ (f\ (f\ x)) & = \\
3 &
\end{array}$$

More generally, if $n \in \mathbb{N}$, then *succ* $\ulcorner n \urcorner$ reduces to $\ulcorner n + 1 \urcorner$.

Addition. To compute $\ulcorner m + n \urcorner$ from $\ulcorner m \urcorner$ and $\ulcorner n \urcorner$, the idea is similar to that for successor, except that here we wish to add not just one f to the left of $\underbrace{f \cdots (f}_{n}\ x)$, but m applications of f. For then we would have $m + n$ applications of f as desired for $\ulcorner m + n \urcorner$. And fortunately, $\ulcorner m \urcorner$ itself gives us the power to apply f m times to some starting value Q, by writing $m\ f\ Q$. Here, we want $n\ f\ x$ for Q. So the definition of addition is:

$$\mathit{add} := \lambda\, m.\, \lambda\, n.\, \lambda\, f.\, \lambda\, x.\, m\ f\ (n\ f\ x)$$

Predecessor. Kleene was the first to crack the puzzle of how to compute the $\ulcorner n \urcorner$ from $\ulcorner n + 1 \urcorner$. His definition is somewhat complicated, so here is a simpler one. To my knowledge, this is original.

$$\begin{array}{lll}
\mathit{just} & := & \lambda\, n.\, \lambda\, j.\, \lambda\, k.\, j\ n \\
\mathit{pred} & := & \lambda\, n.\, n\ (\lambda\, m.\, \mathit{just}\ (m\ \mathit{succ}\ 0))\ 0\ \mathit{id}\ 0
\end{array}$$

Writing F for $\lambda\, m.\, \mathit{just}\ (m\ \mathit{succ}\ 0)$, let us first see how $2\ F\ 0$ computes:

$$\begin{array}{ll}
\underline{2\ F}\ 0 & \leadsto \\
\underline{(\lambda\, a.\, F\ (F\ a))\ 0} & \leadsto \\
F\ (\underline{F\ 0}) & \leadsto \\
F\ (\mathit{just}\ (\underline{0\ \mathit{succ}}\ 0)) & \leadsto \\
F\ (\mathit{just}\ (\underline{\lambda\, a.\, a\ 0})) & \leadsto \\
F\ (\underline{\mathit{just}\ 0}) & \leadsto \\
\underline{F\ \lambda\, j.\, \lambda\, k.\, j\ 0} & \leadsto \\
\mathit{just}\ (\underline{(\lambda\, j.\, \lambda\, k.\, j\ 0)\ \mathit{succ}}\ 0) & \leadsto \\
\mathit{just}\ (\underline{(\lambda\, k.\, \mathit{succ}\ 0)\ 0}) & \leadsto \\
\underline{\mathit{just}\ (\mathit{succ}\ 0)} & \leadsto \\
\lambda\, j.\, \lambda\, k.\, j\ (\mathit{succ}\ 0) &
\end{array}$$

Now here is the reduction for *pred* 2:

$$\begin{array}{ll}
\underline{\mathit{pred}\ 2} & \leadsto \\
\underline{2\ F\ 0}\ \mathit{id}\ 0 & \leadsto^{*} \text{ [by above reduction sequence]} \\
\underline{(\lambda\, j.\, \lambda\, k.\, j\ (\mathit{succ}\ 0))\ \mathit{id}}\ 0 & \leadsto \\
\underline{(\lambda\, k.\, \mathit{id}\ (\mathit{succ}\ 0))\ 0} & \leadsto \\
\underline{\mathit{id}\ (\mathit{succ}\ 0)} & \leadsto \\
\underline{\mathit{succ}\ 0} & \leadsto^{*} \\
1 &
\end{array}$$

Another predecessor. Here is a different, trickier definition of predecessor, which one can find online (sadly, I do not know who invented it):

$$\mathit{pred} := \lambda\, n.\, \lambda\, f.\, \lambda\, x.\, n\ (\lambda\, g.\, \lambda\, h.\, h\ (g\ f))\ (\lambda\, h.\, x)\ \mathit{id}$$

To understand how this works, let us write F for $\lambda\, g.\, \lambda\, h.\, h\ (g\ f)$, and A for $\lambda\, h.\, x$, and see how $3\ F\ A$ computes:

$$\begin{array}{ll}
\underline{3\ F}\ A & \leadsto \\
(\lambda\, x.\, F\ (F\ (F\ x)))\ A & \leadsto \\
F\ (F\ (\underline{F\ A})) & = \\
F\ (F\ (\underline{(\lambda\, g.\, \lambda\, h.\, h\ (g\ f))\ \lambda\, h.\, x})) & \leadsto \\
F\ (F\ (\lambda\, h.\, h\ \underline{((\lambda\, h.\, x)\ f)})) & \leadsto \\
F\ (\underline{F}\ (\lambda\, h.\, h\ x)) & = \\
F\ (\underline{(\lambda\, g.\, \lambda\, h.\, h\ (g\ f))\ (\lambda\, h.\, h\ x)}) & \leadsto \\
F\ (\lambda\, h.\, h\ \underline{((\lambda\, h.\, h\ x)\ f)}) & \leadsto \\
\underline{F}\ (\lambda\, h.\, h\ (f\ x)) & = \\
\underline{(\lambda\, g.\, \lambda\, h.\, h\ (g\ f))\ (\lambda\, h.\, h\ (f\ x))} & \leadsto \\
\lambda\, h.\, h\ \underline{((\lambda\, h.\, h\ (f\ x))\ f)} & \leadsto \\
\lambda\, h.\, h\ (f\ (f\ x)) &
\end{array}$$

What is happening here? We see that $3\ F\ A$ has reduced to something similar to $f\ (f\ (f\ x))$, but with a critical twist: we have λ-abstracted away the function for the first call to f, leaving the other calls intact. This gives us what we could think of as a "flexible" version of $f\ (f\ (f\ x))$, where we get to choose which function to call instead of f for the outer application. And the definition of predecessor makes use of this flexibility by applying the whole result to *id*. That produces, then, just $f\ (f\ x)$. So, understanding F and A to be grafted into the expression on the second line below (capturing their free variables f and x), we have

$$\begin{array}{ll}
\mathit{pred}\ 3 & \leadsto^* \\
\lambda\, f.\, \lambda\, x.\, \underline{3\ F\ A}\ \mathit{id} & \leadsto^* \\
\lambda\, f.\, \lambda\, x.\, \underline{(\lambda\, h.\, h\ (f\ (f\ x)))\ \mathit{id}} & \leadsto \\
\lambda\, f.\, \lambda\, x.\, \underline{(\mathit{id}\ (f\ (f\ x)))} & \leadsto \\
\lambda\, f.\, \lambda\, x.\, f\ (f\ x) & = \\
2 &
\end{array}$$

In some ways this is similar, as perhaps is inevitable, to the first version of predecessor we saw: a value is computed from $\ulcorner n \urcorner$ that is like $\ulcorner n \urcorner$ but allows calling another function – in particular, *id* – instead of a final successor. In this second version of predecessor, that value is computed underneath bindings of f and x, so that *id* gets called on applications of f to x. In the first version of predecessor, *id* gets called on the entire predecessor term, including bindings of f and x.

3.4 Representing booleans

A simpler datatype than that of the natural numbers is the boolean type, with values *true* and *false*. The Church encoding of this type is

$$\begin{array}{lll}
\mathit{true} & := & \lambda\, x.\, \lambda\, y.\, x \\
\mathit{false} & := & \lambda\, x.\, \lambda\, y.\, y
\end{array}$$

Each boolean accepts two inputs (one at a time), and returns one of these. *true* returns the first, while *false* returns the second. Based on this idea, it is easy to see how to define various boolean operations:

$$\begin{array}{lll}
\mathit{not} & := & \lambda\, x.\, x\ \mathit{false}\ \mathit{true} \\
\mathit{and} & := & \lambda\, x.\, \lambda\, y.\, x\ y\ \mathit{false}
\end{array}$$

We negate (with *not*) a boolean by applying it to *false* and then *true*. If the boolean itself is *true*, then it will return the first of these two inputs, namely *false*; if it is *false*, it will return *true*. This is the desired behavior. Similarly, *and* takes in inputs *x* and *y*. It returns the result of applying *x* to *y* and then *false*. If *x* is *true*, then the result will be *y*; and this is what we would like for conjunction, since if the first boolean (*x*) is true, the conjunction's value coincides with the value of the second (y): true if y is true, and false if y is false. And if x is *false*, then the second input (out of y and *false*) will be chosen; again, the desired behavior, since this means conjoining *false* with anything will reduce to *false*.

It is worth emphasizing that applying boolean operations to values that are not booleans does not result in an error as it might in some programming languages. Here, every lambda term has a well-defined behavior in the form of its β-reductions. But the results of applications violating the intuitive typings we have in mind for these operations may be somewhat inscrutable.

3.5 Ordered pairs

It is often convenient to program with a representation of ordered pairs (x, y), given representations of x and y. To construct the representation of a pair, we use this function:

$$\mathit{pair} \;:=\; \lambda\, x.\, \lambda\, y.\, \lambda\, c.\, c\; x\; y$$

The idea is that given the components x and y of the pair, we represent (by the *pair* function) the pair itself as $\lambda\, c.\, c\; x\; y$. This definition embodies the idea that a pair of x and y is something that can make x and y available for subsequent computation. This is done in the encoding by applying the pair to a function which is expecting the components.

For example, we may define *fst* (“first”) and *snd* (“second”; these names are often used for these operations in functional programming languages) as follows:

$$\begin{array}{rcl} \mathit{fst} & := & \lambda\, p.\, p\; \mathit{true} \\ \mathit{snd} & := & \lambda\, p.\, p\; \mathit{false} \end{array}$$

Since *true* returns the first of two arguments, and *false* the second, they are used to select either the first or the second component, respectively, when passed as an argument to the pair. (As usual, these functions assume input *p* is a pair of the form $\lambda\, c.\, c\; x\; y$, and may give unexpected results if applied to terms not of that form.)

3.6 Representing numbers with the Scott encoding

In Section 3.2, we saw an elegant representation of numbers as lambda terms, called the Church encoding. Every number n is represented as the n-fold composition operator. While many functions are concisely definable this way, the predecessor operation required quite some ingenuity, and is asymptotically less efficient than we might reasonably expect (taking time linear in n, instead of constant time). In this section, we consider an alternative lambda encoding due to Dana Scott, which has a straightforward constant-time predecessor. With the Scott encoding, each number can be thought of as a function t that informs the caller whether t represents a successor number or zero. In the former case, it also provides the caller with the representation of the predecessor. The definition is:

$$\begin{array}{lcl} 0 & := & \lambda\, f.\, \lambda\, x.\, x \\ 1 & := & \lambda\, f.\, \lambda\, x.\, f\; 0 \\ 2 & := & \lambda\, f.\, \lambda\, x.\, f\; 1 \\ \ldots & & \\ \ulcorner n+1 \urcorner & := & \lambda\, f.\, \lambda\, x.\, f\; \ulcorner n \urcorner \\ \ldots & & \end{array}$$

So every $\ulcorner n \urcorner$ accepts two inputs f and x, and if n is 0, returns x; and if n is $m+1$ for some m, returns $f\; \ulcorner m \urcorner$. This makes available the predecessor $\ulcorner m \urcorner$, and thus the actual predecessor function is easily defined:

$$\mathit{pred} \;:=\; \lambda\, n.\, n\; \mathit{id}\; 0$$

Here, *id* is passed for f and 0 for x. This means that for any Scott-encoded successor number, we have the following reduction:

$$\begin{array}{ll}
\underline{\mathit{pred}}\,\ulcorner n+1 \urcorner & = \\
\underline{(\lambda\, n.\, n\; \mathit{id}\; 0)\; \ulcorner n+1 \urcorner} & \leadsto \\
\underline{\ulcorner n+1 \urcorner}\; \mathit{id}\; 0 & = \\
\underline{(\lambda\, f.\, \lambda\, x.\, f\, \ulcorner n \urcorner)\; \mathit{id}}\; 0 & \leadsto \\
\underline{(\lambda\, x.\, \mathit{id}\, \ulcorner n \urcorner)\; 0} & \leadsto \\
\underline{\mathit{id}\, \ulcorner n \urcorner} & \leadsto \\
\ulcorner n \urcorner &
\end{array}$$

And this is the desired result: $\mathit{pred}\,\ulcorner n+1 \urcorner \leadsto^* \ulcorner n \urcorner$. Furthermore, we can see that this reduction required four steps of β-reduction, independent of the value of n. This is in contrast to the case with the Church encoding, where the number of steps was proportional to n.

It is obvious from the encoding that the successor function *succ* for Scott-encoded numbers should be:

$$\mathit{succ} \;:=\; \lambda\, n.\, \lambda\, f.\, \lambda\, x.\, f n$$

3.7 The Y combinator

While it is very straightforward to define predecessor on Scott-encoded numbers, other operations pose a problem. The Church encoding takes n-fold iteration as the representation of n, and hence has no difficulty defining iterative functions. Not so the Scott encoding, and indeed, the only natural way to recurse is to avail ourselves of a term implementing *general recursion* (this is recursion that may fail to terminate). (It should be noted that there is an extremely tricky way to derive iteration for Scott encodings, but we will not consider this here [11].)

General recursion in lambda calculus is provided using a term traditionally denoted Y:

$$Y \;:=\; \lambda\, f.\, (\lambda\, x.\, f\; (x\; x))\; (\lambda\, x.\, f\; (x\; x))$$

This term is usually called a *combinator*, which is an informal notion indicating that a lambda term is of interest primarily for use as a building block for defining other functions (as opposed, say, to implementing some particular algorithm valuable in its own right). In this sense, some other terms we have encountered so far, like identity and composition functions (*compose*, *id* of Section 3.1), are also reasonably considered combinators.

Terminology aside, let us see how the Y combinator works and how we can use it to define operations on Scott-encoded numbers. Suppose t is any lambda term not containing x free. Then we have:

$$\begin{array}{ll}
\underline{Y}\; t & = \\
\underline{(\lambda\, f.\, (\lambda\, x.\, f\; (x\; x))\; (\lambda\, x.\, f\; (x\; x)))\; t} & \leadsto \\
\underline{(\lambda\, x.\, t\; (x\; x))\; (\lambda\, x.\, t\; (x\; x))} & \leadsto \\
t\; \underline{((\lambda\, x.\, t\; (x\; x))\; (\lambda\, x.\, t\; (x\; x)))} & =_\beta \\
t\; (Y\; t) &
\end{array}$$

So we see that $Y\; t$ is β-equivalent to $t\; (Y\; t)$. This fact is so important that it is worth highlighting as an equation:

$$Y\; t \;=_\beta\; t\; (Y\; t)$$

Swapping sides will shortly be revealing:

$$t\; (Y\; t) \;=_\beta\; Y\; t$$

This matches the form of a fixed-point equation for t. In mathematics, a fixed point of a function F is an input X such that

$$F(X) = X$$

Here, with application of lambda terms playing the role of function invocation, and β-equivalence taking the place of equality, we can write this as:

$$F\; X =_\beta X$$

For term t, this becomes

$$t\ X =_\beta X$$

And indeed, the equation we derived above is of this form, with $Y\ t$ for X.

Now what is the significance of this? It shows us that, contrary to what we usually find in mathematics, in lambda calculus every function has a fixed point. How peculiar! Certainly some mathematical functions have fixed points. Take (mathematical) predecessor on natural numbers, with the assumption that *pred* of 0 is 0. Then 0 is a fixed point of *pred*. But consider boolean negation. There is no boolean b such that *not* b equals b (neither possible value for b, namely *true* or *false*, works). Strangely, though, in lambda calculus, we have just seen the general equation of $t\ (Y\ t)$ and $Y\ t$. This means that

$$\mathit{not}\ (Y\ \mathit{not}) =_\beta Y\ \mathit{not}$$

Something unusual is going on, and indeed, as we will see when we turn to denotational semantics of lambda calculus, interpreting lambda calculus in set theory requires significant ingenuity.

But to remain at the linguistic level for the moment, let us try to get an intuition for how every term t can have $Y\ t$ for a fixed point. Let us write U for $\lambda\, x.\, t\ (x\ x)$. We have seen that

$$\begin{array}{ll} \underline{Y\ t} & \leadsto \\ \underline{U\ U} & \leadsto \\ t\ (U\ U) & \end{array}$$

This reduction sequence may then be continued as long as we wish:

$$\begin{array}{ll} t\ \underline{(U\ U)} & \leadsto \\ t\ (t\ \underline{(U\ U)}) & \leadsto \\ t\ (t\ (t\ \underline{(U\ U)})) & \leadsto \\ \cdots & \end{array}$$

If we had some notion of infinite lambda term, we might identify the limit of this infinite reduction sequence, as this infinite right-nested application of t:

$$t\ (t\ (t\ \cdots$$

One can indeed develop an infinitary lambda calculus allowing infinitary terms like this [9]; but this is beyond the scope of the book. But with an infinitary term like this as an informal guiding intuition, we can see how the fixed-point equation makes sense. $Y\ t$ denotes (informally) an infinite right-nested application of t. Applying t one more time to this does not change the infinite application, as it is still infinite!

Note that $U\ U$ is a lot like $\delta\ \delta$:

$$\begin{array}{lcl} U\ U & = & (\lambda\, x.\, t\ (x\ x))\ \lambda\, x.\, t\ (x\ x) \\ \delta\ \delta & = & (\lambda\, x.\, x\ x)\ \lambda\, x.\, x\ x \end{array}$$

We have just inserted t, but otherwise retain the central idea of self-application for divergence.

How is this esoterically explained construction useful for programming? Contrast the situation with iteration using Church-encoded numbers. There, $\ulcorner n \urcorner$ gives us the power to repeat a function n times:

$$\underbrace{t \cdots (t}_{n}\ x)$$

But what if we need to repeat a function more times than just n times? We could imagine somehow increasing how many times the composition is iterated, to some bigger number n'. But the most computationally powerful option is to extend the n-fold iteration of t to an infinite iteration of t:

$$t\ (t\ (t\ \cdots$$

But this is just what (informally) $Y\ t$ gives us! So we are using the power of diverging computation which we get through self-application, to allow ourselves as many iterations of t as we could possibly need. Fundamental results of recursion theory then imply that we will of necessity need to accept the possibility of divergence: we have given ourselves the ability to apply t as many times as we wish, and we cannot rule out the possibility that it gets applied infinitely many times with no normal form reachable.

But there is still a puzzle. How can we ever reach any normal form when $Y\ t$ has an infinite reduction sequence? The answer is that existence of a single infinite reduction sequence does not mean all reduction sequences are infinite. Indeed, for a very simple example, consider

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

This term has both an infinite reduction sequence, and also infinitely many finite reduction sequences. For examples of the first and second, in order, consider:

$$\begin{array}{l}
(\lambda\, x.\, \lambda\, y.\, y)\ \underline{\Omega}\ \rightsquigarrow\ (\lambda\, x.\, \lambda\, y.\, y)\ \underline{\Omega}\ \rightsquigarrow\ \cdots \\
\underline{(\lambda\, x.\, \lambda\, y.\, y)\ \Omega}\ \rightsquigarrow\ \lambda\, y.\, y
\end{array}$$

The normalizing reduction sequence (the second one) drops out the non-normalizing Ω subterm. Similarly, in our infinitary term

$$t\ (t\ (t \cdots$$

it could happen that there is a reduction to a normal form where an application of t ends up dropping its argument. We will see an example next.

3.8 Recursive operations on Scott-encoded numbers

Let us define addition on Scott-encoded numbers using the Y combinator. The idea is that we wish to implement the following system of recursive equations, using Y to implement the recursion:

$$\begin{array}{lcl}
\mathit{add}\ 0\ m & = & m \\
\mathit{add}\ (\mathit{succ}\ p)\ m & = & \mathit{succ}\ (\boxed{\mathit{add}}\ p\ m)
\end{array}$$

Since the Scott-encoding gives us a way to distinguish whether a number is 0 or a successor number, we can easily choose whether between these equations based on the first input. We then need to use Y to implement the framed recursion on the right-hand side of the second equation. The definition is, the following, using helper definition *addh* for easier consideration below:

$$\begin{array}{lcl}
\mathit{addh} & := & \lambda\, \mathit{add}.\, \lambda\, n.\, \lambda\, m.\, n\ (\lambda\, p.\, \mathit{succ}\ (\mathit{add}\ p\ m))\ m \\
\mathit{add} & := & Y\ \mathit{addh}
\end{array}$$

Let us see how this works with an example, writing U for $\lambda\, x.\, \mathit{addh}\ (x\ x)$:

$$\begin{array}{ll}
\underline{\mathit{add}}\ 2\ 2 & = \\
\underline{Y\ \mathit{addh}}\ 2\ 2 & \rightsquigarrow \\
\underline{U\ U}\ 2\ 2 & \rightsquigarrow \\
\underline{\mathit{addh}}\ (U\ U)\ 2\ 2 & = \\
\underline{(\lambda\, \mathit{add}.\, \lambda\, n.\, \lambda\, m.\, n\ (\lambda\, p.\, \mathit{succ}\ (\mathit{add}\ p\ m))\ m)\ (U\ U)}\ 2\ 2 & \rightsquigarrow \\
\underline{(\lambda\, n.\, \lambda\, m.\, n\ (\lambda\, p.\, \mathit{succ}\ (U\ U\ p\ m))\ m)\ 2\ 2} & \rightsquigarrow^2 \\
\underline{2}\ (\lambda\, p.\, \mathit{succ}\ (U\ U\ p\ 2))\ 2 & = \\
\underline{(\lambda\, f.\, \lambda\, x.\, f\ 1)\ (\lambda\, p.\, \mathit{succ}\ (U\ U\ p\ 2))\ 2} & \rightsquigarrow^2 \\
\underline{(\lambda\, p.\, \mathit{succ}\ (U\ U\ p\ 2))\ 1} & \rightsquigarrow \\
\mathit{succ}\ \underline{(U\ U\ 1\ 2)}) & \rightsquigarrow^* \\
\mathit{succ}\ (\mathit{succ}\ \underline{(U\ U\ 0\ 2)}) & \rightsquigarrow^* \\
\mathit{succ}\ (\mathit{succ}\ 2) & \rightsquigarrow^* \\
4 &
\end{array}$$

We see in detail that $U\ U\ \ulcorner n+1 \urcorner\ m$ reduces to $\mathit{succ}\ (U\ U\ \ulcorner n \urcorner\ m)$. So we peel successors off the first argument until we reach 0, and then we return the second argument (i.e., m).

3.9 A direct approach to recursion on Scott-encoded numbers

Instead of using the Y combinator, it is possible to recurse directly on Scott-encoded numbers. A Scott-encoded number takes in a function f to call with the predecessor if the number is non-zero, and a value x to return if the number is zero. The key idea, attributed by Lepigre and Rafalli to Michel Parigot, is to have f and x themselves expect to be called with f and x again [12]. This enables in particular f to recurse.

Here is a definition of addition for the Scott encoding, using this idea. I have abstracted out f and x for this case, to help make clear where there is a self-application happening. The base case x depends, in the definition of addition, on the second addend, so we need to write $x\ m$ in the definition of *add*, instead of just x.

$$\begin{array}{rcl} f & := & \lambda\, p.\, \lambda\, s.\, \lambda\, z.\, \mathit{succ}\ (p\ s\ z\ s\ z) \\ x & := & \lambda\, m.\, \lambda\, s.\, \lambda\, z.\, m \\ \mathit{add} & := & \lambda\, n.\, \lambda\, m.\, n\ f\ (x\ m)\ f\ (x\ m) \end{array}$$

Let us see this definition in action:

$$\begin{array}{ll} \underline{\mathit{add}\ 2\ 2} & \rightsquigarrow^2 \\ \underline{2\ f}\ (x\ 2)\ f\ (x\ 2) & \rightsquigarrow \\ \underline{f\ 1\ f\ (x\ 2)} & \rightsquigarrow^3 \\ \mathit{succ}\ \underline{(1\ f\ (x\ 2)\ f\ (x\ 2))} & \rightsquigarrow^4 \\ \mathit{succ}\ (\mathit{succ}\ \underline{(0\ f\ (x\ 2)\ f\ (x\ 2))}) & \rightsquigarrow \\ \mathit{succ}\ (\mathit{succ}\ \underline{((x\ 2)\ f\ (x\ 2))}) & \rightsquigarrow^3 \\ \underline{\mathit{succ}\ (\mathit{succ}\ 2)} & \rightsquigarrow^* \\ 4 & \end{array}$$

3.10 The Parigot encoding

Before (it seems) his discovery of a way to recurse on Scott-encoded numbers, Parigot proposed an encoding that combines the Church and Scott encodings:

$$\begin{array}{lcl} 0 & := & \lambda\, f.\, \lambda\, x.\, x \\ 1 & := & \lambda\, f.\, \lambda\, x.\, f\ 0\ x \\ 2 & := & \lambda\, f.\, \lambda\, x.\, f\ 1\ (f\ 0\ x) \\ 3 & := & \lambda\, f.\, \lambda\, x.\, f\ 2\ (f\ 1\ (f\ 0\ x)) \\ \cdots & & \\ \ulcorner n+2 \urcorner & := & \lambda\, f.\, \lambda\, x.\, f\, \ulcorner n+1 \urcorner\, (f\, \ulcorner n \urcorner\, (\cdots\ (f\ 0\ x))) \\ \cdots & & \end{array}$$

Another way to see the encoding is to observe that for every n,

$$\ulcorner n+1 \urcorner \approx \lambda\, f.\, \lambda\, x.\, f\, \ulcorner n \urcorner\, (\ulcorner n \urcorner\ f\ x)$$

where ≈ denotes β-equivalence (Definition 2.8.8)

3.11 Exercises

3.11.1 β-reductions for some simple terms

1. For each of the following terms, write down a β-normal form to which the term reduces. You do not need to write out all the steps in a β-reduction sequence. Please just give a β-normal form.

   (a) *app* $\circ$ *id*

(b) *app* $\circ$ *app*

(c) $\lambda\, z.\, 2\; z$

(d) δ *app*

(e) $2\; 2$

(f) *not* 2 (as noted above, terms like this which violate intuitive typings do have a well-defined behavior)

(g) *and* 1

2. Please write out a maximal β-reduction sequence (renaming is not necessary, so we can use just $\rightsquigarrow_\beta$ instead of $\rightsquigarrow$) starting with *pred* 2.

3.11.2 Programming in lambda calculus

1. Define a disjunction operator (i.e., boolean "or") on Church-encoded booleans, and demonstrate that it is working by writing a maximal $\rightsquigarrow$-reduction sequence starting with *or false true*.

2. **[Challenge]** Find an alternative definition of *pred* with similar form as above, namely

$$\lambda\, n.\, \lambda\, f.\, \lambda\, x.\, n\; F'\; A'\; t_1 \cdots t_k$$

for some terms F' and A' grafted into this expression (which hence might have free occurrences of f or x that get bound by the λ-abstractions of those two variables), and some extra terms $t_1, \cdots, t_k$. The critical requirement is that where $\ulcorner n+1 \urcorner\; F\; A$ reduces (with the definition of F and A in the text) to $\lambda\, h.\, h\; (\underbrace{f \cdots (f}_{n}\; x))$, your version with your F' and A' and some of your extra terms should reduce to $\lambda\, h.\; \underbrace{f \cdots (f}_{n}\; (h\; x))$.

3. Define a function *flip* which reverses the order of components in a pair.

4. Define a subtraction operation on Scott-encoded numbers. Your term is free to invoke *pred* for predecessor, and the Y combinator (and other terms we have defined so far, if you wish).

5. The term $Y\; Y$ has many different (infinite) reductions. Try to indicate a little of the complexity of this term by showing prefixes of some of its reduction sequences. It would be interesting to organize these initial parts of reduction sequences into a tree, so we can see how reduction can branch out in different ways from the starting point of $Y\; Y$. (This problem is not concerned with giving an exact correct answer, but rather with showing that you have explored the reduction behavior of this rather exotic term.)

Chapter 4

Confluence

In this chapter, we prove a basic property of the lambda calculus, called confluence. Given a binary relation → on a set A, an element $a \in A$ is confluent iff no matter which pair of →-paths we follow from a, ending in some elements b and c, there is a pair of →-paths from b and c ending in a common element d. This can be expressed pictorially as:

$$\begin{tikzcd}
& a \arrow[dl, "*"'] \arrow[dr, "*"] & \\
b \arrow[dr, dashed, "*"'] & & c \arrow[dl, dashed, "*"] \\
& d &
\end{tikzcd}$$

Confluence can be seen as a generalization of determinism (Definition 2.5.5, the property that whenever $a \to b$ and $a \to c$, we have $b = c$). For it might happen that we have paths from a that can reach distinct elements b and c (that is, $a \to^* b$ and $a \to^* c$, with $b \neq c$), but these elements can be joined back at some element d (so $b \to^* d$ and $c \to^* d$). So a is nondeterministic, yet in a somewhat controlled way: no matter which two paths we follow from a, there is always some way to reconverge.

In this chapter, we will prove that the relation $\rightsquigarrow_\alpha \cup \rightsquigarrow_\beta$ is confluent. This relation subsumes the relation $\rightsquigarrow$ of single-step β-reduction with renaming (Definition 2.8.2), because it allows any sequence of $\rightsquigarrow_\alpha$ and $\rightsquigarrow_\beta$ steps, while a non-trivial $\rightsquigarrow$-reduction sequence must end with a $\rightsquigarrow_\beta$ step. Working with the more permissive relation will enable a cleaner formulation of confluence. The proof we will follow is attributed to William Tait and Per Martin-Löf (but never published by either of them). Some of our discussion is quite generic, however, and applies to any binary relation → over some set of elements (not necessarily terms).

We will use the notation $\hookrightarrow$ for the relation that we will study in this chapter:

Definition 4.0.1. $\hookrightarrow$ *is defined to be* $\rightsquigarrow_\alpha \cup \rightsquigarrow_\beta$.

4.1 The diamond property

A property quite similar to confluence of a relation → is the following:

Definition 4.1.1 (Diamond property)**.** *An element x has the diamond property with respect to relation → on set A iff $x \to y$ and $x \to z$ imply that there exists an element q with $y \to q$ and $z \to q$. The relation → itself has the diamond property, denoted Diamond(→), iff every element of A has the diamond property with respect to →.*

Pictorially, this is very similar to the diagram for confluence, but without the stars:

$$\begin{tikzcd}
& a \arrow[dl] \arrow[dr] & \\
b \arrow[dr, dashed] & & c \arrow[dl, dashed] \\
& d &
\end{tikzcd}$$

Indeed, an alternative definition of confluence of $\to$ is simply to say that $\to^*$ has the diamond property. In this form, the following lemma states that reflexive-transitive closure preserves the diamond property:

Theorem 4.1.2 (Star preserves diamond)**.** *Diamond*$(\to)$ *implies Diamond*$(\to^*)$

Proof. Assume *Diamond*$(\to)$ and x, y, z with $x \to^* y$ and $x \to^* z$. We proceed by induction on the derivation of $x \to^* y$ (recall the three rules defining the reflexive-transitive closure, in Figure 2.10):

Case :

$$\frac{x \to y}{x \to^* y}$$

Here we will proceed by an inner induction on the derivation of $x \to^* z$ to show that there is a q with $y \to^* q$ and $z \to q$, for all x, y, and z with $x \to y$.

Case (inner):

$$\frac{x \to z}{x \to^* z}$$

We have exactly the assumptions of the diamond property, so we can conclude that there is a q with $y \to q$ and $z \to q$. Our inner induction requires us to show $y \to^* q$, which follows from this same inclusion rule whose inferences we are presently considering.

Case (inner):

$$\frac{}{x \to^* x}$$

We have learned in this case that $x = z$, since that is the only way the reflexivity rule could be applied to prove $x \to^* z$. For whatever q we select, we must prove $y \to^* q$ and $z \to q$. Since $x = z$, it suffices to show $y \to^* q$ and $x \to q$. Let us take q to be y. So we must show $y \to^* y$, which follows by the reflexivity rule; and $x \to y$, which follows by assumption in this inner induction.

Case (inner):

$$\frac{x \to^* w \quad w \to^* z}{x \to^* z}$$

We may apply the induction hypothesis to the first premise of this inference. So the x, y, and z of the induction hypothesis are instantiated with x, y, and w, respectively. The induction hypothesis then tells us that there is an element q such that $y \to^* q$ and $w \to q$. We may now apply the induction hypothesis to the second premise, where we instantiate x, y, and z with w, q, and z, respectively. This tells us that there is a q' with $q \to^* q'$ and $z \to q'$. Combining a couple of the facts we have so far (namely, $y \to^* q$ and $q \to^* q'$) using the transitivity rule gives us $y \to^* q'$. And we have $z \to q'$. So we may take the element q which we are supposed to identify, to be this q'. This concludes the inner induction. Note that this induction could be (and often is) broken out as a separate lemma, since it does not need to invoke the outer induction hypothesis.

We may now return to our outer induction:

Case :

$$\frac{}{x \to^* x}$$

In this case, we have learned that $x = y$, since that is the only way a reflexivity inference could prove $x \to^* y$. Our goal is to identify an element q with $y \to^* q$ and $z \to^* q$, but since $x = y$, it suffices to find a q with $x \to^* q$ and $z \to^* q$. Take q to be z, and we have $x \to^* z$ by assumption of this induction, and $z \to^* z$ by reflexivity.

Case :

$$\frac{x \to^* w \quad w \to^* y}{x \to^* y}$$

By the induction hypothesis applied to the first premise, there is a q with $w \to^* q$ and $z \to^* q$. We may now apply the induction hypothesis to the second premise, instantiating x, y, and z with w, y, and q, respectively. This produces a q' with $y \to^* q'$ and $q \to^* q'$. Let us take q' to be the q required by the theorem. Since $z \to^* q$ and $q \to^* q'$, we have $z \to^* q'$ by transitivity; and $y \to^* q'$ was already concluded.

□

Thanks to Theorem 4.1.2, we know that if we would like to establish confluence of $\hookrightarrow$, it would suffice to prove that this relation has the diamond property. But this is easily seen not to be the case. Consider the diagram in Figure 4.1. The term at the peak (top) of the diagram has two redexes, shown underlined. Down the left side of the peak we reduce the leftmost redex, and down the right side, the rightmost. We can indeed join the resulting terms, at term $z\ z$, but this requires two steps along the left side of the valley (running diagonally right to $z\ z$), while needing just one step along the right side of the valley. This example shows:

Theorem 4.1.3. $\hookrightarrow$ *lacks the diamond property*

It is the beautiful observation at the heart of the Tait–Martin-Löf proof of confluence that while β-reduction lacks the diamond property, still another relation $\Rightarrow^\alpha$ can be defined which satisfies the diamond property, and where $(\Rightarrow^\alpha)^* = \leadsto^*$. This will lead us to our goal, thanks to the following theorem:

Theorem 4.1.4. *Let S be a relation on a set A, and suppose that there is a relation R on A such that Diamond(R) and $R^* = S^*$. Then S is confluent.*

Proof. Confluence of S, is equivalent to *Diamond*(S^*). Since $R^* = S^*$ by assumption, it suffices then to prove *Diamond*(R^*). By Theorem 4.1.2, this follows from *Diamond*(R), which we have by assumption. □

Actually, the task of confluence is made even easier by observing the following:

Lemma 4.1.5. *If $S \subseteq R \subseteq S^*$, then $R^* = S^*$.*

Proof. By monotonicity of the reflexive-transitive closure operator (Lemma 2.6.2), $S \subseteq R$ implies $S^* \subseteq R^*$. So we have inclusions both directions between S^* and R^*, implying (in set theory) that they are equal. □

So to use Theorem 4.1.4, by this lemma, we just need to identify some relation intermediate between $\hookrightarrow$ and $\hookrightarrow^*$, which satisfies the diamond property. This relation is $\Rightarrow^\alpha$, defined next.

4.2 Parallel reduction

In this section, we define a relation $\Rightarrow^\alpha$ for parallel reduction with renaming. This relation will be the desired intermediate relation between $\hookrightarrow$ and $\hookrightarrow^*$, with the diamond property. $\Rightarrow$ is defined by the rules of Figure 4.2 (i.e., $\Rightarrow$ is the set of pairs (t, t') such that $t \Rightarrow t'$ has a finite derivation using those rules). It allows reduction, in a single step, of any subset of the redexes of the first related term, to obtain the second. One could reduce a single redex, several redexes (even nested ones), or no redexes at all.

We then define $\Rightarrow^\alpha$ from $\Rightarrow$ similarly to the way we defined $\leadsto$ from $\leadsto_\beta$ (Definition 2.8.2):

Definition 4.2.1. $\Rightarrow^\alpha$ *is* $=_\alpha\Rightarrow$.

$$\begin{tikzcd}
 & \underline{(\lambda\, x.\, x\; x)\; \underline{((\lambda\, y.\, y)\; z)}} \arrow[dl] \arrow[dr] & \\
((\lambda\, y.\, y)\; z)\; ((\lambda\, y.\, y)\; z) \arrow[d] & & (\lambda\, x.\, x\; x)\; z \arrow[ddl] \\
z\; ((\lambda\, y.\, y)\; z) \arrow[dr] & & \\
 & z\; z &
\end{tikzcd}$$

Figure 4.1: Counterexample showing that β-reduction lacks the diamond property.

$$\frac{}{x \Rightarrow x} \qquad \frac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'} \qquad \frac{t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{t_1\; t_2 \Rightarrow t_1'\; t_2'} \qquad \frac{t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{(\lambda\, x.\, t_1)\; t_2 \Rightarrow [t_2'/x]t_1'}$$

Figure 4.2: The definition of parallel reduction

So one takes an α-equivalence step, then a $\Rightarrow$ step.

Intuitively, we can see that $\Rightarrow$ is intermediate between $\rightsquigarrow_\beta$ and $\rightsquigarrow_\beta^*$, because it allows reducing any subset of the term's redexes. Critically, though, it does not allow reduction of *created redexes*. These are redexes that appear when a variable is replaced by a λ-abstraction. For example, we have the reduction

$$(\lambda\, x.\, x\; y)\; \lambda\, z.\, z \;\rightsquigarrow_\beta (\lambda\, z.\, z)\; y$$

The latter term is a created redex, because there is no corresponding redex in the starting term, from which it is derived.

4.2.1 Examples

Here are some examples of parallel reduction.

1. Here are all the terms to which $(\lambda\, x.\, x\; x)\; ((\lambda\, y.\, y)\; z)$ parallel reduces:

   (a) $(\lambda\, x.\, x\; x)\; ((\lambda\, y.\, y)\; z)$, which is the starting term itself;

   (b) $(\lambda\, x.\, x\; x)\; z$, where the right redex in the starting term has been contracted;

   (c) $((\lambda\, y.\, y)\; z)\; ((\lambda\, y.\, y)\; z)$, where the left redex has been contracted; and

   (d) $z\; z$, where both redexes have been contracted.

   The derivation of the parallel reduction to the last of these is (for an example):

$$\frac{\dfrac{\dfrac{}{x \Rightarrow x} \quad \dfrac{}{x \Rightarrow x}}{x\; x \Rightarrow x\; x} \quad \dfrac{\dfrac{}{y \Rightarrow y} \quad \dfrac{}{z \Rightarrow z}}{(\lambda\, y.\, y)\; z \Rightarrow z}}{(\lambda\, x.\, x\; x)\; ((\lambda\, y.\, y)\; z) \Rightarrow z\; z}$$

2. We have the following parallel reduction:

$$(\lambda\, x.\, \lambda\, y.\, x\; y)\; (\lambda\, x.\, \lambda\, y.\, x\; y) \Rightarrow \lambda\, y.\, (\lambda\, x.\, \lambda\, y.\, x\; y)\; y$$

This same starting term requires α-equivalence to reduce further:

$$\begin{array}{ll}
\underline{(\lambda\, x.\, \lambda\, y.\, x\ y)\ (\lambda\, x.\, \lambda\, y.\, x\ y)} & \rightsquigarrow_\beta \\
\lambda\, y.\, (\lambda\, x.\, \underline{\lambda\, y.\, x\ y})\ y & \rightsquigarrow_a \\
\lambda\, y.\, \underline{(\lambda\, x.\, \lambda\, w.\, x\ w)\ y} & \rightsquigarrow_\beta \\
\lambda\, y.\, \lambda\, w.\, y\ w &
\end{array}$$

But with parallel reduction, as long as all bound variables are distinct from the free variables, α-equivalence is not required for a single $\Rightarrow$-step. And a single $\Rightarrow$-step cannot reduce the starting term of this example past where the next redex is created.

4.2.2 Basic properties of parallel reduction

Lemma 4.2.2. *$t \Rightarrow t$ for all terms t.*

Proof. The proof is by induction on t.

Case : t is a variable x. Then we may construct this derivation:

$$\frac{}{x \Rightarrow x}$$

Case : t is an application $t_1\ t_2$ for some t_1 and t_2. Then we may construct:

$$\frac{\dfrac{}{t_1 \Rightarrow t_1}\ \mathit{IH} \qquad \dfrac{}{t_2 \Rightarrow t_2}\ \mathit{IH}}{t_1\ t_2 \Rightarrow t_1\ t_2}$$

Case : t is a λ-abstraction $\lambda\, x.\, t'$ for some x and t'. Then we construct:

$$\frac{\dfrac{}{t' \Rightarrow t'}\ \mathit{IH}}{\lambda\, x.\, t' \Rightarrow \lambda\, x.\, t'}$$

□

Lemma 4.2.3. $\hookrightarrow\ \subseteq\ \Rightarrow^\alpha$.

Proof. It suffices to assume $t \hookrightarrow t'$ for some t and t', and show $t \Rightarrow^\alpha t'$. Since $\hookrightarrow$ is defined (Definition 4.0.1) to be $\rightsquigarrow_\alpha \cup \rightsquigarrow_\beta$, let us consider each possibility. If we have $t \rightsquigarrow_\alpha t'$, then we have $t \Rightarrow^\alpha t'$, because

1. $t =_\alpha t'$, and
2. $t' \Rightarrow t'$, by Lemma 4.2.2

So we get $t =_\alpha t' \Rightarrow t'$, which suffice to prove $t \Rightarrow^\alpha t'$ (by definition of relational composition).

Now suppose we have $t \rightsquigarrow_\beta t'$. Then the proof is by induction on the derivation of $t \rightsquigarrow_\beta t'$ (via the rules of Figure 2.5).

Case :

$$\frac{}{(\lambda\, x.\, t)\ t' \rightsquigarrow_\beta [t'/x]t}$$

We may construct the following, where we invoke Lemma 4.2.2 where indicated, so that we can limit the β-reduction rule for $\Rightarrow$ just to this redex $(\lambda\, x.\, t)\ t'$:

$$\frac{\dfrac{}{t \Rightarrow t}\ 4.2.2 \qquad \dfrac{}{t' \Rightarrow t'}\ 4.2.2}{(\lambda\, x.\, t)\ t' \Rightarrow [t'/x]t}$$

Case :

$$\frac{t \rightsquigarrow_\beta t'}{\lambda\, x.\, t \rightsquigarrow_\beta \lambda\, x.\, t'}$$

We construct:

$$\dfrac{\dfrac{t \ \leadsto_\beta \ t'}{t \Rightarrow t'}\ \mathit{IH}}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'}$$

Case :

$$\dfrac{t_1 \ \leadsto_\beta \ t_1'}{t_1\ t_2 \ \leadsto_\beta \ t_1'\ t_2}$$

Construct the following, again invoking Lemma 4.2.2:

$$\dfrac{\dfrac{t_1 \ \leadsto_\beta \ t_1'}{t_1 \Rightarrow t_1'}\ \mathit{IH} \qquad \dfrac{}{t_2 \Rightarrow t_2}\ \mathit{4.2.2}}{t_1\ t_2 \ \Rightarrow \ t_1'\ t_2}$$

Case :

$$\dfrac{t_2 \ \leadsto_\beta \ t_2'}{t_1\ t_2 \ \leadsto_\beta \ t_1\ t_2'}$$

Similarly to the previous case, construct:

$$\dfrac{\dfrac{}{t_1 \Rightarrow t_1}\ \mathit{4.2.2} \qquad \dfrac{t_2 \ \leadsto_\beta \ t_2'}{t_2 \Rightarrow t_2'}\ \mathit{IH}}{t_1\ t_2 \ \Rightarrow \ t_1\ t_2'}$$

□

Lemma 4.2.4. $\Rightarrow^\alpha \subseteq \hookrightarrow^*$

Proof. Assume $t \Rightarrow^\alpha t'$. This implies that there exists some t'' with

$$t =_\alpha t'' \Rightarrow t'$$

Since $\hookrightarrow$ includes $\leadsto_\alpha$ by definition, we have $t \hookrightarrow^* t''$. So it suffices to show $t'' \hookrightarrow^* t'$. In fact, we will show the stronger property $t'' \leadsto_\beta^* t'$. This is done by induction on the assumed derivation of $t'' \Rightarrow t'$.

Case :

$$\dfrac{}{x \Rightarrow x}$$

We construct:

$$\dfrac{}{x \leadsto_\beta^* x}$$

Case :

$$\dfrac{t \Rightarrow t'}{\lambda\, x.\, t \Rightarrow \lambda\, x.\, t'}$$

We construct the following, making use of Lemma 2.6.7 which implies that the compatible closure of $(\leadsto_\beta^*)$ is the same as just $\leadsto_\beta^*$. So the inference at the bottom of the derivation, using one of the rules for the compatible closure (Figure 2.8), is actually a legal inference for concluding a $\leadsto_\beta^*$ step:

$$\dfrac{\dfrac{t \Rightarrow t'}{t \leadsto_\beta^* t'}\ \mathit{IH}}{\lambda\, x.\, t \leadsto_\beta^* \lambda\, x.\, t'}\ 2.6.7$$

Case :

$$\dfrac{t_1 \Rightarrow t_1' \qquad t_2 \Rightarrow t_2'}{t_1\ t_2 \Rightarrow t_1'\ t_2'}$$

We construct the following, again applying Lemma 2.6.7, and ending with transitivity:

$$\dfrac{\dfrac{\dfrac{t_1 \Rightarrow t_1'}{t_1 \rightsquigarrow_\beta^* t_1'}\;\mathit{IH}}{t_1\ t_2 \rightsquigarrow_\beta^* t_1'\ t_2}\;2.6.7 \qquad \dfrac{\dfrac{t_2 \Rightarrow t_2'}{t_2 \rightsquigarrow_\beta^* t_2'}\;\mathit{IH}}{t_1'\ t_2 \rightsquigarrow_\beta^* t_1'\ t_2'}\;2.6.7}{t_1\ t_2 \rightsquigarrow_\beta^* t_1'\ t_2'}$$

<u>Case :</u>

$$\dfrac{t_1 \Rightarrow t_1' \quad t_2 \Rightarrow t_2'}{(\lambda\, x.\, t_1)\ t_2 \Rightarrow [t_2'/x]t_1'}$$

We construct the following, again applying Lemma 2.6.7, and ending with two uses of transitivity. We assemble proofs about the reductions of t_1 and t_2, and finish off with a β-inference.

$$\dfrac{\dfrac{\dfrac{\dfrac{t_1 \Rightarrow t_1'}{t_1 \rightsquigarrow_\beta^* t_1'}\;\mathit{IH}}{\lambda\, x.\, t_1 \rightsquigarrow_\beta^* \lambda\, x.\, t_1'}\;2.6.7}{(\lambda\, x.\, t_1)\ t_2 \rightsquigarrow_\beta^* (\lambda\, x.\, t_1')\ t_2}\;2.6.7 \qquad \dfrac{\dfrac{\dfrac{t_2 \Rightarrow t_2'}{t_2 \rightsquigarrow_\beta^* t_2'}\;\mathit{IH}}{(\lambda\, x.\, t_1')\ t_2 \rightsquigarrow_\beta^* (\lambda\, x.\, t_1')\ t_2'}\;2.6.7 \quad \dfrac{\dfrac{\dfrac{}{(\lambda\, x.\, t_1')\ t_2'\ \beta\ [t_2'/x]t_1'}}{(\lambda\, x.\, t_1')\ t_2' \rightsquigarrow_\beta [t_2'/x]t_1'}}{(\lambda\, x.\, t_1')\ t_2' \rightsquigarrow_\beta^* [t_2'/x]t_1'}}{(\lambda\, x.\, t_1')\ t_2 \rightsquigarrow_\beta^* [t_2'/x]t_1'}\;2.6.7}{(\lambda\, x.\, t_1)\ t_2 \rightsquigarrow_\beta^* [t_2'/x]t_1'}$$

□

4.2.3 Parallel reduction has the diamond property

In this section, we show *Diamond*$(\Rightarrow^\alpha)$. In general, the diamond property (Definition 4.1.1) for a relation follows from the following stronger property:

Definition 4.2.5. *An element $x \in A$ has the triangle property with respect to relation $\rightarrow$ on set A iff there exists $x' \in A$ such that for every $y \in A$ with $x \rightarrow y$, we have $y \rightarrow x'$. The relation $\rightarrow$ has the triangle property, denoted Triangle$(\rightarrow)$, iff every element of A has the triangle property with respect to $\rightarrow$.*

The crucial point here is that for each a, there exists an a' independent of the elements to which a is related. The triangle property is stronger than the diamond property, as is easily seen by a small example:

Example 4.2.6. *Suppose A is $\{1, 2, 3, 4, 5, 6, 7\}$, and we have relation $\rightarrow$ satisfying exactly these facts:*

- $1 \rightarrow 2$
- $1 \rightarrow 3$
- $1 \rightarrow 4$
- $2 \rightarrow 5$
- $3 \rightarrow 5$
- $2 \rightarrow 6$
- $4 \rightarrow 6$
- $3 \rightarrow 7$
- $4 \rightarrow 7$

The element 1 has the diamond property, because it is related to 2, 3, and 4, which can all be joined. But since they are joined at different points (2 and 3 at 5, but 2 and 4 at 6), it does not have the triangle property with respect to $\rightarrow$. To have the triangle property, 1, 2, and 3 would all have to be joined at a single point.

Lemma 4.2.7. *Triangle*$(\to)$ *implies Diamond*$(\to)$.

Proof. To prove the diamond property, assume a, b, and c with $a \to b$ and $a \to c$. By the triangle property, there exists a' such that $b \to a'$ and $c \to a'$. (Again, note that the critical point is that this a' is determined solely from a, not b or c.) This a' is then the joining element required by the diamond property. □

Theorem 4.2.8. *Triangle*$(\Rightarrow^{\alpha})$.

Proof. Assume t, and let t' be any term α-equivalent to t but where all bound variables are distinct from each other and the free variables of t. Now □

4.3 Exercises

4.3.1 Confluent terms

In the following problems, terms t, t_1, and t_2 are given, such that $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$. Find a term t' such that $t_1 \rightsquigarrow^* t'$ and $t_2 \rightsquigarrow^* t'$. You do not have to write out any reduction sequences. Please just give the term t'. For fun, you can try to find the minimal such term t', viewing $\rightsquigarrow$ as an ordering (so try to find t' where there is no other t'' satisfying the same property and having $t'' \rightsquigarrow^* t'$) – but this is optional.

1. $$\begin{array}{ll} t: & (\lambda\, x.\, x\ (x\ x))\ ((\lambda\, y.\, y)\ z) \\ t_1: & z\ (((\lambda\, y.\, y)\ z)\ ((\lambda\, y.\, y)\ z)) \\ t_2: & ((\lambda\, y.\, y)\ z)\ (z\ ((\lambda\, y.\, y)\ z)) \end{array}$$

2. Let U abbreviate $\lambda\, x.\, \mathit{true}\ (x\ x)$. Recall the definition of the Y combinator from Section 3.7, and *true* from Section 3.4.
$$\begin{array}{ll} t: & Y\ \mathit{true} \\ t_1: & \mathit{true}\ (U\ U) \\ t_2: & \lambda\, y.\, \mathit{true}\ (U\ U) \end{array}$$

3. This problem again uses the Y combinator; recall also *false* from Section 3.4. Let U abbreviate $\lambda\, x.\mathit{false}\ (x\ x)\ (\mathit{false}\ (x\ x))$.
$$\begin{array}{ll} t: & Y\ (\lambda\, u.\mathit{false}\ u\ (\mathit{false}\ u)) \\ t_1: & \mathit{id}\ (\mathit{false}\ (U\ U)) \\ t_2: & \mathit{false}\ (U\ U)\ \mathit{id} \end{array}$$

4. This problem uses just $\rightsquigarrow_\alpha$ steps:
$$\begin{array}{ll} t: & \lambda\, x.\, \lambda\, y.\, x\ \lambda\, z.\, y \\ t_1: & \lambda\, x.\, \lambda\, z.\, x\ \lambda\, y.\, z \\ t_2: & \lambda\, y.\, \lambda\, z.\, y\ \lambda\, y.\, z \end{array}$$

4.3.2 Confluence

1. Because of our explicit treatment of variable-renaming, single-step β-reduction with renaming ($\rightsquigarrow$, Definition 2.8.2) is not confluent. To show this, give an example of a term t and distinct normal forms t_1 and t_2 where $t \rightsquigarrow^* t_1$ and $t \rightsquigarrow^* t_2$.

2. For each of the following relations, argue briefly why it does or does not have the diamond property:

   - α (Figure 2.11)
   - β (Figure 2.9)
   - $\mathcal{T}[\alpha]$ (Figure 2.8)

4.3.3 Parallel reductions

1. For each of the following terms, write out all the terms to which they parallel reduce in one step.

   (a) $\lambda\, x.\, (\lambda\, y.\, y\; y)\; ((\lambda\, z.\, x)\; x)$

   (b) $(\lambda\, x.\, \lambda\, y.\, y\; y)\; ((\lambda\, z.\, z)\; x)\; \lambda\, w.\, w$

2. Let us define a family I_n of terms by recursion on $n \in \mathbb{N}$ (recall that *id* is $\lambda\, x.\, x$):

   $$\begin{array}{lcl} I_0 & = & \mathit{id} \\ I_{n+1} & = & I_n\; I_n \end{array}$$

   So I_2, for example, is *id id* $(\mathit{id}\ \mathit{id})$. Prove by induction on n that $I_{n+1} \Rightarrow I_n$.

3. Give an example of a term t such that $t\; t$ is normalizing but there is no normal term t' such that $t\; t \Rightarrow t'$.

Part II

Typed Lambda Calculus

Chapter 5

Simply Typed Lambda Calculus

5.1 Syntax for types

We assume a non-empty set B of base types. These are just any mathematical objects we wish, that will play the role of atomic (indivisible) types. We will use b as a meta-variable for elements of type B. Similarly as for our metavariables for λ-calculus variables (see the start of Section 2.1), we will adopt the convention that different meta-variables refer to different base types, in any particular meta-linguistic discussion. The syntax of types is then:

$$\textit{simple types } T \ ::= \ b \mid T \to T'$$

There is one parsing convention for simple types, which is that arrow is right-associative. So a type like $a \to b \to c$ should be parsed as $a \to (b \to c)$.

Let us consider some examples of simple types. We might have the type *bool* → *bool* for boolean negation, and other unary (1-argument) boolean operations. Similarly, a type like *bool* → *bool* → *bool* could describe conjunction, disjunction, and any other binary boolean operations. For a higher-order example, a type like (*nat* → *bool*) → *nat* could be the type for a minimization function *minimize*, where *minimize* p returns the smallest natural number n such that $p\ n$ returns *true*.

Now, it will happen that our notion of typing will not allow interesting computations with values of atomic types like *bool*. So we will not actually be able to type functions like the ones just described in pure simply typed lambda calculus (STLC). But STLC is the right framework for characterizing the functional behavior (via arrow types $T \to T'$) of lambda terms, and thus forms the core of most other more advanced type systems, including ones where types like *bool* are definable within the system.

5.2 Realizability semantics of types

One very natural way to understand a type is as a specification of the behavior of programs. For example, in a programming language like Java, suppose a function is declared with the signature

```
  int f(int x, int y);
```

Then intuitively, the meaning of this is that function `f` expects two integers `x` and `y` as input and, if it terminates normally (without raising an exception, diverging, etc.), then it will return an integer as output.

This idea that a type is a form of specification for programs can be made precise for STLC using a form of *realizability semantics*. This semantics was introduced by Kleene for intuitionistic arithmetic [10]. To explain this further, we first need this notion:

Definition 5.2.1 (β-expansion closed)**.** *A set S of closed terms is β-expansion closed if $t \in S$ and $t' \rightsquigarrow t$ imply $t' \in S$, for closed t'.*

$$\begin{array}{lcl} \llbracket b \rrbracket & = & I(b) \\ \llbracket T_1 \to T_2 \rrbracket & = & \{t \in \mathit{ClosedTerms} \mid \forall\, t' \in \llbracket T_1 \rrbracket .\, (t\; t') \in \llbracket T_2 \rrbracket\} \end{array}$$

Figure 5.1: Realizability semantics of types, with respect to an assignment I of meanings for base types

Such a set S is closed under β-expansion in the sense that one cannot leave S by following β-expansion steps to closed terms.

Figure 5.1 defines an interpretation $\llbracket T \rrbracket$ for any simple type T, assuming a function I which interprets the base types of B. We require that $I(b)$ is a β-expansion closed set, for every $b \in B$.

The values computed by the semantic function and I are sets of closed terms. So mathematically, writing *Types* for the set of all simple types and *ClosedTerms* for the set of all closed terms of untyped λ-calculus (and using the standard notation $\mathcal{P} S$ for the set of all subsets of a set S), we have:

- $\llbracket - \rrbracket \in \mathit{Types} \to \mathcal{P}\ \mathit{ClosedTerms}$
- $I \in B \to \mathcal{P}\ \mathit{ClosedTerms}$

Given that $I(b)$ is required to be β-expansion closed, we have:

Lemma 5.2.2. *$\llbracket T \rrbracket$ is β-expansion closed, for all T.*

Proof. The proof is by induction on T. If T is a base type b, then $\llbracket T \rrbracket = I(b)$, which is β-expansion closed by assumption. So assume T is an arrow type $T_1 \to T_2$, and assume $t \in \llbracket T \rrbracket$, and closed $t' \leadsto^* t$. We must show $t' \in \llbracket T \rrbracket$. For that, assume $t'' \in \llbracket T_1 \rrbracket$, and show $t'\ t'' \in \llbracket T_2 \rrbracket$. By the induction hypothesis, $\llbracket T_2 \rrbracket$ is β-expansion closed. So to show $t'\ t'' \in \llbracket T_2 \rrbracket$, it suffices to show that $t\, t'' \in \llbracket T_2 \rrbracket$, since $t'\ t''$ reduces to $t\, t''$ (since $t' \leadsto^* t$). But $t\, t'' \in \llbracket T_2 \rrbracket$ because $t \in \llbracket T_1 \to T_2 \rrbracket$ by assumption (and $t'' \in \llbracket T_1 \rrbracket$). □

Recall that the notation $t \downarrow$ (Definition 2.8.6) means that term t is normalizing: there exists some t' such that $t \leadsto^* t'$ and t' is in normal form (i.e., does not reduce to any term).

Definition 5.2.3. *Define $\mathcal{N}$ to be $\{t \in \mathit{ClosedTerms} \mid t \downarrow\}$.*

Lemma 5.2.4. *If $I(b) = \mathcal{N}$ for all $b \in B$, then $\llbracket T \rrbracket$ is non-empty and a subset of $\mathcal{N}$ for all T.*

Proof. The proof is by induction on T. For the base case, suppose T is some $b \in B$. Then $\llbracket T \rrbracket = \llbracket b \rrbracket = I(b) = \mathcal{N}$. And this set contains an element, for example $\lambda\, x.\, x$. For the step case, suppose T is a function type $T_1 \to T_2$, and assume $t \in \llbracket T \rrbracket$. We must show $t \downarrow$. Since $\llbracket T_1 \rrbracket$ is non-empty by induction hypothesis, we may select some $t' \in \llbracket T_1 \rrbracket$. Then by the semantics of arrow types, $t\, t' \in \llbracket T_2 \rrbracket$. By the induction hypothesis (for T_2), this implies that $t\, t' \downarrow$. In turn, this implies $t \downarrow$ (proof omitted). For an element of $\llbracket T_1 \to T_2 \rrbracket$, there exists $t \in \llbracket T_2 \rrbracket$ by the induction hypothesis. We then have $\lambda\, x.\, t \in \llbracket T_1 \to T_2 \rrbracket$, by the semantics of arrow types. □

5.2.1 Examples

Let us see some examples of the semantics. Suppose that B consists of two base types, b and b'. Let us write $\mathbb{B}$ for the set of Church-encoded booleans (Section 3.2), and $\mathbb{N}$ for the set of Church-encoded natural numbers. Then define I by:

$$\begin{array}{lcl} I(b) & := & \{t \in \mathit{ClosedTerms} \mid \exists\, b \in \mathbb{B}.\, t \leadsto^* b\} \\ I(b') & := & \{t \in \mathit{ClosedTerms} \mid \exists\, n \in \mathbb{N}.\, t \leadsto^* n\} \end{array}$$

Example. First, we can observe that $\mathit{not} \in \llbracket b \to b \rrbracket$. To show this, it suffices to assume an arbitrary closed t' with $t' \leadsto^* \mathbb{B}$, and show that $\mathit{not}\ t' \leadsto^* \mathbb{B}$. Suppose $t' \leadsto^* \mathit{true}$. Then we have

$$\mathit{not}\ t' \ \leadsto^*\ \mathit{not}\ \mathit{true} \ \leadsto^*\ \mathit{false}$$

And similarly, if closed $t' \leadsto^* \mathit{false}$, we have

$$\mathit{not}\ t' \ \leadsto^*\ \mathit{not}\ \mathit{false} \ \leadsto^*\ \mathit{true}$$

Example. Next, let us define this function:

$$\mathit{even} \ := \ \lambda\, x.\, x \ \mathit{not}\ \mathit{true}$$

Given a Church-encoded natural number n, this function iterates boolean negation n times starting with *true*. This will result in *true* iff n is indeed even. Let us prove that $\mathit{even} \in [\![b' \to b]\!]$. Assume $t \in [\![b']\!]$ and show $\mathit{even}\ t \in [\![b]\!]$. Since $t \in [\![b']\!]$, there is some natural number n such that

$$t \rightsquigarrow^* \lambda\, f.\, \lambda\, x.\, \underbrace{f \cdots (f}_{n}\ x)$$

Then we have

$$\mathit{even}\ t \ \approx \ t\ \mathit{not}\ \mathit{true} \ \approx \ \underbrace{\mathit{not} \cdots (\mathit{not}}_{n}\ \mathit{true})$$

We may easily prove that the latter term is β-equivalent to *true* if n is even, and to *false* if n is odd.

Example. Suppose there is a base type $b \in B$, and define $I(b)$ to be $\mathcal{N}$. Then $\lambda\, x.\, \lambda\, y.\, x \in [\![b \to b]\!]$. To prove this, it suffices by the semantics of arrow types to assume $t' \in [\![b]\!]$, and show $(\lambda\, x.\, \lambda\, y.\, x)\ t' \in [\![b]\!]$. Since $[\![b]\!] = I(b)$, we are assuming closed $t' \downarrow$, and need to show $(\lambda\, x.\, \lambda\, y.\, x)\ t' \downarrow$. For the latter:

$$\underline{(\lambda\, x.\, \lambda\, y.\, x)\ t'} \rightsquigarrow \lambda\, y.\, t'$$

and the latter is normalizing (and closed) since t' is. Also, $\lambda\, x.\, \lambda\, y.\, x \in [\![b]\!]$, since $\lambda\, x.\, \lambda\, y.\, x$ is in normal form (and hence normalizing), and closed. With this example, the same term is in $[\![b]\!]$ and $[\![b \to b]\!]$. Let us next consider an example (with the same interpretation I for the base type b) where we have a term in $[\![b]\!]$ that is not in $[\![b \to b]\!]$.

Example. We have $\lambda\, x.\, x\ x \in [\![b]\!]$, since $\lambda\, x.\, x\ x$ is in normal form and closed. But $\lambda\, x.\, x\ x \notin [\![b \to b]\!]$. To prove that, we must exhibit $t \in [\![b]\!]$ such that $(\lambda\, x.\, x\ x)\ t$ is not in $[\![b]\!]$. We may use $\lambda\, x.\, x\ x$ for that t, because we just observed that it is in $[\![b]\!]$, but $(\lambda\, x.\, x\ x)\ (\lambda\, x.\, x\ x)$ (i.e., the term Ω) is definitely not in $[\![b]\!]$, since Ω is not normalizing.

Example. Finally, let us change the interpretation $I(b)$ to be $\{t \in \mathit{ClosedTerms} \mid t \rightsquigarrow^* \ \lambda\, x.\, x\}$. Then we have an example opposite to the one we just found: a term in $[\![b \to b]\!]$ that is not in $[\![b]\!]$. The term is again $\lambda\, x.\, x\ x$. This term does not reduce to $\lambda\, x.\, x$, and so it is not in $[\![b]\!]$. But it is in $[\![b \to b]\!]$. To show that, assume $t \in [\![b]\!]$, and show $(\lambda\, x.\, x\ x)\ t \in [\![b]\!]$. Since $t \in [\![b]\!]$, we have $t \rightsquigarrow^* \ \lambda\, x.\, x$. Then we have the following reduction confirming that the starting term is in $[\![b]\!]$:

$$(\lambda\, x.\, x\ x)\ \underline{t} \rightsquigarrow^* \underline{(\lambda\, x.\, x\ x)\ \lambda\, x.\, x} \rightsquigarrow \underline{(\lambda\, x.\, x)\ \lambda\, x.\, x} \rightsquigarrow \lambda\, x.\, x$$

5.3 Type assignment rules

To obtain a computable approximation of the realizability semantics of the previous section, we use a system of rules for deriving facts of the form $\Gamma \vdash t : T$; such facts are called *typing judgments*. Here, Γ is a *typing context*, with the following syntax:

$$\mathit{typing\ contexts}\ \Gamma \ ::= \ \cdot \mid \Gamma, x : T$$

There is an empty context $\cdot$, and a context may be extended on the right with a binding $x : T$. This represents an assumption that x has type T. We will type open terms (terms with free variable occurrences) by making assumptions, in typing contexts, about the types of their free variables. The typing rules are in Figure 5.2.

5.3.1 Examples

An example typing derivation is given in Figure 5.3. Let us adopt the convention that we do not show derivations of *Find* judgments. Thus, we will allow derivations to terminate in applications of the variable rule (first rule in Figure 5.2) with premise elided, as long as that elided premise is actually derivable.

$$\frac{\mathit{Find}\ x : T\ \mathit{in}\ \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T' \vdash t : T}{\Gamma \vdash \lambda\, x.\, t : T' \to T} \qquad \frac{\Gamma \vdash t_1 : T' \to T \quad \Gamma \vdash t_2 : T'}{\Gamma \vdash t_1\ t_2 : T}$$

$$\frac{}{\mathit{Find}\ x : T\ \mathit{in}\ (\Gamma, x : T)} \qquad \frac{\mathit{Find}\ x : T\ \mathit{in}\ \Gamma \quad x \neq y}{\mathit{Find}\ x : T\ \mathit{in}\ (\Gamma, y : T')}$$

Figure 5.2: Type-assignment rules for simply typed lambda calculus, with rules for looking up a variable declaration in the context Γ

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{}{\Gamma \vdash x : b \to b \to b} \quad \cfrac{}{\Gamma \vdash y : b}}{\Gamma \vdash x\ y : b \to b} \qquad \cfrac{}{\Gamma \vdash y : b}}{\Gamma \vdash x\ y\ y : b}}{\cdot, x : b \to b \to b \vdash \lambda\, y.\, x\ y\ y : b \to b}}{\cdot \vdash \lambda\, x.\, \lambda\, y.\, x\ y\ y : (b \to b \to b) \to b \to b}$$

Figure 5.3: Example typing derivation in STLC, where Γ abbreviates the typing context $\cdot, x : b \to b \to b, y : b$

5.3.2 Soundness with respect to the realizability semantics

We have given two meanings for types, and it is now time to relate them. In this section, we will prove soundness of the typing rules (Figure 5.2) with respect to the realizability semantics (Figure 5.1).

In logic generally, suppose we have two ways of defining the meaning of some formulas, via sets S_1 and S_2 of formulas that are considered (by the two semantics respectively) to be true. Then S_1 is sound with respect to S_2 iff $S_1 \subseteq S_2$, and S_1 is complete with respect to S_2 iff $S_2 \subseteq S_1$. One way to think about this is as if S_1 consists of statements made by a person, and S_2 consists of statements that are true in reality. Then soundness means that the statements the person makes are, in fact, true. So S_1 (the set of statements the person makes) is a subset of S_2 (the statements that are really true). Completeness is the inverse of this: if a statement is really true (i.e., in S_2), then the person affirms it (i.e., it is in S_1). It is easy to be sound: one affirms nothing. It is also easy to be complete: one affirms everything. The ideal, which is more difficult or even impossible to achieve, depending on the logic, is to be both sound and complete.

To consider these properties for STLC, we need to formulate the formulas that our two semantics are affirming. The basic formula for typing is $\vdash t : T$. The corresponding formula for our realizability semantics is $t \in \llbracket T \rrbracket$. But the typing rules make use of typing contexts Γ and use more general formulas $\Gamma \vdash t : T$. So we will need a corresponding generalization of the formulas of our realizability semantics.

Definition 5.3.1. *Define $\llbracket \Gamma \rrbracket$ to be the set of substitutions σ such that whenever Find $x : T$ in Γ is derivable (with the rules of Figure 5.2), then $\sigma(x) \in \llbracket T \rrbracket$.*

So $\sigma \in \llbracket \Gamma \rrbracket$ means that the substitution σ maps variables to terms that are in the interpretations of the types that Γ says they have. Let $\sigma\ t$ denote the application of the substitution σ to t, with definition given in Figure 5.4. Since we are assuming that the range of σ consists of closed terms (since $\llbracket T \rrbracket$ is a set of closed terms for every T), we do not need to worry about variable capture: none is possible.

Theorem 5.3.2. *If $\Gamma \vdash t : T$, then for every $\sigma \in \llbracket \Gamma \rrbracket$, $\sigma\, t \in \llbracket T \rrbracket$, for all interpretations I where $I(b)$ is β-expansion closed.*

Proof. The proof is by induction on the typing derivation.

Case :

$$\frac{\mathit{Find}\ x : T\ \mathit{in}\ \Gamma}{\Gamma \vdash x : T}$$

Assume $\sigma \in \llbracket \Gamma \rrbracket$. By definition of the interpretation of Γ, this means that $\sigma(x) \in \llbracket T \rrbracket$, and hence $\sigma\, x$, which equals $\sigma(x)$, is in $\llbracket T \rrbracket$ as required.

$$\begin{array}{lcl} \sigma\, x & = & \begin{cases} \sigma(x), \text{ if } x \in \mathit{dom}(\sigma) \\ x, \text{ otherwise} \end{cases} \\ \sigma\,(t_1\ t_2) & = & (\sigma\, t_1)\ (\sigma\, t_2) \\ \sigma\,(\lambda\, x.\, t) & = & \lambda\, x.\,((\sigma\backslash x)\, t) \end{array}$$

Figure 5.4: Applying a substitution of closed terms (thus avoiding danger of variable capture). $\sigma\backslash x$ is the function that is just like σ except that it does not map x to anything.

$$\begin{array}{lcl} \llbracket b \rrbracket & = & I(b) \\ \llbracket T_1 \to T_2 \rrbracket & = & \{(t_1, t_2) \mid \forall\ (t', t'') \in \llbracket T_1 \rrbracket.\ (t_1\ t', t_2\ t'') \in \llbracket T_2 \rrbracket\} \end{array}$$

Figure 5.5: Relational semantics of types

Case :

$$\frac{\Gamma, x : T' \vdash t : T}{\Gamma \vdash \lambda\, x.\, t : T' \to T}$$

Assume $\sigma \in \llbracket \Gamma \rrbracket$. To show $\sigma\, \lambda\, x.\, t \in \llbracket T' \to T \rrbracket$, it suffices by the definition of substitution and the semantics of arrow types to assume $t' \in \llbracket T' \rrbracket$, and show that $(\lambda\, x.\, \sigma\, t)\ t'$ is in $\llbracket T \rrbracket$. Since $\llbracket T \rrbracket$ is β-expansion closed by Lemma 5.2.2, it suffices to show $[t'/x]\sigma\, t \in \llbracket T \rrbracket$.

Let σ' be the same as σ except that x is mapped to t'. Then $\sigma' \in \llbracket \Gamma, x : T' \rrbracket$, since $\sigma'(x) \in \llbracket T \rrbracket$. The desired conclusion now follows directly by the induction hypothesis. □

Corollary 5.3.3. *If* $\Gamma \vdash t : T$, *then* $t \downarrow$.

Proof. Let $I(b) = \mathcal{N}$ (Definition 5.2.3). Then by Theorem 5.3.2, for any $\sigma \in \llbracket \Gamma \rrbracket$, we have $\sigma t \in \llbracket T \rrbracket$. By Lemma 5.2.4, $\llbracket T \rrbracket \subseteq \mathcal{N}$, so $\sigma t \downarrow$. This implies $t \downarrow$ (proof omitted). □

5.4 Normalization using a well-founded ordering

In the previous section, we proved normalization for simply typed terms as a corollary of soundness of the typing rules with respect to our realizability semantics. We can give an interpretation for base types under which $\llbracket T \rrbracket$ is a set of normalizing terms, and then soundness (Theorem 5.3.2) does the rest. This semantic approach is a powerful way to prove normalization for other type systems, and indeed, once the type systems are powerful enough, it is essentially the only known way to obtain that result.

In this section, we pursue a different approach, in which we compute a measure for terms, and then show that we can always reduce a non-normal term in a way that reduces the corresponding measure. So we are showing that we can reduce terms in a way that makes this measure smaller. And the measure assigned to each term comes from a well-founded set, so it cannot decrease forever. Thus, each term has a finite reduction sequence.

5.5 Relational semantics

Realizability semantics (Section 5.2) interprets types as sets of terms. We may also interpret types as relations on terms. The definition is in Figure 5.5, where we assume now that we have $I \in (B \to \mathcal{P}\ (\mathit{Terms} \times \mathit{Terms}))$, and we then define $\llbracket - \rrbracket \in (\mathit{Types} \to \mathcal{P}\ (\mathit{Terms} \times \mathit{Terms}))$. The set $\mathcal{P}\ (\mathit{Terms} \times \mathit{Terms})$ is the set of all subsets of the cartesian product $\mathit{Terms} \times \mathit{Terms}$. Since such a subset is just a relation, we are interpreting base types and then types as relations on terms.

$$\frac{\textit{Find } F \textit{ in } S}{S \vdash F} \qquad \frac{S, F_1 \vdash F_2}{S \vdash F_1 \to F_2} \qquad \frac{S \vdash F_1 \to F_2 \qquad S \vdash F_1}{S \vdash F_2}$$

$$\frac{}{\textit{Find } F \textit{ in } (S, F)} \qquad \frac{\textit{Find } F \textit{ in } S}{\textit{Find } F \textit{ in } (S, F')}$$

Figure 5.6: Proof rules for minimal implicational logic, with rules for looking up an assumption in a list S of formulas

5.5.1 Examples

Suppose we have base types b and b', interpreted as just below. Recall that $t \uparrow$ means that t is not normalizing (Definition 2.8.7). The examples will also use some defined terms from Chapter 3: *false* for $\lambda\, x.\, \lambda\, y.\, y$, *id* for $\lambda\, x.\, x$, and Ω for the diverging term $(\lambda\, x.\, x\ x)\ \lambda\, x.\, x\ x$.

$$\begin{array}{lcl} I(b) & := & \{(t, t') \mid t \approx t'\} \\ I(b') & := & \{(t, t') \mid t \approx t' \approx \textit{false}\} \end{array}$$

Then we have the following relational facts:

- $\lambda\, x.\, x\ \Omega$ and $\lambda\, x.\, x\ \textit{id}$ are related by $\llbracket b' \to b \rrbracket$. To prove this using the semantics of Figure 5.5, we must assume we have terms t and t' which are related by $\llbracket b' \rrbracket$, and show that $(\lambda\, x.\, x\ \Omega)\ t$ is related to $(\lambda\, x.\, x\ \textit{id})\ t'$ by $\llbracket b \rrbracket$. Since $\llbracket b \rrbracket = I(b)$, the latter may be shown this way:

$$(\lambda\, x.\, x\ \Omega)\ t\ \approx\ (\lambda\, x.\, x\ \Omega)\ \textit{false}\ \approx\ \textit{false}\ \Omega\ \approx\ \textit{id}\ \approx\ \textit{false}\ \textit{id}\ \approx\ (\lambda\, x.\, x\ \textit{id})\ \textit{false}\ \approx\ (\lambda\, x.\, x\ \textit{id})\ t'$$

- That same pair of terms is not related by $\llbracket b \to b \rrbracket$, which we can show, by the semantics of Figure 5.5, by finding terms t and t' related by $\llbracket b \rrbracket$, but where $(\lambda\, x.\, x\ \Omega)\ t$ and $(\lambda\, x.\, x\ \textit{id})\ t'$ are not related by $\llbracket b \rrbracket$. Take t and t' both to be *id*, and we have:

$$(\lambda\, x.\, x\ \Omega)\ t\ =\ (\lambda\, x.\, x\ \Omega)\ \textit{id}\ \approx\ \Omega\ \neq_\beta\ \textit{id}\ \approx\ (\lambda\, x.\, x\ \textit{id})\ \textit{id}\ =\ (\lambda\, x.\, x\ \textit{id})\ t'$$

5.6 The Curry-Howard isomorphism

Curry observed the deep connection between typed lambda calculus and logic which, developed further by Howard, is known as the Curry-Howard isomorphism. The starting point is to connect STLC with minimal implicational logic. This logic is for proving formulas of the following form, where p is from some nonempty set P of atomic propositions:

$$F ::= p \mid F_1 \to F_2$$

This syntax is the same, disregarding the names of the metavariables, as that for simple types T (introduced at the start of Section 5.1). Figure 5.6 gives inference rules for deriving expressions of the form $S \vdash F$, where S is a list of formulas, taken as assumptions. These rules are (again, disregarding differences in the names of the meta-variables in question) exactly those of STLC, except without the terms.

Every STLC typing derivation can be translated to a derivation in minimal implicational logic, assuming that the set B of base types in STLC is translated to a subset of the set P of atomic propositions. For simplicity, in the following example, let us assume that $B \subseteq P$ (so the translation is the identity function). Then we may translate the derivation of Figure 5.3 to the proof in Figure 5.7. The derivation contains an unnecessary derivation of $S \vdash b$ from the top to about the middle of the overall derivation. It is unnecessary because we can already derive $S \vdash b$ just using the rule for assumptions (first rule of Figure 5.6). Does this mean the correspondence with the STLC derivation is somehow awry? Not at all. For we could just as well derive $\cdot \vdash \lambda\, x.\, \lambda\, y.\, y : (b \to b \to b) \to b \to b$ in STLC. The structure of the shorter proof in minimal implicational logic exactly mirrors this simpler lambda term.

Where one may be content to have proved a theorem without minding too much the details of the proof, in typed lambda calculus the term that corresponds to a different proof may be a computationally different function, as in the example just considered: $\lambda\, x.\, \lambda\, y.\, y$ behaves very differently, from a computational perspective, from $\lambda\, x.\, \lambda\, y.\, x\ y\ y$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\overline{S \vdash b \to b \to b} \quad \overline{S \vdash b}}{S \vdash b \to b} \qquad \overline{S \vdash b}}{S \vdash b}}{\cdot, b \to b \to b \vdash b \to b}}{\cdot \vdash (b \to b \to b) \to b \to b}$$

Figure 5.7: Example derivation in minimal implicational logic, where S abbreviates $b \to b \to b, b$

5.7 Exercises

5.7.1 Realizability semantics for types

1. Suppose B is $\{b_1, b_2, b_3\}$, and define I, recalling from Definition 2.8.6 that $t \downarrow$ means that t is normalizing:

$$\begin{array}{lcl} I(b_1) & = & \{\, t \mid \exists t'.\, t \ \rightsquigarrow^* \ t' \ \rightsquigarrow \ t' \,\} \\ I(b_2) & = & \{\, t \mid t \downarrow \} \\ I(b_3) & = & \{\, t \mid t \rightsquigarrow^* \ \lambda\, x.\, x \} \end{array}$$

Also, define the term t as follows:

$$t \ = \ \lambda\, f.\, (\lambda\, x.\, x\ x)\ (f\ \lambda\, x.\, x\ x)$$

(a) Prove that t is in $[\![b_2]\!]$.

(b) Prove that t is also in $[\![b_3 \to b_1]\!]$.

(c) Prove that t is also in $[\![(b_2 \to b_3) \to b_3]\!]$.

(d) Find a term t' that is in $[\![(b_3 \to b_2) \to b_2]\!]$ and also in $[\![b_1 \to b_1]\!]$; please explain why your term is in both those sets.

5.7.2 Type assignment rules

1. Write out typing derivations, using the rules of Figure 5.2, for the following typing judgments, assuming base types a, b, and c. You do not need to write out the derivations for the *Find* judgment for looking up typings of variables in the context.

(a) $\cdot, x : b, y : b \to b \vdash y\ (y\ x) : b$

(b) $\cdot \vdash \lambda\, x.\, \lambda\, y.\, x : a \to b \to a$

(c) $\cdot \vdash \lambda\, x.\, \lambda\, y.\, \lambda\, z.\, x\ z\ (y\ z) : (a \to b \to c) \to (a \to b) \to a \to c$

5.7.3 Relational semantics

1. Suppose we have a base type b, and let $I(b)$ be

$$\{(t, t') \mid (t\ t') \ \downarrow\}$$

Recall that $t \ \downarrow$ means that t is normalizing (Definition 2.8.6).

(a) Argue in detail that $\lambda\, x.\, \lambda\, y.\, x\ (y\ id)$ and $\lambda\, y.\, \lambda\, z.\, z\ y$ are related by $[\![b \to b]\!]$.

(b) Give another example of a pair of terms in $[\![b \to b]\!]$. Please argue in detail for membership in this relation.

5.7.4 Curry-Howard isomorphism

1. Translate the typing derivations you did in Section 5.7.2 above, into proofs in minimal implicational logic.

Bibliography

[1] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

[2] Hendrik Pieter Barendregt, Wil Dekkers, and Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[3] Felice Cardone and J. Roger Hindley. Lambda-calculus and combinators in the 20th century. In Dov M. Gabbay and John Woods, editors, *Logic from Russell to Church*, volume 5 of *Handbook of the History of Logic*, pages 723–817. North-Holland, 2009.

[4] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[5] Alonzo Church. A set of postulates for the foundation of logic (second paper). *Annals of Mathematics*, 34(4):839–864, 1933.

[6] Alonzo Church. *The Calculi of Lambda Conversion. (AM-6)*. Princeton University Press, 1941.

[7] Samuel Frontull, Georg Moser, and Vincent van Oostrom. α-avoidance. In Marco Gaboardi and Femke van Raamsdonk, editors, *8th International Conference on Formal Structures for Computation and Deduction, FSCD 2023, July 3-6, 2023, Rome, Italy*, volume 260 of *LIPIcs*, pages 22:1–22:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[8] J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, USA, 2 edition, 2008.

[9] J.R. Kennaway, J.W. Klop, M.R. Sleep, and F.J. de Vries. Infinitary lambda calculus. *Theoretical Computer Science*, 175(1):93 – 125, 1997.

[10] S. C. Kleene. On the interpretation of intuitionistic number theory. *The Journal of Symbolic Logic*, 10(4):109–124, 1945.

[11] Rodolphe Lepigre and Christophe Raffalli. Practical subtyping for Curry-style languages. *ACM Trans. Program. Lang. Syst.*, 41(1):5:1–5:58, February 2019.

[12] Rodolphe Lepigre and Christophe Raffalli. Practical subtyping for curry-style languages. *ACM Trans. Program. Lang. Syst.*, 41(1), feb 2019.

[13] Alfred North Whitehead and Bertrand Arthur William Russell. *Principia mathematica; 2nd ed.* Cambridge Univ. Press, Cambridge, 1927.

Index